Constraint satisfaction approach to product configuration with cost estimation

Wang, Lin

2013

Wang, L. (2013). Constraint satisfaction approach to product configuration with cost estimation. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/54837>

<https://doi.org/10.32657/10356/54837>

NANYANG
TECHNOLOGICAL
UNIVERSITY

CONSTRAINT SATISFACTION APPROACH TO PRODUCT CONFIGURATION WITH COST ESTIMATION

WANG LIN

SCHOOL OF COMPUTER ENGINEERING

2013

CONSTRAINT SATISFACTION APPROACH TO PRODUCT CONFIGURATION WITH COST ESTIMATION

WANG LIN

SCHOOL OF COMPUTER ENGINEERING

A thesis submitted to Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2013

Abstract

Due to the dynamism and heterogeneity of today's markets, mass customization has been widely applied in the manufacturing industry with the objective of satisfying individual customer needs while maintaining near mass production efficiency. Product family design is an effective approach for mass customization where problem modeling and solution are two significant concerns. As a large amount of data with diverse variety need to be represented (including production data, customer requirements and design knowledge), powerful knowledge representation models are necessary to capture the large variety and complexity of data. Given a configuration problem description and representation, efficient reasoning methods are also required for configuration solution search and optimization. Cost is a critical composition and criterion for product family design; however, the integration of cost estimation and product family design has not been successfully addressed.

In this thesis, an extended Dynamic Constraint Satisfaction Problem (DCSP) based product configuration and cost estimation system is proposed to effectively represent the large amount of data with great variety and complexity in product family design as DCSP terms (i.e., variables, domains and constraints). In the extended DCSP-based system, structural constraints are proposed to capture the inherent hierarchy among modules and components. Activation constraints are composed to represent derivation relationships among variables in the extended DCSP-based system. As both structural and activation constraints can activate variables into the DCSP according to triggering conditions (i.e., hierarchical and derivation relationships among variables), they are utilized to dynamically extend the search space of

the extended DCSP-based system such that only a subset of the original variables are involved in the final solution. To integrate cost estimation with product family design tightly, cost calculation constraints are introduced into the extended DCSP-based system, by which results of product configuration and cost estimation can be obtained simultaneously. Thus, cost can be utilized as a coherent criterion for product configuration. To study the impact of changes to cost drivers and design parameters on the cost of a product, the entire product family, or even the entire product lifecycle, cost variation constraint are proposed. Mapping relationships between concepts of the extended DCSP-based system and the data traditionally stored in product data models are also analyzed to facilitate the construction of the extended DCSP-based system.

Based on the extended DCSP-based product configuration and cost estimation system, an augmented backtracking algorithm is designed. In the algorithm, structural constraint handler and activation constraint handler are embedded into a classical Constraint Satisfaction Problem (CSP) solving procedure, both of which are able to trigger the extension of the extended DCSP-based system. Thus, the solving process of the extended DCSP-based system can be viewed as a sequence of stable phases, each of which is classical backtracking performed on a fixed classical CSP. To address issues of augmented backtracking (trashing and late conflict detection), a hybrid solving algorithm—forward checking and backjumping with fail-first heuristic (FC-BJ with FF)—is applied to the extended DCSP-based system to enhance solving efficiency. Furthermore, a hybrid heuristic (a combination of an amended constraint-most heuristic with fail-first heuristic) is integrated with FC-BJ to control backtracking occurrence in the implementation of the algorithm FC-BJ with FF on the extended DCSP-based system.

Finally, the extended DCSP-based configuration and cost estimation system is evaluated using two configuration tasks (desktop PC configuration and cable-operated elevator configuration). Three proposed algorithms are performed on the two DCSP-based configuration tasks respectively and good solving efficiency is achieved.

Acknowledgments

There are a lot of people I would like to thank for a variety of reasons. First of all, I would like to express my gratitude and sincere appreciation to my supervisor Associate Professor Ng Wee Keong, for giving me the opportunity to pursue Ph.D. at NTU, and for his unending patience, guidance, encouragement, and sharing of his knowledge. From him, I started to know the world of research. From him, I have learned how to tackle a problem systematically and deeply. Also from him, I have learned how to write a paper and present my paper properly. This thesis could not have been completed without his full support.

I also would like to thank my co-supervisor Dr. Song Bin from the Singapore Institute of Manufacturing Technology (SIMTech), for his help and patience in guiding me in the knowledge of cost estimation. Without his support, I would never have achieved the output in the thesis.

I would like to thank Dr. Sourav Bhowmick, Director of CAIS, for ensuring a conducive research environment, and the support staff (Lai Chee Keong, Chua Chiew Song, Ang Ah Giat and Loo Kian Hock) for their technical and administrative support.

I am grateful to all my friends in CAIS, CAIS-NIG² and CEMNET: Han Shuguo, Zhu Linhong, Pan Sijie, Huang Youliang, Liu Xin, Li Chenliang, Liu Yong, Wu Min, Sun Hongyang, Cao Yangjie, Lin Miao, Zhao Peilin, Zhuang Jinfeng, Chu Shumo, Zhang Jianling, Ma Kai, Lv Ying, Fang Hui, Ma Zongyang, Lu Feng, Guan Wei, Li Gang, Cheng Xiangang, Yang Shengbo, Li Yifan, Mao Yuanyuan, Fang Shou, Duan Qi, Lu Haifeng, Guo Tiantian, Wan Li, and Zhai Ke for their generous sharing of great ideas in data mining, bringing much happiness and maintaining a good working environment.

I am also grateful to friends in Singapore: Dr. Li Xiang, Dr. Qi Guojun, Dr. Yang Zhenlin, Ms. Li Hua, Dr. Sam Zhang Shanyong, Ms. Xie Xiaohua, Zhang Jianyi, Gu Yuan, Lu Jie, Hu Jinhua, Qi Xiaoying, Chen Xiuying, and Zhang Juan. Without their help and support, I would not have such a smooth study and happy life in Singapore.

Last but not least, my deepest gratitude is given to my parents, my sister, my boyfriend, and my best friend Li Ying for their love, support and encouragement. Without them, I have no way to be who I am today.

Contents

Abstract	i
Acknowledgment	ii
List of Figures	ix
List of Tables	xii
1 Introduction	1
1.1 Motivation	1
1.2 Research Objectives	7
1.3 Thesis Outline	10
2 Literature Review	12
2.1 Fundamental Issues on Product Family	13
2.1.1 Product Family and Platform	13
2.1.2 Product Family Modeling	14
2.1.3 Product Family Architecture	16
2.1.4 Modularity, Commonality and Variety	18
2.2 Platform-based Product Family Design	20
2.2.1 Scale-Based Product Family Design	21
2.2.2 Module-Based Product Family Design	22
2.2.3 Metrics on Cost for Product Family Design	23
2.3 Product Configuration	25
2.4 CSP-Based Configuration	29
2.4.1 CSP Definitions	30

2.4.2	CSP Solving Algorithms	31
2.4.3	CSP-Based Configuration	37
2.5	Discussion	43
2.6	Summary	44
3	DCSP-Based Product Configuration System with Cost Estimation	46
3.1	Overview	46
3.2	Configuration Domain in DCSP-Based Configuration System	49
3.3	Constraints in DCSP-Based Configuration System	51
3.3.1	Structural Constraints	51
3.3.2	Activation Constraints	54
3.3.3	Compatibility and Incompatibility Constraints	56
3.3.4	Equality and Inequality Constraints	57
3.3.5	Cost Constraints	58
3.3.6	Preference Constraints	65
3.4	DCSP-Based Configuration Problems	66
3.5	Mapping Relationship between PDMs and DCSP-Based Configuration System	68
3.5.1	PDM for Requirement	68
3.5.2	PDM for Product Configuration	69
3.5.3	PDM for Design Knowledge	72
3.5.4	Mapping Relationship	73
3.6	Constraint Acquisition	75
3.6.1	Association Rule Mining Approach	75
3.6.2	Constraint Acquisition	77
3.6.3	Experiment Result	80
3.7	Summary	82

4	Augmented Backtracking for DCSP-Based Product Configuration with Cost Estimation	83
4.1	Augmented Backtracking for DCSP-Based Configuration problem . . .	84
4.2	Implementation of Cost Constraints in the BT Solving Process	91
4.3	Experiments	93
4.3.1	Modeling DCSP-Based Desktop PC Configuration Problem from PDMs	94
4.3.2	Solving DCSP-Based PC Configuration by Augmented BT Algorithm	101
4.3.3	Experimental Results	106
4.4	Summary	110
5	Hybrid Algorithms for DCSP-Based Product Configuration with Cost Estimation	112
5.1	FF-BJ Algorithms for DCSP-Based Configuration	113
5.2	A Hybrid Heuristic for DCSP-Based Configuration	121
5.3	Experiments	127
5.3.1	DCSP-Based PC Configuration Using FC-BJ with FF	127
5.3.2	DCSP-Based PC Configuration Using Algorithm FC-BJ with Hybrid Heuristic	133
5.3.3	Experimental Results	138
5.4	Summary	143
6	Experiments on Elevator System Configuration	145
6.1	Modeling DCSP-Based Elevator Configuration System from PDMs . . .	146
6.2	Solving DCSP-Based Elevator Configuration by Augmented BT Algorithm	161
6.3	Solving DCSP-Based Elevator Configuration Using FC-BJ with FF . .	167
6.4	Solving DCSP-Based Elevator Configuration Using FC-BJ with the Hybrid Heuristic	170

6.5	Experimental Results	172
6.6	Summary	177
7	Conclusions	178
7.1	Summary	178
7.2	Future Work	183
7.3	List of Publications	185
	References	187

List of Figures

1.1	Leverage effect of product family architecture [39].	4
1.2	Example of a product family design.	5
1.3	Evaluation of product configuration [54].	7
2.1	Multiple views of a PFA [88].	17
2.2	Comparison of propagation techniques [13].	35
3.1	Constraint-based configuration system flowchart.	47
3.2	A product family architecture (PFA).	52
3.3	The structure of desktop personal computer.	53
3.4	Cost impact of the change of design parameters or cost drivers on PLCC.	62
3.5	A data model of the requirement.	69
3.6	Product family architecture of desktop computer.	70
3.7	Assembly view and cost view of product configuration.	71
3.8	Mapping relationships between PDM and DCSP.	73
3.9	Pruning association rules using the confidence measure.	79
3.10	Mining process for compatibility and incompatibility constraint.	80
4.1	Flowchart for BT algorithm on DCSP-based configuration.	85
4.2	Search trees of GT and BT for non-solution CSP problem.	90
4.3	System architecture of DCSP-based desktop PC configuration task.	93
4.4	Class diagram of DCSP-based desktop PC configuration task.	94
4.5	A example for requirement XML file.	95
4.6	PFA of Desktop PC configuration.	95

4.7	A backtracking process for the example PC configuration.	103
4.8	Experimental results for generating a feasible solution.	107
4.9	Experimental results for generating all configuration solutions ("DCSP+BT" vs. "CSP").	107
4.10	Experimental results for generating all configuration solutions ("DCSP+BT" vs. "Case")	109
4.11	Comparison of result optimization between "CSP+BB" and "DCSP+BT+Cost".	109
4.12	Experimental results of DCSP with BT for the first feasible result (with vs. without cost optimization).	110
5.1	Flowchart for FC-BJ with FF algorithm on DCSP-based configuration.	115
5.2	A FC-BJ with FF solving process for the example PC configuration. . .	130
5.3	Experimental results for generating a feasible solution.	138
5.4	Experimental results for generating all configuration solutions ("CSP" vs. "DCSP+FC-BJ-FF").	139
5.5	Experimental results for generating all configuration solutions ("DCSP+BT" vs. "DCSP+FC-BJ-FF").	140
5.6	Experimental results with cost optimization among "CSP" and "DCSP+BT" and "DCSP+FC-BJ-FF".	140
5.7	Experimental results of FC-BJ-FF on DCSP for the first feasible result (with vs. without cost optimization).	141
5.8	Experimental results for generating all configuration solutions ("DCSP+FC-BJ-FF" vs. "DCSP+FC-BJ-hybrid").	142
5.9	Experimental results for generating all configuration solutions ("DCSP+FC-BJ-FF" vs. "DCSP+FC-BJ-hybrid" vs. "Case")	143
6.1	Vertical section of a cable-operated elevator system [216].	146
6.2	PFA of an elevator configuration system.	147

6.3	Experimental results for generating a feasible configuration.	173
6.4	Experimental results for generating all configuration solutions ("DCSP+BT" vs. "CSP").	174
6.5	Experimental results for generating all configuration solutions ("DCSP+BT" vs. "DCSP+FC-BJ-FF" vs. "DCSP+FC-BJ-Hybrid"). .	175
6.6	Experimental results with cost optimization.	175
6.7	Experimental results of DCSP+BT for the first feasible solution (with vs. without cost optimization).	176

List of Tables

2.1	Product configuration approaches: advantages and limitations. [40, 80, 81]	29
3.1	Impact of part changes on PLCC (partial) [198]	63
3.2	Transactions (With Partial Items) for Association Rule Mining	78
3.3	Compatibility Constraints Generated from Experiment	81
3.4	Incompatibility Constraints Generated from Experiment	81
4.1	Variables and activation constraints in DCSP.	96
4.2	The activation of V^a and C^a in the backtracking searching process.	102
4.3	Backtracking searching process of the example case.	104
5.1	The activation of V^a and C^a in the FC-BJ+FF searching process.	127
5.2	FC-BJ with FF searching process for the example case.	129
5.3	FC-BJ with FF searching process for the example case with new requirement (partial).	131
5.4	The activation of V^a and C^a in the FC-BJ+hybrid searching process.	134
5.5	FC-BJ-FF searching process for the augmented example case (partial).	135
5.6	The searching process of FC-BJ-hybrid for the augmented example case (partial).	137
6.1	Variables and C_a in the extended DCSP-based elevator system (partial).	148
6.2	The activation of V^a and C^a in the extended DCSP-based elevator system using the augmented BT algorithm.	162

6.3	The activation of V^a and C^a in the extended DCSP-based elevator system using FC-BJ with FF.	168
6.4	The activation of V^a and C^a in the extended DCSP-based elevator system using FC-BJ with the hybrid heuristic.	171

Chapter 1

Introduction

1.1 Motivation

In the last two decades, many markets have exhibited increasing demand heterogeneity. Customers do not just want a product, they want a product that meets specific requirements. Thus, markets have fragmented into more and smaller market niches, and have also become much more dynamic. This leads to shorter order-to-market time in production and more uncertainty in market conditions. As a result, manufacturing companies face a dilemma of providing a wide variety of products for satisfying diverse customer needs with prices that can compete with mass produced products. They struggle with trade-off among market uncertainty, time-to-market, product variety and mass efficiency. Many companies have begun searching for ways to combine efficiency of mass production with variety of customer-oriented products; i.e., mass customization [39, 54, 214].

Mass customization aims at satisfying individual customer's requirements while staying near mass production efficiency. It increases product variety and reduces production cost at the same time [39, 86, 170]. As reducing production costs can be achieved by keeping production volumes at an economically high level, product family has been proved to be an effective method for mass customization—maintaining high production volumes while producing a variety of products [39, 54, 86, 170]. In a product family, a set of similar products share common functionality, parts or techniques, and also possess differentiation features for satisfying customers' specific re-

quirements [86, 88, 196]. This sharing principle results in the reduction of component types and amounts in inventory and manufacturing processes, and maintains relatively high production volume of each component. It contributes to benefits of product family design [39, 86, 127, 149, 159, 170, 196, 204] such as providing a variety of products for the market, achieving economy of scale, balancing the cost of offering variety by reusing modules and processes in manufacturing, reducing development time and product lead time, diminishing development risks, simplifying system complexity, improving the flexibility of producing processes, heightening the capability of product upgrade, and reducing testing and certification of complex products, such as aircraft [150, 170].

Product family design has been widely applied to a wide range of areas including *industrial products*, such as drills [135, 190], power supply [39, 88, 120], industrial mixers [66, 175], flow control valves [48], lighting control products [142]; *consumer products*, such as coffeemaker [4, 164], camera [196, 204], electronic products [158], computer [11, 115, 120], office furniture [41, 120], elevator [104, 212, 213], electric motor [119, 173]; and *complex products*, such as automobile and related products (e.g., engine, braking system, cooling circuit, underbody) [28, 127, 140, 151, 206], aircraft and spacecraft [1, 26, 141, 150].

In product family design, designers need to take into account a huge amount of information from various perspectives, such as product definition, customer requirements, resources, fabrication, assembly, manufacturing processes, inventory, and even supply chain. Now, manufacturers are also required to extend their operations beyond traditional manufacturing practice [214]. For example, more customers are concerned not only with product functionality and cost but also after-sale services. Customer concerns on environment protection and global and domestic environmental regulations and laws are also forcing manufacturers to consider environmental impacts of their processes and products. Recycling and disposal of products after use are gradually becoming manufacturers' responsibility. To respond to these new challenges,

an approach for manufacturing industry is to apply the concept of Product Lifecycle Management (PLM) to product development process. This uses information of the entire product life cycle (PLC) (design and development stage, production stage, use stage and retirement stage) to support production strategies, planning and management [214]. Therefore, larger amount and more heterogenous data are involved in product family design. How to efficiently and effectively represent all of these data is a challenge in product family design.

Many efforts have been expended in modeling data in product family design, such as generic bill-of-material based product family models [38, 39, 44, 91, 118, 208], product family classification tree [15], generic variety structure [91], and generic product structure [39]. Unified modeling language [50, 86], set theory [52], and graph grammars [40, 164] have also been applied in product family modeling. Product family architecture is the most popular product family modeling paradigm in which modularity, commonality and variety are three inherent characteristics [86, 88]. Based on the three characteristics, product family architecture is able to decompose product family structure into a set of module types. Based on the module types, a variety of products can be derived by instantiating a generic structure including all common module types and plugging-in differentiation module variants for delivering specific features to meet customers' needs [39]. The leverage among modularity, commonality and variety needs to be managed systematically [39], since high commonality may reduce manufacturing complexity and costs but make products more indistinguishable from one another. This makes products more desirable for the low market segment but less desirable for the high segment [86, 94], and may lead to low-end products to be over-designed and high-end products to be under-designed [34]. Figure 1.1 illustrates the procedure of product derivation in a product family based on modularity, commonality and variety. From the sales point of view, customers are concerned with functional features $\{f\}$ and corresponding values $\{f^*\}$. A family of products $\{P_1, P_2, \dots, P_n\}$ are designed to address the requirements of $\{Customer_1, \dots, Customer_m\}$ who share common requirement $\{f_0^*\}$ along with some similar and/or distinct requirements $\{f_1^*, f_2^*, \dots, f_m^*\}$.

Figure 1.1: Leverage effect of product family architecture [39].

From the engineering point of view, products $\{P_i\}$ are generated by combining predefined common elements $\{C_j\}$ and differentiation elements $\{D_k\}$. Configuration rules determine the generation of products so that only technology-feasible and customer-wanted products can be derived [10]. Although diverse product family models have been developed, most of them are not computation-efficient.

After modeling a product family design task, how to efficiently and effectively achieve product configuration is another issue in product family design. The core of product configuration within a product family is selecting and arranging combinations of components based on pre-defined configuration rules for satisfying given specifications [81, 155]. Many AI techniques have been applied in product family design and have shown great promise in automatic product configuration. Examples include case based reasoning configuration [30, 77, 103, 145, 155, 186], grammar based configuration [3, 4, 40, 41, 166], rule based configuration [12, 155, 203, 201], template based configuration [29, 112, 167], structure based configuration [70, 146], and resource based configuration [75, 178, 155]. As a promising approach for modeling and solving product family design tasks, Constraint Satisfaction Problem (CSP) has attracted much attention due to its highly declarative feature, generality and flexibility. A CSP can be simply described as a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restrict combinations of values that the variables can simultaneously take. The task of a CSP is to find solution(s) that assigns a value to each variable satisfying

Figure 1.2: Example of a product family design.

all the constraints. The concept in product family design tasks can be easily and straightforwardly modeled by the formalisms of CSP (i.e., variables, domains and constraints), and results can be easily understood. For example, Figure 1.2 illustrates a simple CSP-based computer family design task. In this case, constitutes of a computer and their properties are modeled as module variables and property variables in CSP respectively (e.g., module variable *monitor*, *mouse*, and property variable *color* of *mouse*). Models of each constitute are involved in the domain of corresponding variable (e.g., $\text{domain}(\text{mouse}) = \{\text{Logitech G9X}, \text{Microsoft ArcMouse}, \text{Dell MOC5UO}\}$). And restrictions among these constitutes and properties are represented as constraints in CSP (e.g., restriction “the color of keyboard and mouse are the same” is represented as constraint $\text{keyboard.color} = \text{mouse.color}$). Based on the product family model, various products can be generated by instantiating all common module variables and some differentiation module variables; i.e., selecting a model of each module, such that a value is assigned to each variable in the CSP without conflicting with any constraint. For instance, in Figure 1.2, products “PC Model A” and “PC Model B” are derived from product family “Desktop PC” by selecting different models of a module and selecting, or not, a differentiation module “Floppy Disk”. The product generation process within a product family can be performed by vari-

ous CSP solving algorithms (e.g., systematic search, repair strategy and consistency techniques) without considering the design background; that is, domain independent. Solutions to the CSP are just product configuration results required. Product configuration optimization problem can also be solved by CSP. A number of CSP-based configuration systems [56, 66, 98, 123, 124, 151, 155, 177, 212] have been proposed and widely applied. However, some inherent natures of product family design, such as dynamic inclusion of components and the hierarchy of components, are not efficiently represented and utilized to facilitate CSP solving.

Cost is another significant issue in product family design, reasons of which can be analyzed from several perspectives. First, in general cases, based on a product family architecture, several candidate products that match customers' requirements can be configured and are evaluated for selecting an optimal one, as illustrated in Figure 1.3. Cost is one of the critical criteria for product configuration evaluation. Second, in product family design, companies need to balance all benefits achieved by increasing commonality across products and all the costs incurred [54]. For example, sharing components in a product family may result in low-end products to be over-designed [34], which causes additional production costs [54]. Such trade-off can be measured with the help of costs. Third, due to more dynamic market conditions and shorter time-to-market, manufacturers face more uncertainty in production orders that may lead to shorter product and component life cycle and increase operational costs in product development and manufacturing. All of these result in further pressure on cost reduction in product family design. Finally, for customers, product cost is as important as functionality and quality for a competitive product [102]. Therefore, proper cost estimation before final configuration is critical for production management. As product design influences about 70%-85% of the total product cost [134], separating cost estimation from product configuration reduces the effectiveness of both tasks. Furthermore, in PLM, Product Lifecycle Cost (PLCC), which includes all the costs incurred throughout the entire PLC, is an important criterion. Hence, by

Figure 1.3: Evaluation of product configuration [54].

integrating proper cost estimation into product configuration, a product is designed with optimal combination of functionality, quality, and cost.

Many efforts have been made on relationships between costs and modularity, commonality and variety in product family design [35, 55, 61, 94, 141, 163, 170, 220], and various cost estimation methodologies have been developed (e.g., case based reasoning [27, 51], rule-based [148, 161], neural network [21, 27, 106, 219], activity-based costing models [7, 24, 83, 141, 138]). However, the integration of product family design and cost estimation has not been satisfactorily addressed, and thus, cost is not able to provide concurrent and effective support for product configuration.

1.2 Research Objectives

The goal of this thesis is to develop a Dynamic Constraint Satisfaction Problem (DCSP) based product configuration system to address the following issues:

- In mass customization, a large amount of heterogeneous data need to be modeled in product family design tasks. The information included in product family design has extended from a single product to a family of products, and from traditional production data to the entire life cycle data. Therefore proper and effective data representation is critical in product family design.
- Although a large amount and variety of data are involved in a product family design task, generally, only a subset of the entire data set are involved in its final

solutions. In this case, if the entire data set is involved in a product configuration process, the searching process is time-consuming.

- In product family design, the representation of data has to express not only the semantics of data, but also, if not more important, the structure and topology of data that is core to product family design.
- Cost is a significant composition of and a critical criterion for product family design. However, the integration of product family design and cost estimation has not been satisfactorily solved.
- The efficiency of searching algorithms of CSP-based product configuration system determines whether it is applicable to many time-sensitive real-life applications. The efficiency issue must be addressed without sacrificing the accuracy and quality of configuration results.

Specifically, main contributions of this thesis are summarized as follows:

- (i) An extended DCSP-based product configuration system is proposed. By concepts of DCSP (i.e., variables, domains and constraints), the extended DCSP-based system is able to straightforwardly and effectively represent a large amount and variety of data in product family design tasks. These data include *production data* representing domain objects in a product family design task, and engineering and cost relationships among these objects; *description of the domain knowledge* consisting of design background knowledge, such as configuration rules, function decomposition; and *specifications of desired products* describing customer requirements and, possibly, optimization criteria that are used to guide the search for a good solution.
- (ii) In the extended DCSP-based product configuration system, structural constraints are proposed to straightforwardly capture the inherent hierarchy among modules and components in product family design tasks. Activation constraints

are also proposed to represent derivation relationships among variables in the extended DCSP-based system. Both structural constraints and activation constraints can be utilized to dynamically extend the search space of the extended DCSP-based system by activating variables into the DCSP according to certain triggering conditions (i.e., hierarchical and derivation relationships among variables) such that only a subset of the original variables are involved in the final solution to the DCSP. Thus, the searching space can be controlled to minimum.

- (iii) To integrate cost estimation tasks with product family design tasks tightly, cost calculation constraints are proposed into the extended DCSP-based product configuration system, by which results of product configuration and cost estimation can be obtained simultaneously, and cost can be utilized as a coherent criterion as functionality and performance. To study the impact of the change of cost drivers and design parameters on the cost of a product, or the entire product family, or even the whole product lifecycle, cost variation constraint are also proposed into the extended DCSP-based product configuration system.
- (iv) Mapping relationships between concepts of the extended DCSP-based product configuration system and the data traditionally stored in product data models that are used by manufacturers are also discussed to facilitate the construction of the extended DCSP-based system. We also implement an automated constraint generation method using association rule mining approach to generate compatibility and incompatibility constraints that represent co-existence relationship among components from historical configuration data.
- (v) Based on the extended DCSP-based product configuration system, an augmented backtracking algorithm is designed, where structural constraint handler and activation constraint handler are embedded into a classical CSP solving procedure, both of which are able to trigger the extension of the extended DCSP-based system. Then the solving process of the extended DCSP-based system can

be viewed as a sequence of stable phases where classical backtracking algorithm is performed on a fixed classical CSP.

- (vi) Based on the augmented framework of the augmented backtracking algorithm above, a hybrid solving algorithm (i.e., a combination of algorithm forward checking and backjumping with fail-first heuristic) is proposed, which is applied on the extended DCSP-based system to address issues of the augmented backtracking algorithm (i.e., trashing and late conflict detection), and to enhance the solving efficiency.
- (vii) Furthermore, to address issues of forward checking and backjumping with fail-first heuristic above, a hybrid heuristic (i.e., a combination of an amended most-constrained heuristic with fail-first heuristic), together with forward checking and backjumping algorithms, is proposed to control the backtracking occurrence in the implementation of forward checking and backjumping on the extended DCSP-based system.

1.3 Thesis Outline

This thesis is organized as follows. In Chapter 2, preliminary knowledges of product family design and constraint satisfaction problem are provided, and the state-of-the-art on product family design and product configuration are reviewed. In Chapter 3, an extended DCSP-based product configuration and cost estimation system is proposed, which extends classical CSP by dynamically creating new active variables, captures the inherent hierarchy of modules in product configuration, and integrates cost estimation into product configuration tightly. Based on the extended DCSP-based configuration and cost estimation model, an augmented backtracking algorithm is proposed in Chapter 4 where a structural constraint handler and an activation constraint handler are embedded into a classical CPS solving procedure, both of which are able to trigger the extension of the DCSP-based system. Then in the solving process, the

extended DCSP-based system can be treated as a sequence of stable phases, in which a general backtracking algorithm is performed on a fixed classical CSP. The implementation of two categories of cost constraints are also discussed in this chapter. To address issues of the augmented backtracking algorithm (i.e., trashing and late detection of conflicts), a hybrid solving algorithm for the DCSP-based configuration and cost estimation system; i.e., a combination of algorithm forward checking and backjumping with fail-first heuristic, is proposed in Chapter 5. A hybrid heuristic (that is, a combination of an amended most-constrained heuristic with fail-first heuristic) is also advocated in this chapter to help control backtracking occurrences in the hybrid solving algorithm, and improve its solving efficiency. In addition, in Chapter 4 and Chapter 5, a desktop personal computer configuration task is modeled as an extended DCSP-based configuration and cost estimation system, and solved using the three proposed algorithms respectively. Chapter 6 elaborates the modeling and solving process of the extended DCSP-based configuration and cost estimation system on a more complicated example—a cable-operated elevator configuration task. Finally, Chapter 7 concludes the thesis. Future extensions to this study are also identified in this chapter.

Chapter 2

Literature Review

Product family has been recognized as an effective approach to achieve the target of a mass customization; i.e., increasing product variety to satisfy individual customer's requirements across various market niches, while maintaining near mass production efficiency [39, 86, 143, 170, 183, 188, 192, 204]. It provides high product differentiation while taking advantage of high production volumes within the manufacturing capabilities of companies [86]. The rationale of developing product families has been well recognized in both industry and academia [86, 170], and the benefits of product family design have been observed in a multitude of perspectives, such as providing a variety of products, achieving the economy of scale, balancing the cost of offering variety by reusing modules and processes in manufacturing, reducing development time and product lead time, diminishing development risks, simplifying system complexity, improving the flexibility of production process, heightening the capability of product upgrade [39, 86, 127, 149, 159, 170, 196, 204], and reducing testing and certification of complex products [150, 170]. In a sizeable literature, product family design has been tackled from diverse viewpoints, such as marketing/sales, functional requirements, engineering, manufacturing and assembly, process planning, logistics, information technology, business strategy, and general management [39, 86, 88, 170]. Correspondingly, the entire product family design process can be segmented into product definition, product design, process design, and supply chain design [86, 182]. In this work, the focus is the domain of product design, where functional requirements

derived from customers' needs are mapped into design parameters, and product family design results are generated [86]. In this domain, modeling and solving product family design problems are the two main tasks. Powerful knowledge-representation models are necessary to capture heterogeneous data with great variety and complexity in product family design, and efficient reasoning methods are required for the generation and optimization of product family design solutions.

This chapter reviews the state-of-the-arts on product family design. Section 2.1 provides some fundamental definitions and issues on product family. Section 2.2 reviews platform-based product family design. Section 2.3 introduces various product configuration techniques. As a promising configuration approach, several CSP-based product configuration systems, together with their respective advantages and limitations, are reviewed in detail in Section 2.4. A Discussion on existing CSP-based configuration systems is provided in Section 2.5. Finally, summary is presented in Section 2.6.

2.1 Fundamental Issues on Product Family

2.1.1 Product Family and Platform

A **product family (PF)** refers to a set of similar products that possess some common features/functionality, share common structures or production technologies, and hold specific features/functionality to meet particular customer needs [39, 86, 88, 120, 137, 196]. Each individual product within a PF is called a **product variant** [39, 86]. While a PF aims at a specific market segment, each product variant in the PF satisfies a particular set of customer requirements within the market segment [39, 86, 88, 170].

The common structure and/or common product technology shared by all product variants form a **platform** of PF, which refers to the technological foundation of product families, and acts as the basic architecture for all product variants in a PF [39, 41, 42, 44, 86, 88]. Implementing platform can improve the economy of scale and scope [143, 170], leverage engineering across product variants in a PF [88, 157],

and facilitate customization by efficiently deriving a variety of products from a platform. There are two prevailing streams in the study of platform. The first handles platform as a physical one. That is, “a set of common components, modules, or parts from which a stream of derivative products can be efficiently developed and launched” [120, 170]. Thus its main concern is how to identify the common components, features, and subsystems that are stable and shared across product variants within a PF [39, 45, 86, 126, 149, 206].

The second stream refers a platform as “a set of technologies, components, or functions, and their arrangements” [32, 196]. The main issue in this view is the shared logic and underlying structure of a platform. Meyer and Lehnerd [86, 120] defined a platform as a common structure involving subsystems and interfaces from which product variants are derived. McGrant [86, 117, 170] represented a platform as a collection of core technologies. Robertson and Ulrich [86, 149, 170] stated that a platform is a collection of shared assets including components, processes, knowledge, people and relationships. Three aspects of platform are defined: modular architecture, interfaces, and rules to which modules must conform [11, 86]. Three types of platforms are discerned, namely modular, scalable, and generational platforms [86, 170, 218], which generate product variants within a PF through configuring existing modules [120], extending the same function with varying capacities [173], and leveraging product life cycles for rapid next generation development [86, 111], respectively.

2.1.2 Product Family Modeling

In product family design, product family modeling is one of the main tasks. One of the challenges in PF modeling is how to represent families of products rather than single products [39, 59]. Bill-of-Materials (BOM) defines “items or raw materials that go into the product” [62]. While BOM links parts to variants in a product, each of which is specified by a unique BOM, Modular BOM (MBOM) [74], an extension of BOM, links parts to planning modules that refer to a set of components satisfying

certain options of a product. Another extension of BOM, Generic BOM (GBOM) [39, 74] has been introduced for PF modeling, which is based on the assembly structure of a PF, and represents multiple products in a single structure, where variants are specified only once [170]. Within a GBOM, Primary Generic Product and Generic Subassembly Products represent the set of variants of a primary product and the set of subassemblies respectively [170]. There are many GBOM based product family representations. van Veen [194] modeled PF based on GBOM from the production view. Generic Product Modeling (GPM) is investigated from both commercial and assembly perspectives for PF representation [208]. Erens *et al.* [43, 44] proposed a GBOM-based product modeling language for product variety. Based on GBOM, McKay *et al.* [86, 118, 170] integrated product modeling data with product variety from multiple views, such as manufacturing, assembly, and sales. Jiao *et al.* [91] proposed an extension of GBOM to include operations. De Lit *et al.* [38] provided a GBOM-based assembly planning model for PF.

Baldwin and Chung [10] proposed a discipline-independent product data model with optional contents. Unified Modeling Language has been applied to represent configuration knowledge bases for mass customizable products [50, 86]. Bei and MacCallum [15] advocated Product Family Classification Tree for the classification of modules and end-products from a functional viewpoint. Jiao, Tseng and their team [39, 90, 170] investigated diverse data relationships within a PF; i.e., product-to-product, product-to-family and family-to-family relationships. They also developed a Generic Variety Structure to embody variety and its derivation by common product structures, variety parameters, and configuration constraints [91]. Du *et al.* [39] proposed Generic Product Structure to provide a generic data structure of PF as a platform for tailoring product variants to individual customer's needs. Finch [52] developed a set-based model for product platform design and manufacturing processes. Graph grammar has been applied for platform design [164], and PF modeling [40]. The conceptual modeling of PF was investigated by Mänistö [125], which focuses on the evolution of PF

descriptions and products generated according to them. Product family architecture (PFA) has been widely applied on the representation of PF, which will be reviewed in detail in the Section 2.1.3.

2.1.3 Product Family Architecture

As a simple but efficient PF modeling paradigm, **product family architecture (PFA)** refers to an architecture representing the conceptual structure and overall logical organization of a PF. It includes “the arrangement of functional elements; the mapping from functional elements to physical components; and the specification of the interfaces among interacting physical components.” [191, 192]. Fujita *et al.* [59, 61, 88] pointed out that one important characteristic to discern PFA from the architecture of a single product is the simultaneous handling of multiple products. From both functional and physical perspectives, Erens and Verhulst [44] described PFA as a package of single product models, instead of a unified PF model. A PFA involves the relationships of part-to-part, part-to-product, product-to-product, and product-to-family, but excludes family-to-family relationship involved in the architecture of product families [39, 59, 188].

A PFA can be investigated from three spectrums: functional, behavioral, and structural [88, 218], which are respectively characterized by functional features (FFs) depicting customer requirements [139, 191, 218], technical parameters (TPs) representing modules and modular structure [88, 97, 139, 218], and component/assemblies (CAs) strongly related to the physical construction of a PF [88, 193, 218]. As illustrated in Figure 2.1, each view captures a specific aspect of the product information; i.e., functionality, technological feasibility, manufacturability and economic evaluation respectively [88]. The mapping relationships between each views are realized by design activities, manufacturing and logistics activities, and sales and marketing activities [88].

A PFA can be modular if the mapping from functional elements to physical components is one to one or many to one, or integral if there is a complex or coupled

Figure 2.1: Multiple views of a PFA [88].

mapping from functional elements to physical components and interfaces between them [86, 128, 170, 192]. For example, computers are predominantly modular [11, 170], and automotive and aircraft are relatively more integral [1, 127, 170, 165, 196]. Cutherrell [33, 86] pointed out that an integral PFA is driven by product performance or cost, while a modular PFA is driven by variety, product change, engineering standards, and service requirements. Sosa *et al.* [86, 176] emphasized on the importance of integrality and modularity in design and used a design structure matrix to identify whether a system is modular or integral. Robertson and Ulrich [149] stated that increasing modularity with proper integrality can balance tradeoffs between distinctiveness and commonality in a PFA. From design economy perspective, Whitney [205] focused on modularity and interfaces in a PFA.

PFA embodies three inherent characteristics of PF design: *modularity* of product structure, *commonality* and *variety* among product variants [39, 86, 88, 170, 196]. Modularity offers the flexibility for a base product to be extended so as to meet diverse customer requirements. Commonality embodies the difference between PFA and the architecture of single products. Based on modularity, variety delivers specific features for product variants by plugging-in modules [39]. A PFA provides a generic structure to capture and utilize commonality, based on which new product variants are generated with certain variety in response to individual customer's requirements [39]. These concerns will be elaborated in Section 2.1.4.

2.1.4 Modularity, Commonality and Variety

A PFA consists of a set of **modules** that refer to physical or conceptual groupings of components that share some features [88, 132]. It can be categorized as process, function, organ, and component modules [82, 88], or primitive and compound modules [39, 81], or common and differentiation modules [39, 81], or standard and parametric modules [81]. **Modularity** refers to “an attribute of a system describing the degree of overall coupling between function units ... [where] ... this coupling may occur at multiple length-scales and dimensions” [196, 197]. Therefore, modularity concerns on decomposition of product structures, and describes module types. It separates a product into independent modules [88, 132], which are identified in such a way that between-module interactions are minimal while within-module interactions are high [86, 88, 191, 192]. Corresponding to the multiple views of PFA, there are three types of modularity, namely functional, technical, and physical modularity [86, 88], Ulrich & Tung, and Pine [143, 193] defined six categories of modularity: component swapping, component sharing, fabricate-to-fit, bus, sectional modularity, and mix modularity. The most of the latter extensions are built upon these modularity types [39, 86, 101].

Commonality reveals the difference between PFA and the architecture of a single product. The latter is concerned with modularity of product structure, and the former also focuses on the commonality among product variants [86, 88]. Commonality characterizes the grouping of similar module variants under specific module types defined by modularity, and thus clustering is its major concern [86, 88]. Commonality in PF design leads to a maximum repetition of modules to meet customer needs, and thus, results in effective and efficient design, manufacturing/assembly, distribution, and services [88]. Corresponding to three types of modularity, there are three types of commonality; that is, functional, technical, and physical commonality [86].

The relationship between modularity and commonality can be explained by class-member relationships [86, 88]. While a PFA and corresponding modular types (i.e.,

class) are specified in terms of modularity, product variants within the PFA deploy different instances (i.e., member) of the module types. In other words, a product family is represented by modularity and product variants are diversified based on the commonality between module instances [86, 88].

Product variety refers to the diversity of product variants that a PF offers to customers [86], and can be classified into two types: functional variety and technical variety [86, 88]. While functional variety is mostly related to the differentiation in the attributes and customer satisfactions, technical variety is closely associated with technologies, components and assemblies, manufacturability, and costs [86]. PF design tries to maintain or increase functional variety in the functional view, and reduce technical variety in the behavioral and structural views by systematic planning of modularity to facilitate plug-in modules [86]. Correspondingly, there are two design strategies: design for functional variety that emphasizes on increasing functional varieties (e.g., product line structuring [39, 140] and product positioning [78, 88, 86]), and design for technical variety that tries to reduce technical varieties for easy-manufacturability and cost advantages (e.g., design for variety [39, 59, 110], variety reduction program [184], design for modularity [45, 46], function sharing [192], reconfiguration for variety [25]).

The relationship in the generation of variety within a PF can also be interpreted by the class-member relationship. Based on the modules defined in a PF, product variants are derived from two ways [39, 88]: product variants differentiate in terms of their attribute variables (such as FFs and TPs); or product variants variegate by diverse instances of specific attribute variables. That is, for each type of product variants (class) specified by a set of attribute variables (class attributes), product variants of this type can take different values (members) of certain attribute variables [88].

Some methods have focused on the fulfillment of variety. Du *et al.* [39] discerned two sources of variety; i.e., different variants of modules, and possible existence of certain structural relationships. Jiao *et al.* [91] suggested that each child node inherits parameters from its parent according to a direct correspondence. Malmqvist [109]

studied the situation that parameters of a child are functions of the same parameters of its parent. Fujita and Ishii [59] applied component selection and attached features to achieve product variety. Simpson *et al.* [172] gained variety by scaling from a common product platform. Component combination based on modular PF design was proved to be practical in variety generation [191, 193]. Du *et al.* [39] proposed four elements for variant derivation (i.e., selection constraints, parameter propagation, include conditions, and variety generation), and distinguished three basic variety generation and nesting mechanisms (namely attaching/removing, swapping, and scaling). Simpson [170] suggested that product variety can be implemented by adding, substituting, and/or removing one or more functional modules from the platform, or stretch or shrink the platform in one or more dimensions.

The leveraging of modularity and commonality in PFA development is observed by Siddique *et al.* [164, 165]. Du *et al.* [39] studied the balance between commonality and distinctiveness within a PFA. Jiao, Tseng and their team have extensively studied the relationship between PFA and mass customization [39, 188, 87, 88]. They stated a PFA embodies variety through two layers (systematic planning of modularity and commonality), across three views (functional, behavioral, and structural views), and involves four main components (product line taxonomy, building blocks, configuration rules, and economic evaluation) [88].

2.2 Platform-based Product Family Design

PF design can be described as an product variant derivation process based on a platform [39]. There are two basic approaches [170, 173]. One is a *top-down* approach, by which a family of products are generated based on a product platform and its derivatives. For example, platform-based Kodak and Fuji camera family design facilitates them to develop product variants faster and more cheaply [196]. The other is a *bottom-up* approach, wherein distinct products are redesigned or consolidated for component standardization. For instance, after designing more than 100 lighting control products for individual customer's needs, Lutron Electronics company redesigns

its platform including 15-20 standard components that can be configured into these more than 100 models [142]. The key in both approaches is the platform from which product variants are derived.

Corresponding to scalable and modular platforms introduced in Section 2.1.1, platform-based PF design approaches are categorized into two types [86, 170, 173, 196]. one type is the scale-based PF design, wherein some scaling variables “stretch” or “shrink” a PF in one or more dimensions to satisfy a variety of customer needs [131, 173, 196]. The other type is the module-based PF design, wherein based on a modular platform, product variants are generated by adding, substituting, and/or removing one or more functional modules [11, 39, 158, 170, 191, 218].

2.2.1 Scale-Based Product Family Design

The scale-based approach is commonly used in the aircraft [1, 150, 207], automotive [130, 165], and motor industry [119, 173]. It consists of two basic tasks [86, 170]: The first is platform selection; i.e., determining which design parameters take common values, which can be pre-determined [60] or determined during optimization [5, 34]. The second task is to determine the optimal values of common and distinctive variables to meet functional and economic requirements. While most methods consider only a single platform shared across the entire PF [34, 86], the others consider multiple platforms such that design variables can be shared by any subset of product variants within a PF. These approaches enhance exploration of the solution space, but sacrifice the computational efficiency [36].

Simpson, Maier & Mistree [86, 173] provided a product platform concept exploration method to minimize the sensitivity of performance variations in scaling factors. Nayak *et al.* [131] combined the platform selection problem into the commonality and performance trade-off procedure. Messac *et al.* [119] proposed a PF penalty function for selecting or combining common and scaling variables. Using multidisciplinary optimization, Willcox and Wakayama [207] investigated the effect of design variable scaling

across multiple aircrafts sharing common components. Many optimization algorithms have been used on scale-based PF design [170], such as 0-1 integer programming [60], the compromise decision support problem [173], physical programming [119], genetic algorithms [170], and exhaustive search techniques [76] for small design space.

2.2.2 Module-Based Product Family Design

Module-based PF design is based on modular PFA wherein a functional module can be changed independently by only changing the corresponding part [86, 191]. It allows standardization that is essential for the economy of scale, thus, customer-built products can be derived from standard models, and variety can be created by configuring existing modules/components [86]. It has been widely applied. For example, Sony built its module based Walkmans family [158]; Volkswagen constructed a module-based platform for all its brands (e.g., Volkswagen, Audi, Skoda) [206]; Feitzinger and Lee [49] developed Hewlett Packard ink jets and laser jet printers based on modular components; Baldwin and Clark [11] generated the first modular computer family for IBMs System/360.

As introduced in Section 2.1.4, modularity is a key concern in PF design. Related methods abound in literature [11, 86, 170]. Quality Function Deployment (QFD) has been widely applied in module identification within a PF [45, 111]. Its extension Modular Function Deployment (MFD), using module identification matrix with the assessment of module integration, was proposed by Erixon and Ostgren [46]. Mattson and Magleby [113] presented concept selection techniques in the early stages of modular PF design. McAdams and Wood [114] built an approach for functional module representation in a quantitative manner to facilitate the identification of core modules for a PF. A set of heuristics have been presented for module identification [179, 218]. Using Design Structure Matrix (DSM), Yu *et al.* [217] clustered highly interactive groups of product elements into modules. Similar methods can be found in [20, 100]. Hölttä and Salonen [79] compared 3 modularization methods; i.e., MFD, the computerized DSM, and the heuristic approach. Gershenson *et al.* [67] compared several

DSM-based methods for identifying modular architectures. Otto *et al.* [136] proposed a modularity matrix for a PF allowing commonalties to be easily identified. Modularity also plays an important role in product evolution, upgrade, and retirement [85]. Newcomb *et al.* [133] proposed an algorithm for module decomposition based on different life cycle viewpoints. Kimura *et al.* [95] discussed the impact of modularity on component reuse, whereas Zhang *et al.* [220] investigated the impact on product retirement costs. Martin and Ishii [111] developed a method for designing modular platform according to multiple generations of products.

The importance of interface standardization in PFA was studied by Meyer and Lehnerd [120]. A method for managing module interfaces within a PF is presented by Sundgren [183]. Blackenfelt [20] introduced robust interface design to increase interface commonality within a PF. van Wie and his coauthors [195] focused on the embodiment issues of module interfaces in terms of the connections between modules. Sosa *et al.* [176] clarified the difference among interfaces from multiple perspectives; i.e., energy, material, spatial and informational. A function for assessing modularity degree by concerning the number of components, interfaces, and substitutability factor of a PFA was proposed by Mikkola and Gassmann [122].

2.2.3 Metrics on Cost for Product Family Design

Product family design essentially can be viewed as multi-objective optimization problems [174]. That is, a PF design requires the leverage among cost, revenue and performance [54]. As a critical criterion in PF design, cost has attracted numerous efforts as reviewed next.

One measure of a product platform's success is how quickly and cheaply new products can be derived [170]. Meyer *et al.* [120, 121] proposed a platform efficiency metric measuring the ratio of the cost for developing derivative products to that for developing a platform in the PF, and a platform effectiveness metric measuring the ratio of the revenue created by a platform and its derivatives to the cost required to develop

them. Simpson [170] assumed that PF design maximizes commonality among products while minimizes production costs, and pointed out without explicitly modeling demands and associated manufacturing costs, the assumption may lead to suboptimal PF designs. Ulrich [191] studied life cycle cost issues related to module-based PFA. Similar job was done by Dahmus and Otto [35]. Martin and Ishii [110] developed a quantitative indices for measuring setup costs and product differentiation that correlates with many indirect costs of providing variety. A design to cost model [47] integrates cost modeling into QFD for making trade-off decisions between quality and cost at the early stages of product design. Design for Manufacturing and Design for Assembly proposed by Boothroyd and his coauthors [22, 37] entail links between design decisions and costs. Jiao and Tseng [87] proposed time-estimating relationships (TERs) to establish the relationship between cost-related design features and standard production time, and estimated costs subject to TERs. Siddique [163] stated that the saving from design reuse affects both development cost and time. Based an economic model, Kim and Chhajed [94] concluded that high commonality decreases production costs but makes products more indistinguishable. Zhang *et al.* [220] investigated the impact of modularity on product retirement costs. Fujita [58, 141] proposed a cost model for product variety, including the cost of design, development, facility, material, and processing. He assumed fixed costs are proportional to module's attributes, and variable costs can be saved from similar and same design instances. Fujita and Yoshida [61] defined a monotonic cost model to assess benefits of commonality. Fixson [53, 55] developed a multi-dimensional PFA description method to link modularity and cost, and examined the cost of PFA from a product life cycle perspective. Based on activity based costing approach, Park and Simpson [141] investigated the effects of commonality on individual costs. Jiao and Tseng [89] separately proposed indices of design and process customizability for evaluating the cost effectiveness for customization. H'midaa *et al.* [84] introduced cost entity to establish a tight link between technical and economic variables.

Various cost estimation techniques have also been developed for accurate cost estimate to support PF design optimization, such as case based [27, 51, 106, 148], rule based [148, 161], fuzzy logic [134, 161], neural network [21, 27, 106, 219], parametric [19, 27], operation-based [92, 134], feature-based [63, 92, 219], and activity-based costing (ABC) models [7, 24, 83, 141, 138]. More information can be found in [134].

Although a large amount and range of research on the impact and estimation of cost in PF design have been made, the integration of cost estimation with the product design process has not been satisfactorily solved. Most cost estimation techniques are implemented only after a product model is built, thus are not able to offer efficient and concurrent support for PF design optimization. Less than half of the optimization approaches [170] integrate manufacturing costs directly within PF design information formulation. Most of the approaches that include costs in their formulation use cost as the single optimization objective, rather than multi-objective [170].

2.3 Product Configuration

Besides modeling PF design tasks, how to solve the models efficiently and effectively is another issue in PF design. The core of product configuration within a PF is selecting and arranging combinations of components satisfying given specifications [81, 155]. It entails two key features: A product variant involves instances of a fixed set of well-defined component types (i.e., modules of PF), and components interact with each other in predefined ways (i.e., interact according to configuration rules) [155]. Many AI techniques have been applied in PF design, and have shown great promise in automatic product configuration.

Gonzalez-Zugasti *et al.* [69] solved a platform-based PF design problem as a general constraint-based optimization problem. Yigit *et al.* [215] generated modular products as a generalized subset selection problem in terms of a balance of the quality loss due to modularization and the cost of reconfiguration. Kamrani and Gonzalez [93] applied a genetic algorithm based method for design for modularity. Agard and Kusiak [2]

utilized data mining techniques in PF design that consider the mapping relationship between functional requirements and technical structure. Siddique *et al.* [112, 167] proposed a template based approach for generating CAD models automatically, and extended it to include parametric design, mating relationships, and modularity. Chandrasekaran *et al.* [29] proposed a template approach for platform-based product design based on historical product design knowledge and the conversion of energy through a product. Saab automobile deployed function-means-trees and a chromosome model to represent a parameterized set of design solutions [31, 170]. Agent-based techniques for module based PF design has been explored [105] due to its advantages for concurrent product design and configuration [162]. In structure-based configuration, ontologies are generated for representing the knowledge of concept, procedure, and task specification in configuration domain [70], and some structure oriented configuration systems have been developed; e.g., Drive Solution Designer (DSD) [146], KONWERK [71], and EngCon [146].

Case-based reasoning has been widely applied in product configuration [30, 77, 103, 145, 155, 186]. Its major steps involve representing customer needs for a new configuration; retrieving similar cases from historical designs; adapting the retrieved cases to satisfy the new case, and resulting in a solution; validating the solution, and revision may be needed if the solution proves incorrect; and storing the newly adapted cases [30, 77].

Grammar-based approach has also been employed for automatic product configuration within PF. As the first successful application, Agarwal and Cagan [3, 4] developed shape grammars for product variety, using coffeemaker PF as illustration. Siddique and his coauthors [164] proposed a graph grammar based approach for product platform design, where graphs represent elements and structures in a platform, and grammars characterize relationships between platform and product variants. The same coffeemaker example was employed. Later they utilized graph grammars in a web-based PF reasoning system [166]. Du, Jiao and Tseng [40] proposed Programmed

Attribute Graph Grammar for PF design by rewriting the starting graph using four module operations based on graph grammars (i.e., attaching, removing, swapping and scaling), and these operations are performed according to a control diagram that captures the complex relationships and configuration constraints between modules. Later they extended this approach, and proposed a graph rewriting system to transform product family graphs into product variant graphs [41].

Rule-based configuration methods, also known as expert systems [155], use production rules with the form of “*if condition then consequence*” to represent both domain knowledge and control strategy. For instance, R1\XCON system using production-rule language OPS5 [12, 115], RIME providing structuring concepts for rule based programming [9], JESS [203], and DROOL [81, 201]. In rule-based systems, product information is represented in a specific rule language with predefined syntax. Requirement specifications act as the input and are checked in the conditional part of rules. If the condition of a rule is satisfied, then the system executes its consequence. A final configuration is generated when all rules are checked. As rule-based systems don’t represent domain knowledge and control strategy separately, their knowledge maintenance becomes very difficult.

To address the issue of rule-based systems, the model-based configuration methodology has been popularly used, due to its better separation between domain knowledge and how the knowledge is used, enhanced robustness, compositionality and reusability [155].

Logic-based approaches are a type of module-based configuration systems. They are based on description logic (DL) [116], a formalism for representing and reasoning with knowledge [155], and provide a framework to capture semantics and simple logic operations during product generation. There are three fundamental elements of DLs [116]: *individuals* referring to objects, *concepts* referring to sets of individuals, and *roles* representing binary relations between individuals. Complex composite descriptions can be formulated using *constructors*. For instance, RED-BALL $\equiv$ (*and*

(**prim** BALL) (**fills** *color* RED)), where *concept* RED-BALL is defined using *constructor* “and” to combine primitive *individuals* “BALL” with the description that *role* “color” is filled by *individual* RED. The main inference mechanism in DLs is subsumption that refers to the decision whether one concept is more general than another, and is used to capture concept hierarchy in product configuration tasks. Examples of logic-based systems involve Classic [209], Prose [210], Beacon [160], and constructive problem solving [96].

Resource-based approaches [75, 178, 155], another type of module-based configuration systems, provide a producer-consumer model, where interfaces between systems, components and the environment are represented by abstract resources [75], and each element is characterized by the amount and type of resources it supplies and consumes [155]. The final configuration solution is acceptable when the overall resources reach a balance state, in which all requirements are satisfied. A configuration starts with resources required by the environment subject to requirements. A resource type that is not balanced yet is selected and component types that can supply the resource are instantiated. This process repeats until the required amount of resources are not more than the supplied amount. Backtracking is performed in case of dead end.

Due to its flexibility and generality, constraint-based product configuration has been one of the most important model-based approaches [124], which models a configuration task as a Constraint Satisfaction Problem (CSP) [124]. A CSP is formulated as a set of variables with corresponding domains, and a set of constraints restricting the values that the variables can simultaneously take. Product configuration knowledge including requirement specifications, design consideration, and product data can be easily modeled as a CSP with the concept of variables, domains and constraints. Thereafter, various solving algorithms can be implemented on a CSP, independently on the configuration knowledge. A final configuration is a solution to the CSP in which all variables are assigned values such that all constraints are satisfied.

The evaluation of some configuration methodologies introduced above is listed in Table 2.1. Through the evaluation, it is recognized that the CSP-based product con-

Table 2.1: Product configuration approaches: advantages and limitations. [40, 80, 81]

Approaches		Advantages	Challenges
Case-based system		Domain knowledge stored in historical cases, no knowledge acquisition phase.	Dependent on past cases; may results in inaccurate result; not suitable for new techniques.
Rule-based system		Easy & intuitive in representing design knowledge in early stage; representing & evaluating reasoning heuristic relations.	Lack of separation between design knowledge and reasoning strategy, knowledge maintenance very difficult.
Grammar-based system		Modeling task as structured system; excel in complex system modeling & solving; visual representation; formal definition; ease of computational implementation; extensibility.	Difficult in modeling production rules properly, & complex relationships bw platform & products by simple graph grammar; generation process needs to be controlled separately
Model-based system	Logic-based system	Clear semantics; simple logical operations. Easy & efficient knowledge acquisition & maintenance; construct concept hierarchy by subsumption	Trade-off between reasoning efficiency & knowledge-representation expressiveness; Unable to represent complex systems as restricted expressiveness
	Resource-based system	Straightforward; provide a producer-consumer model based on resources; efficient in function-component matching.	No explicit representation of compositional relations between objects; Unable to handle technical constraints, and structural & specific placement requirement
	CSP-based system	Domain independent; generic & flexible in modeling knowledge; explicit task description; able to include customer decision	Potential complexity in reasoning; i.e., consistency check; completeness of configuration modeling is to be improved.

figuration excels in its generality and flexibility with sufficient robustness and accuracy in modeling and solving the domain problem. Despite its potential drawback in solving efficiency, the complexity can be reduced with proper problem representation and efficient solving algorithms. The detailed formalism and solving algorithms of CSP and CSP-based configuration systems will be further discussed in Section 2.4.

2.4 CSP-Based Configuration

As an attempt to represent a product configuration task as a generic and domain-independent model, CSP-based configuration has been a promising approach in mod-

eling and solving product family design problems. The reasons are threefold:

- It is highly declarative in design knowledge representation. CSP representation is close to original problems: domain objects directly map to the variables of CSP, and constraints can be expressed without being translated into specific forms, such as linear inequalities for a mathematical programming problem [14]. Thus the formulation is simpler, the solution is easier to understand, and the solution acquirement is more straightforward.
- It is domain independent. As the design knowledge is modeled as a CSP with the concept of variables, domains, and constraints, various solving algorithms can be applied directly on the CSP, without considering the design domains.
- CSP solving algorithms are relatively mature and simple, compared to, such as, integer programming methods [14]. They are able to reduce searching space more efficiently and find solution more quickly.

In this section, some basic definitions and solving algorithms of CSP are introduced in Section 2.4.1 and Section 2.4.2 respectively. Section 2.4.3 introduces and assesses some methodologies of CSP-based configuration.

2.4.1 CSP Definitions

Definition 2.1 [14, 104]: A **Constraint Satisfaction Problem (CSP)** is a triple $P = \{ V, D, C \}$, where

- $V = \{v_1, v_2, \dots, v_n\}$ is the set of variables called the domain variables;
- $D = \{Dom(v_1), Dom(v_2), \dots, Dom(v_n)\}$ is the set of domains. Each domain is a non-empty set containing all possible values for the corresponding variable;
- $C = \{c_1, c_2, \dots, c_n\}$ is the set of constraints restricting the values that the variables can simultaneously take. A constraint c_i is a relation defined on a subset $v_i, \dots, v_k$ of all the variables; that is, $\{Dom(v_i), \dots, Dom(v_k)\} \supseteq c_i$.

Definition 2.2 [104]: A **constraint** is defined on a subset of all variables, $v_i, \dots, v_k$, of the CSP, and restricts the values of the variables that they can take simultaneously. It can be classified as **unary**, **binary**, or **n-ary** in terms of the number of variables involved. That is, a constraint involves one, two, or more variables respectively.

Definition 2.3 [104]: An **assignment** is a mapping from a variable to its corresponding domain. Let v_i be a variable and Dom_i be its domain. The process that v_i takes a value $d_i \in Dom_i$ is called an assignment, denoted as (v_i, d_i) . When all variables are assigned a value, the assignment is **complete**, otherwise **partial**.

Definition 2.4 [104]: **Consistent assignment**: An assignment is consistent if each of its variables gets a value in its domain such that the assignment satisfies all constraints.

Definition 2.5 [14, 199]: A **solution** to a CSP is a complete assignment that satisfies all the constraints; i.e., a consistent complete assignment.

Definition 2.6 [13, 14, 104]: **Constraint Satisfaction Optimization Problems (CSOP)** consists of a standard CSP (defined in Definition 2.1) and an objective function $Z(V)$ that is defined in terms of some or all of the variables, and maps every solution to a numerical value. The goal is to find such a solution that is optimal regarding the objective function; namely, the solution minimizes or maximizes the objective function.

The goal of a CSP is solution finding. It refers to find either all solutions; or just one solution; or an optimal, or at least a good solution, measured by objective functions; or non-solution conditions.

2.4.2 CSP Solving Algorithms

A CSP is essentially a combinatorial problem that can be solved by search. There are two categories of CSP solving strategies [104]: *systematic search* and *heuristics*.

While a systematic search strategy assigns consistent values to variables one by one through a systematic way, heuristic strategy assigns values randomly regardless of constraints, and then repairs the assignments violating constraints.

As **systematic search algorithms** can solve all CSPs by searching systematically through all possible assignments, they are guaranteed to find a solution, or to prove that the problem is insoluble, and are therefore sound and complete. The main disadvantage is that they may be very time-consuming [14]. They can be further classified as follows [14]:

The first kind searches the assignments of all variables, until consistent complete assignments are found. In ***Generate-and-Test (GT)***, each possible variable value combination is systematically generated and then tested whether it satisfies all the constraints. The consistent combinations are solutions. However, the number of combinations generated is the size of the Cartesian product of all the variable domains [99]. In addition, it is inefficient as it generates many wrong assignments rejected in the testing phase.

The second kind extends a partial consistent assignment for some variables toward a complete assignment, by repeatedly choosing a value for other variables consistent with the current partial assignment [14]. One of such algorithms is ***Backtracking (BT)*** that can be viewed as a merge of generate and test phases in GT. In this method, variables are instantiated sequentially and as soon as all the variables of a constraint are instantiated, its consistency is checked. That is, it detects the inconsistency as soon as it appears and, thus, is more efficient than GT approach. If a partial assignment violates any of the constraints, then backtracking is performed to the most recently instantiated variable that still has available alternatives. Consequently, a subspace is eliminated from the Cartesian product of all variable domains. Although BT is better than GT, its complexity for most problems is still exponential. There are three major drawbacks of BT [14]: First is thrashing. It refers to repeating failure due to the same reason, because BT does not identify the real reason of the

conflict, namely the conflicting variables. Therefore, search in different parts of the space keeps failing for the same reason. The second drawback is having to perform redundant work, since the conflicting values of variables are not remembered for the immediate detection of the same conflict in a subsequent search. Finally, BT detects conflicts too late as it is not able to detect conflicts before conflicts really occurs; i.e., after assigning the values to the variables of the conflicting constraint.

Consistency techniques have been applied to address the issue of late detection of the conflict in BT. This approach forward checks the possible conflicts and removes inconsistent values from variables' domains [13, 14]. It may effectively prune many inconsistent assignments at the early stage, and thus shorten the search for consistent assignment. The names of consistency techniques are derived from graph notions. The structure of a binary CSP, whose constraints are unary or binary, can be expressed by a constraint graph [13, 14, 104, 99, 199]. That is, variables are represented with nodes, whose labels represent the domain of corresponding variables, binary constraints between them are represented with edges, and unary constraints are expressed by cycle edges.

A node representing variable X is **node consistent** if and only if for each value in the current domain of X , all unary constraints on X are satisfied. Node inconsistency can be eliminated by simply removing values that do not satisfy unary constraints on X from its domain. The most widely used consistency technique is **arc consistency (AC)** that checks constraint consistency between two variables. An arc (V_i, V_j) is arc consistent if and only if for each available value $x \in Dom(V_i)$, there is some value $y \in Dom(V_j)$ such that $V_i=x$ and $V_j=y$ is permitted by the binary constraint between V_i and V_j . The consistency of arc (V_i, V_j) can be achieved by deleting those values from $Dom(V_i)$ for which there does not exist a permitted value in $Dom(V_j)$, such that the binary constraint between V_i and V_j is satisfied. The basic algorithm in AC is to repeat revising arcs until a consistent state is reached or some domains become empty [99].

Although consistency algorithms remove inconsistency in advance, there still exist inconsistent values and the algorithm do not eliminate the need for search in general. Therefore, they are rarely used alone to solve CSPs completely, and are generally applied either as a preprocessing step before a search process, or as a propagation step after every assignment during search [13].

Constraints propagation combining systematic search and consistency techniques is a more common solving strategy of CSP. There are two of such schemes: *Look back schemes* that check consistency among already instantiated variables, such as backtracking, backjumping, and backchecking; and *Look ahead schemes* that are proposed to prevent future conflicts, such as forward checking and full look ahead.

Backtracking (BT) can also be viewed as a look back scheme. It performs arc consistency checks among already *instantiated variables*. To avoid the issues of BT, other look back schemes were proposed. **Backjumping (BJ)** is a method to avoid the issue of thrashing in BT. The process of BJ is exactly the same as BT, except for their different processes for looking back. Whenever a constraint is violated, BT moves one step back, just to the immediate past variable, while BJ jumps to the instantiated variable where inconsistency occurs [104]. To address the issue of redundant work in BT, **Backchecking (BC)** is proposed, which is useful for reducing the number of compatibility checks [13, 14]. For example, if BC finds that an assignment (Y,b) is inconsistent with assignment (X,a), then it remembers this incompatibility. As long as (X,a) is still in the partial assignment, (Y,b) will not be considered again.

All look back algorithms share the disadvantage of late detection of the conflict. Therefore look ahead methods are proposed to prevent future conflicts.

Instead of checking AC between instantiated variables, **Forward Checking** performs AC between *an un-instantiated variable and an instantiated variable*. Consequently, for every un-instantiated variable, there exists at least one of its values that is consistent with the values of instantiated variables [14]. Whenever a variable X is assigned; i.e., (X,a), forward checking performs AC check on each unassigned variable

Figure 2.2: Comparison of propagation techniques [13].

Y directly connected to X by a constraint, and temporarily removes from Y's domain values inconsistent with (X, a) [13, 199]. If Y's domain becomes empty, then it is known that the current partial assignment is inconsistent. Thus, forward checking prunes the search space that lead to a failure earlier than BT. Further, AC checking can be extended to latter variables that have no direct connections with instantiated variables. **Full Look Ahead** checks AC *between a future variables and an instantiated variable*, and *between future variables*. Therefore, it will further removes possible conflicts, and reduces more search space.

Figure 2.2 illustrates a comparison among the constraint propagation methods mentioned above in term of which constraints are checked when a value is assigned to current variable. More constraint propagation at each node leads to smaller search space, but the overall cost may be higher, since the work at each node may be more expensive [13].

Heuristic strategy is another strategy of CSP solving algorithms. It assigns values randomly regardless of constraints, and then repairs the assignments that violate constraints. However, it may be ineffective for some large size and tight constraint problems.

Variable Ordering is one of the main concerns in heuristic strategy, which may dramatically affect the efficiency of the solving algorithms introduced previously. For instance, the variable order for instantiation has substantial impacts on the complexity of backtrack search [14]. In look ahead algorithms, variable ordering can affect the size

of search space pruned. The ordering may be *static ordering* where the variable order is predefined before search, and not changed thereafter, or *dynamic ordering* where the choice of the next variable to be assigned depends on the current state of search. The most common approach for variable ordering is *minimum remaining values* heuristic in which the variable with the smallest domain is selected for assignment since it is most likely to cause a failure soon, thereby pruning the search space. Another method, called *most constrained heuristic*, chooses the variable that is involved in the largest number of constraints on other unassigned variables.

Once a variable with several available values is selected to be assigned, **value ordering** has a substantial impact on the efficiency for finding the first solution. If a right value is assigned to each variable, then the problem may be solved without backtracking. However, if all solutions are required or there is no solution, then the value ordering is indifferent [14]. *Least-constraining-value* chooses the value maximizing the number of options available for future assignments. Another heuristic prefers the value that leads to an easiest work to solve CSP, which requires to estimate the difficulty for CSP solving.

Local Search heuristic gradually alters inconsistent assignments to all the variables to final complete solutions. *Hill-climbing* is the most famous algorithm of local search [13]. Starting from a random assignment of variables, it changes the value of some variables at each step such that the resulting assignment satisfies more constraints, and stops when all constraints are satisfied, or some resource is exhausted; i.e., a global minimum. To avoid exploring the whole state's neighborhood in hill-climbing, *min-conflicts* heuristic [13, 199] chooses randomly a conflicting variable and then selects a value that minimizes, or at least does not increase, the number of violated constraints.

In many real-life applications, we do not want to find all solutions or the first solution, but a good solution, whose quality is measured by *objective functions*. That is to solve a constraint satisfaction optimization problems [13, 14, 104]. The most

popular method for CSOP is **Branch and Bound (B&B)** [104]. In B & B, there is a heuristic function $F(V)$ that maps a partial assignment to a numerical value. It is treated as an estimate of the objective function $Z(V)$ applied to a solution (i.e., a complete assignment generated from the partial assignment) [14]. Specifically, it represents an underestimate of $Z(V)$ in case of minimization, and an overestimate of $Z(V)$ for maximization problems. The searching process of B & B is in a depth first manner, and is similar to that of BT algorithm except that after each assignment, the value of $F(V)$ for the assignment is computed, and is compared with a bound that is initialized to be plus infinity. If the value is lower than the bound, then the bound is updated to the value, and the assignment is the best solution found so far. If this value exceeds the bound, then the subtree under the current partial assignment is cut off immediately [13]. However, finding an optimal solution is usually computationally expensive. Using a second “acceptability” bound, B&B can also be adopted to find sub-optimal solutions. If a solution is better than the acceptability bound, then it can be accepted even if it is not proved to be optimal. **Dichotomic Search Method** [104] improves the search efficiency over B&B. It starts with a lower bound L that can be any arbitrary value reasonable for the problem, and an upper bound U that is an initial solution to the problem, on objective function $Z(V)$. A binary search procedure continues until the searching space is exhausted. The last lower bound is taken as the optimum solution.

2.4.3 CSP-Based Configuration

The first attempt to apply CSP to product configuration tasks was presented by Mittal and Frayman [124]. In their approach, a product is described as a composition of a fixed, pre-defined set of components, each of which possesses a set of properties and ports for connecting to other components. All of the components, properties, and ports are represented as variables in a CSP with discrete and finite domains [124, 187]. Restrictions on how to combine various components for a valid configuration are represented as compatibility constraints. Thus the solution to the CSP-based product

configuration task is the required configuration. It has been proved that **the classical CSP-based configuration system** is simple and domain-independent for representing configuration problems. However, due to the assumption of a fixed and pre-defined set of components, this approach suffers from the limitation in representation for more complex configuration problems where the number of components is not known *a priori*, which is a nature of configuration tasks. Moreover, the classical model fails to capture the inherent structure of components in PF.

To address the first issue of the classical CSP-based configuration system, **dynamic constraint satisfaction problem (DCSP)** based model was introduced [123]. The main idea is that only a subset of all the variables need to be instantiated in the solving process. Each variable holds one of two states: active or inactive. And only active variables are involved in final solutions [151, 155]. Besides compatibility constraints, activity constraints are proposed to specify conditions on the values of active variables, under which variables become active. In general, a DCSP is defined as $P = \{V, D, V_i, C_c, C_a\}$, where V and D refer to the set of all active and inactive variables, and corresponding discrete and finite domains; $V_i \subseteq V$ is the non-empty set of initial active variables and has to be assigned in every solutions [65]; C_c and C_a are the set of compatibility and activity constraints respectively. Mittal and Falkenhainer [123] offered 4 types of activity constraints; i.e., *require variable*, *always require*, *require not*, and *always require not*. Haselbock [72, 151] employed same activity constraints in a constraint-based configuration framework *ConfCS*. Bowen and Bahler [23] discussed a formal, mathematically well-founded treatment of the conditional existence of variables. Gelle, Sabin *et al.* [64, 65, 66, 154] renamed DCSP as **conditional CSP (CondCSP)**, and distinguished two types of activity constraints; i.e., *activity of inclusion* and *activity of exclusion*.

The main advantage of DCSP over the classical CSP-based model is that it allows inferences about variable activity [155]. In DCSP, the solving process starts with an initial set of active variables. Other variables are activated into the solving process

as the result of satisfying activity constraints. CSP is thus constructed dynamically in a search process. Correspondingly, a constraint is either “active” if all its variables are active, or otherwise “inactive”. Only active constraints are checked in DCSP solving process. All of these avoid irrelevant work during search [155]. However DCSP approach still suffers from the limitation in representation of the component structure and the hierarchical nature of the domain knowledge in general configuration tasks.

Generative CSP [180] is proposed to extend the DCSP-based configuration system. It consists of a possible infinite set of component variables with corresponding domains, each of which is a finite set of component types, and a finite set of generic constraints that are specified on a meta level; i.e., component types rather than component instances, and are able to capture the relationship between properties and components, component types, and component creation. This approach does not require the declaration of the complete set of variables beforehand, and is able to deal with multiple occurrences of components with similar behavior [155]. Besides activation constraints on a meta level, resource constraints are specified to express monotonic aggregate relations on resource properties of components that are not known in advance. Similar system can be found in the constraint-based configuration framework *ConfCS* proposed by Haselböck [72, 155]. Generative CSP-based system is more generic, as it enhances expressiveness by allowing a infinite set of variables and defining constraints at meta level.

Freuder also handled CSP at a meta level. In his approach[56], a configuration problem (a metaproblem) is decomposed into subproblems (metavariables) [56, 151], each of which includes a subset of the variables in the original problem, together with corresponding domains and associated constraints. Metavariables can overlap by sharing common variables. A metaconstraint between two metavariables must implement all the original constraints that include variables from the corresponding subproblems, and must ensure that the variable shared in metavariables is assigned

the same value (metavalue) in the solution. This approach can reduce the complexity of solving a problem by solving an equivalent problem that is modeled at a different level of abstraction, and since either the metaproblem or the metavariable may have a special structure, they can be solved more efficiently. However a metavariable is associated with a fixed, static subproblem.

In many cases, the values of variables can be organized in a specialization/generalization hierarchy by clustering in terms of common properties and relations [73, 151]. A CSP is a **hierarchical domain CSP** if the domains of all its variables are organized hierarchically. Arc-consistency algorithms can be applied in hierarchical domain CSP subject to the domain hierarchy [98, 107, 151]. A breadth-first linearization of the directed-acyclic graphs is used to represent the hierarchical domains, and deploys a partial order on these flattened domains [151]. There are two inferences on the values of variables, which allow the algorithm to avoid a potentially large number of constraint checks: in the domain hierarchy, if a higher-level value is arc inconsistent, then so are all its descendants; and if a lower-level value is arc consistent, then so are all its ancestors.

Although the hierarchical domain CSP addresses the issue that classical and DCSP based models fail to capture the structural nature of domain knowledge, its space complexity of the optimal worst case for a CSP including two variables with a domain size of d and a binary constraint is $O(d^2)$ [151]. Since in real applications, a configuration problem may involve hundreds of values for each variable and thousands of constraints, at this space complexity this approach is very difficult to represent the entire problem.

A **hierarchical CSP** [177] was proposed for dimensional inspection planning of coordinate measuring machines. In this approach, hierarchical constraints are used to represent requirements for optimization, and are ordered in terms of the restrictiveness of constraints that is determined by structural relationships among variables. A solution to the hierarchical CSP should satisfy all mandatory constraints and minimize the number of violated optimization requirements in the constraint hierarchy;

i.e., a solution should violate the least amount of the least restrictive hierarchical constraints, and then the least amount of the second least ones, and so forth. Similar to the hierarchical domain CSP, the knowledge acquisition of this approach is an expensive procedure involving geometric computations. *ConfCS* [72] also integrated type abstraction, and discussed how hierarchical reasoning can be integrated in the constraint-based system.

Composite CSP [151] was proposed to extend classical CSP model, where values for variables can be not only atomic value, but also entire subproblems that may have certain hierarchical structures. It also enables the CSP to change dynamically. That is, when a variable is instantiated, the variables and constraints representing the composite value are created and added to the constraint network. Several specific CSP algorithms have been applied on composite CSP-based systems for better solving efficiency by utilizing both the hierarchical and compositional structure of the problem and maintaining local consistency of the constraint network during search [57, 152]. Such as object-oriented programming offers a natural implementation framework for composite CSP [153].

By organizing the domain knowledge based on aggregation and classification, the composite CSP model is more powerful and more efficient in representation [155], but limited to fewer constraints, smaller domains. Its dynamic nature is helpful in cutting down the space complexity, which is a key issue in the hierarchical domain CSP, by considering only a small part of the entire problem at one time. However, this approach only considers the compatibility constraints with discrete domain variables. Considering other constraint types and continuous variables in this approach may result in high complexity.

Although generative CSP has been extended to a infinite set of variables, it is still limited to model variables with discrete domains. In real applications, continuous variables, such as component parameters, with a range of possible values are usually taken into account. *Aldanondo et al.* [6] studied requirements for modeling continuous

variables and numerical constraints over both discrete and continuous variables, and concluded that very few commercial configuration systems support continuous variables. *Xie et al.* [212, 213] proposed a CSP model for engineering product configuration tasks, which specifies three types of constraints; i.e., compatibility, equality and inequality constraints. Represented by mathematical expressions or computable procedures, equality and inequality constraints reflect the numeric relationships among both discrete and continuous variables. Gelle and Faltings [64, 65] offered a solving method for CondCSP involving discrete and continuous variables, which derives several classical CSPs from a mixed CondCSP, and applies local consistency to prune their solution spaces respectively.

Most CSP-based configuration systems traditionally focus on binary constraints. Although a n-ary constraint can be equally transformed, in theory, to several binary constraints, the transformation incurs a significant increase in the problem size, so it may not be practical [16], particularly in large tasks. Furthermore a forced binarization results in unnatural formulations that cause extra difficulties in CSP solving [18]. How to handle n-ary constraints efficiently has attracted more attentions. In their CSP model for engineering product configuration, *Xie et al.* [212, 213] discerned two types of variables; i.e., dependent and independent variables, which reflect dependent relationships among variable during the assignment. Based on this distinction, they proposed an updating mechanism for variables in n-ary constraint consistency check using a topological ordering of a directed acyclic graph. Salido [156] presented a preprocess heuristic for n-ary constraints called constraint ordering heuristic that calculates the constrainedness of the problem and classifies the constraints so that the tightest ones are studied first.

Global constraints has also been an interesting topic, which generally involve parameters, and are defined on any number of variables [17]; i.e., it can be n-ary constraints involving both discrete and continuous variables. For example, *alldifferent* constraint, *Among* constraint, *Sum* constraint [17], resource constraint [151], *atmost*

constraint, and *diffn* constraint [169]. Global constraints allow more constraint propagation than a set of primitive constraints, and simplify modeling of problems, especially large scale models, by offering high level constraint abstractions. However, they also adds complexity. Bessière *et al.* [17] provided a generic framework for automatically learning implied global constraints with parameters according to the actual domains of variables, not just from structural or syntactical properties, and applied the framework to both discrete and continuous properties. Similar work is offered by Wu and Hsiao [211]. Simonis [169] studied detailedly the modeling and solving of global constraints in three different applications.

Besides the CSP-based systems introduced above, some CSP-based commercial configuration tools have been developed. *COCOS* [181] is a C language tool for constraint-based and dynamic configuration. It focuses on both reasoning about the structure of systems and the declarative description of the different types of configuration knowledge. *Choco* [200] is a Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. *ILOG* Configurator [17, 80, 202] is a C++ library based on the ILOG Solver, and provides the possibility of generating new components at runtime. Another configurator system based on a constraint solver is *Selectica* [80], which can model component taxonomy, constraints and rules.

2.5 Discussion

From the classical CSP-based configuration systems to its various extensions, the expressive power and efficiency in modeling configuration tasks have been improved. However, the CSP-based configuration modeling is still under development to enhance the completeness. In general, there are four limitations of the CSP-based configuration system:

- Its knowledge representation is not complete. In product configuration tasks, a huge amount of data with great variety need to be represented, which consist of

not only product data (e.g., resource data, production data, and cost data), but also customer requirements and configuration knowledge (e.g., part combination rules, technical processes). Thus, powerful knowledge representation models are necessary to capture the large variety and complexity of data. Semantics losing also exists in the CSP representation in the product configuration task.

- Hierarchy information is still not expressed effectively. As mentioned previously, the structure of modules/components is a inherent nature of PF design. Although many efforts have been made to improve the expressiveness for hierarchy information, in most models it is still unclear, or too complex.
- Integration of cost into product configuration is still insufficient. Although cost is a critical criteria for a successful product configuration, the integration of product configuration and cost estimation has not been solved successfully, since cost estimation is generally implemented after the finish of configuration.
- Computational efficiency needs to be improved. Sometimes the CSP representation for the configuration task hinders the solving efficiency. Although there are diverse CSP solving algorithms, the solving efficiency for specific configuration problems is still hardly satisfied.

In the next chapter, a novel framework will be proposed, which efficiently models the heterogeneous data and captures the inherent hierarchy information in PF design. It also provides a scheme to integrate cost estimation into product configuration process concurrently.

2.6 Summary

In this chapter, the basic concepts of product family, the approaches for product family modeling, and the metrics of product family design are first comprehensively introduced. Thereafter, various product configuration techniques for automatically

achieving product configuration results in PF are reviewed. Their advantages and limitations are also assessed. As a promising product configuration approach, CSP-based product configuration approach is introduced in detail, together with the basic definitions and related solving algorithms of CSP. The advantages and limitations of several CSP-based systems are also evaluated. Finally, a discuss of CSP-based product configuration approach is provided.

Chapter 3

DCSP-Based Product Configuration System with Cost Estimation

3.1 Overview

As we have introduced in previous chapters, in a mass customization environment, product family design has been applied to increase product variety to satisfy diverse customer requirements while maintaining near mass production efficiency [54] such as low cost and fast time-to-market. The core of a product configuration task is to select and arrange combinations of component variants satisfying given specifications on the basis of available product components and predefined rules for component composition [180]. The solution of product configuration is to produce a list of selected components and, more importantly, the structure and topology of the product that satisfy the configuration requirements.

To respond to surging concerns on after-sale services, disposal and recycling, and environmental issues, product lifecycle management has been practiced to manage the product development process. Thus, product family design requires a coherent solution that encompasses the entire product lifecycle [168].

As cost is an important composition and criterion for both product configuration and PLM [102], proper cost estimation before final configuration is critical to production management. As product design influences about 70% to 85% of the total

Figure 3.1: Constraint-based configuration system flowchart.

product cost [134], separating cost estimation from product configuration reduces the effectiveness of both tasks. Hence, by integrating proper cost estimation into product configuration, a product is designed with optimal combination of functionality, quality, and cost. Although much efforts have been made in product family design [39, 40, 87] and cost estimation [106, 148, 161], the integration of these two tasks has not been satisfactorily addressed.

Problem representation and problem solution are the two main issues in product family design. A large amount of data with great variety in product family design needs to be represented consisting of not only product data such as the data of resource, production, and cost, but also customer requirements and configuration knowledge such as part combination rules, technical processes, and the relationship among design parameters and cost elements. Hence, powerful knowledge representation models are necessary to capture the large variety and complexity of data. Based on given configuration problem description and representation, efficient reasoning methods are also required for configuration solution search and optimization.

Constraint satisfaction problem has been recognized as a promising approach to model and solve product configuration and cost estimation tasks. In a CSP-based product configuration and cost estimation system, the data mentioned in the last paragraph are modeled by concepts in CSP (i.e., variables, domains, and constraints). As the data are traditionally stored in product data models (PDMs), formalisms of CSP need to be derived from existing PDMs mastered by manufacturers. A flowchart

of a CSP-based configuration and cost estimation process is illustrated in Figure 3.1. There are three key elements in the flowchart: *Product Data Model (PDMs)*, *CSP-based Model*, and *Constraint Solver*. PDMs are the fundamental elements of a CSP-based model. They capture the semantics and structure of requirements specification and design/cost knowledge, as well as various product data. A CSP-based model maps and reorganizes data in PDMs into certain formalism in CSP. Then the configuration and cost estimation task can be viewed as a CSP solving process. Thereafter in the CSP solver, various algorithms are implemented on a CSP-based configuration model to generate solutions without violating any constraints.

As introduced in the previous chapter, classical CSP is clear and declarative, but static. It is limited in expressiveness for the dynamism of a problem; i.e., the number of components may not be pre-determined. It also fails to capture the inherent hierarchy of components in a product family. Dynamic CSP (DCSP) based configuration system has been proposed to avoid the first problem of classical CSP by introducing activity constraints so that the components can be dynamically included. However, it still suffers limitations in the representation of components hierarchy.

In this chapter, I propose an extended DCSP-based configuration model that integrates PLCC estimation with product configuration, and allows incremental extension of a corresponding constraint network while capturing the internal structure in product family design. Using this model, designers are able to model elements in configuration and PLCC estimation; derive results of configuration and cost estimate at the same time; and utilize cost as an effective measure for configuration. To facilitate problem representation in DCSP terms, mapping relationships among product data in PDMs and the elements in the extended DCSP model is also analyzed in this chapter.

This chapter is organized as follows: Section 3.2 proposes the concept of a configuration domain of the extended DCSP-based model, which includes all possible variables and constraints in a product configuration and cost estimation task. Various constraints in the DCSP-based scheme are described in detail in Section 3.3.

Then a DCSP-based framework for product configuration and cost estimation tasks is established in Section 3.4. In Section 3.5, the mapping relationships among the elements of the DCSP-based model and some PDMs are discussed to facilitate the construction of the DCSP system. How to acquire compatibility and incompatibility constraints by association rule mining method is proposed in Section 3.6. Finally, a summary is presented in Section 3.7.

3.2 Configuration Domain in DCSP-Based Configuration System

In DCSP, variables involved in the final solution are dynamically activated from the original variable set depending on specific conditions. In our DCSP-based configuration system, active variables and associated active constraints whose variables are all active are derived from a predefined knowledge base called the configuration domain.

Definition 3.1 *Configuration Domain.* *A Configuration Domain Φ is described as $\langle V_M, V_{CM}, V_P, V_{CT}, D, C_s, C_a, C_g \rangle$, where*

- V_M : *A possibly infinite set of module variables that are the main constituents of a product family.*
- V_{CM} : *A possibly infinite set of component variables in product variants that are instances of the product family. These variables are instances of the corresponding module variables in the product family. Note that the domain of each variable from V_M is a finite set of component variables; i.e., $\text{Domain}(V_{M_i}) \subseteq V_{CM}$.*
- V_P : *A set of property variables that are associated with module and component variables. The domain of each variable from V_P contains numeric or symbolic values.*
- V_{CT} : *A set of cost variables that are associated with module and component variables. The domain of each cost variable includes non-negative numeric values.*

- D : A set of domains of the variables above. Each domain is a non-empty set containing all possible values of the corresponding variable.
- C_s : A finite set of structural constraints that capture hierarchical relationship among module variables.
- C_a : A finite set of activation constraints that represent possession relationship between V_M and V_P or V_{CT} , and are used to trigger inactive variables to become active.
- C_g : A finite set of generic constraints that govern the above variables. They involve all constraints in a product configuration and cost estimation task except structural constraints C_s and activation constraints C_a ; e.g., compatibility and incompatibility constraints, equality or inequality constraints, cost constraints, preference constraints, etc.

In this definition, modules of a product family, components of product variants, properties and cost drivers associated with modules and components are all modeled as variables in the DCSP model. In fact, the configuration domain Φ can be viewed as a classical CSP consisting of all the variables and constraints that may be used in product configuration and cost estimation tasks. In a specific product family, the common/optional modules and module hierarchy are fixed. To satisfy certain requirements, components of a product variant that correspond to the modules involving all common modules and selected optional modules in the product family are chosen and organized according to the hierarchy of modules that is captured by *structural constraints* C_s . Subsequently, properties and cost drivers associated with selected modules and components are activated by *activation constraints* C_a . In the final configuration, all the properties of selected components are filled with values so that the product variant meets customer requirements.

Here the description of the name of property variables is provided, which is composed of the name of the originating module that owns the properties and the property

name; e.g., if property $p \in V_P$ is defined for module $m \in V_M$, then the name of the property variable is $m.p$. The set of all possible property variables can thus be specified as $V_P = \{m.p \mid m \in V_M, p \in V_P\}$. Moreover, as each component $c \in V_{CM}$ is an instance of corresponding module $m \in V_M$, c inherits all properties of m . The property $p \in V_P$ for component c is named $c.p \mid (c \in V_{CM} \text{ and } c \in \text{Domain}(m), p \in V_P)$. Similarly, the set of all possible cost variables can be specified as $V_{CT} = \{m.t \mid m \in V_M, t \in V_{CT}\}$ and the name of a component's cost variable that inherited from corresponding module is $c.t \mid (c \in V_{CM} \text{ and } c \in \text{Domain}(m), t \in V_{CT})$.

3.3 Constraints in DCSP-Based Configuration System

In DCSP-based product configuration and cost estimation system, constraints may involve logical restrictions, physical restrictions, compatibility restrictions, safety regulation or other requirements [212]. As part of the engine of the DCSP solving process, they control the operation of variables for a valid solution, such as stipulating the rules that product configuration follows, setting the relationships among variables, and restricting the combination of values that variables can simultaneously take. In this section, constraints in the extended DCSP-based model are proposed.

3.3.1 Structural Constraints

In product family based configuration, the inherent hierarchy of modules/components is an important issue as it reflects and utilizes the nature of modularity, commonality and variety in product variant generation within a product family. A product family organizes, with certain structure, common modules that are mandatory during product variant generation and differentiation modules that are optional during product variant generation. Typically, in a PFA, a product family can be represented as a module tree including a root module, several internal modules and leaf modules, as illustrated in Figure 3.2. The root module and internal modules have at least one

Figure 3.2: A product family architecture (PFA).

direct child module. In the extended DCSP-based model, *structural constraints* C_s are introduced to capture hierarchical information among modules that can be viewed as an abstract level of component structure in product variants.

Definition 3.2 Structural Constraints. Let $m_1 \in V_M$ be a non-leaf module in a PFA, $m_{11}, \dots, m_{1n}, m_{1n+1}, \dots$, and $m_{1t} \in V_M$ be a set of direct child modules of m_1 . Such a structure is expressed by structural constraint C_s of the form

$$C_s : m_1 \Downarrow \{m_{11}, \dots, m_{1n}, \sim m_{1n+1}, \dots, \sim m_{1t}\} \quad (3.1)$$

where symbol $\sim$ before a child module indicates that the module is a differentiation module that is optional in product variant generation; e.g., m_{1t} , and modules without symbol $\sim$ are common modules that are mandatory; e.g., m_{11} .

For example, in a desktop personal computer system, shown in Figure 3.3, module “Accessory” includes four child modules consisting of two common modules “Mouse” and “Keyboard” and two differentiation modules “Webcamera” and “Speaker”. The corresponding structural constraint is expressed as follows: Accessory $\Downarrow \{\text{Mouse}, \text{Keyboard}, \sim \text{Webcamera}, \sim \text{Speaker}\}$.

Definition 3.3 Activation of Structural Constraints. Let C_s be a structural constraint of the form $m_1 \Downarrow \{m_{11}, \dots, m_{1n}, \sim m_{1n+1}, \dots, \sim m_{1t}\}$, C_s is said to be active if and only if module variable m_1 is active.

For general constraints, only when all the variables involved in the constraint are active, the constraint is said to be active. Structural constraints C_s are constructed

Figure 3.3: The structure of desktop personal computer.

to capture the hierarchy of product configuration and applied to activate direct child modules of a module based on the hierarchy in the solving process. Only when a module m_1 is active are its direct child modules activated. Hence, when m_1 is active, related structural constraints whose mother module variable is m_1 should be active; other variables (i.e., child module variables) are not active. The child modules can be activated by structural constraint depending on specific requirements.

Definition 3.4 Consistency of Structural Constraints. Let C_s be a structural constraint of the form $m_1 \Downarrow \{m_{11}, \dots, m_{1n}, \sim m_{1n+1}, \dots, \sim m_{1t}\}$. C_s is said to be satisfied if and only if for any active module variable m_1 , all its common child modules $m_{11}, \dots, m_{1n}$ are active and some or none of its differentiation child modules $m_{1n+1}, \dots, m_{1t}$ are active, depending on specific requirements.

In product family design, product variants are derived from the corresponding product family by instantiating all common modules and some or none differentiation modules. Hence, the hierarchy of components in a product variant can be viewed as an instance of the hierarchy of modules in the corresponding product family. Consequently, structural constraints can also be applied to the corresponding components in product variants.

While capturing hierarchical information in product family straightly, C_s uses existing module variables instead of additional port variables in some methods, such as the classical and dynamic CSP-based configuration system proposed by Mittal *et al.* [124, 123], which results in less search space and higher searching efficiency.

Moreover using C_s in the solution search process, the activation of module variables follows the hierarchy of a product family. Higher level module variables in a module tree, such as m_1 , and corresponding components are checked for consistency earlier than the lower level ones, such as m_{11} . This can greatly improve the solving efficiency of the configuration process because if the consistency of lower level modules (e.g., m_{11} - m_{1n}) are checked before higher level ones (e.g., m_1), and later when an assignment of a higher level module is inconsistent, backtracking can be performed to remove each previous assignment to all its direct or indirect lower level modules. Similar conditions exist among components. The backtracking process is tedious and time-consuming.

3.3.2 Activation Constraints

In a DCSP, each variable holds one of two states: active or inactive. Only active variables are assigned values in the solving process, and are involved in the final solution. In product family based configuration, modules possess sets of properties and cost drivers. Correspondingly, in a DCSP-based configuration system, the non-predetermined variables involved in the final solution can be activated based on such possession relationships. That is, when a module variable is selected into a DCSP for specific requirements, an appropriate set of property and cost variables associated with the selected module are also introduced into the DCSP to be assigned specific values satisfying certain requirements. Such derivation relationships among the variables in the DCSP-based configuration model are expressed by *activation constraints* (C_a).

Definition 3.5 Activation Constraints. *Let $var_1 \in V_M$ be a module variable and $var_2 \in V_P \cup V_{CT}$ be a property variable or cost variable respectively. The derivation relationships among the variables in the DCSP-based configuration model are expressed by activation constraints C_a in the general form of*

$$C_a : var_1 \implies var_2. \quad (3.2)$$

More specifically, based on configuration domain $\Phi = \langle V_M, V_{CM}, V_P, V_{CT}, D, C_a, C_s, C_g \rangle$, let $m \in V_M$ be a module, $p \in V_P$ be a property of m , $t \in V_{CT}$ be a cost variable

of m . The activation constraints are represented as

$$m \implies m.p, \text{ and } m \implies m.t. \quad (3.3)$$

Activation constraints provide conditions under which inactive variables become active. Using activation constraints, only specific properties and costs variables associated with the selected modules are dynamically added to the active variable in the solving process of the DCSP model and are involved in the final solution with compatible values. Thus, the number of variables is generally less than, or equal to in the worst case, that in the classical CSP-based configuration system.

Note that within a product family, activated property variable $m.p$ and cost variable $m.t$ of module m are all inherited by m 's instance component c ($c \in \text{Domain}(m)$) in a product variant; that is $c.p$ and $c.t$. In the final product configuration, only $c.p$ and $c.t$ will be assigned with certain values for a specific product variant satisfying customer requirements.

Definition 3.6 *Activation of Activation Constraints.* Let C_a be an activation constraint of the form $var_1 \implies var_2$. C_a is said to be active if and only if module variable var_1 is active.

Similar to structural constraints, activation constraints are applied to activate property and cost variables of a module in the solving process based on belonging relationships among them. Only when a module is active, its property and cost variables can be activated. In general, a constraint is active only when all its variables are active. However, an activation constraint becomes active when its antecedent variable var_1 is active. That is when module variable m is active, related activation constraints where var_1 refers to m should be active even when var_2 (i.e., property or cost variables of m) is not active.

Definition 3.7 *Consistency of Activation Constraints.* Let C_a be an activation constraint of the form $var_1 \implies var_2$. C_a is satisfied if and only if for all active

variables var_1 , var_2 is active. Specifically, C_a is satisfied if and only if for all active module variables m , its property variable $m.p$ or cost variable $m.t$ is active.

To avoid the case where the activation of var_2 is not triggered by var_1 , an assumption is made that in the DCSP-based configuration system, var_2 is active if and only if var_1 is active. For example, if a property variable $c.p$ is active, then component c must be active and its corresponding module m must also be active.

3.3.3 Compatibility and Incompatibility Constraints

In product configuration, compatibility constraints represent implication relationships among components in a product variant on diverse levels, such as compatibility of function, process, component alternatives, attribute value, etc. They specify consistent combinations of components of a valid configuration and thus, restrict the configuration space. One of the general forms of compatibility constraints can be expressed as an *IF-THEN* rule. For example, the constraint “If component a_1 is selected for module A , then component b_3 can be selected for module B ” is represented as

$$IF (A = a_1) THEN (B = b_3) \quad (3.4)$$

Compatibility constraints also generally take the form “ $X \rightarrow Y$ ”. For instance, Equation (3.4) can be expressed as $(A=a_1) \rightarrow (B=b_3)$. Compatibility constraints can also be represented as a set of all consistent combinations of variable values; that is, a set of compatible tuples

$$(var_{1i}, var_{2m}, \dots, var_{tl}) \in D_1 \times D_2 \times \dots \times D_t. \quad (3.5)$$

where var_{nk} represents the k th value for variable var_n ($n \in (1, \dots, t)$), $var_n \in V_M \cup V_{CM} \cup V_P$, and D_n is the corresponding domain for variable var_n .

For example, Equation (3.4) can also be expressed as $(A, B) = (a_1, b_3)$.

On the contrary, an incompatibility constraint defines inconsistent combinations of components. For instance, the following constraint

$$IF (A = a_1) THEN (C \neq c_2) \quad (3.6)$$

means that if component a_1 is selected for module A , then component c_2 cannot be selected for module C . Incompatibility constraints can also be expressed in the form “ $X \rightarrow !Y$ ”. For example, the constraint in Equation (3.6) can be denoted $(A=a_1) \rightarrow !(C=c_2)$. An incompatibility constraint can be transformed into a compatibility constraint. For example, if $Domain(C) = \{c_1, c_2, c_3\}$, the incompatibility constraint in Equation (3.6) can be expressed as a consistent combination of values such as $(A,C) = \{(a_1, c_1), (a_1, c_3)\}$. In fact, we are more interested in incompatibility constraints as they are stricter in terms of removing inconsistent value from variables’ domains, and are thus, more effective in search space pruning.

3.3.4 Equality and Inequality Constraints

Equality or inequality constraints are defined among properties of modules, and consequently are inherited by corresponding instance components of the module. They stipulate relationships among numerical or symbolic values of properties by mathematical expressions. The general form of equality or inequality constraints can be represented as follows respectively:

$$\text{Equality constraint : } Expression_1 = Expression_2 \quad (3.7)$$

$$\text{Inequality constraint : } Expression_1 \otimes Expression_2 \quad (3.8)$$

where $Expression_1$ and $Expression_2$ can be a property variable $p \in V_P$ or a mathematical expressions defined on property variables. $Expression_2$ can also be a constant. $\otimes$ refers to a binary operator, and $\otimes \in \{<, >, \leq, \geq, \neq\}$

For instance, “The color of mouse and keyboard must be the same” can be denoted as $Mouse.Color = Keyboard.Color$ where both sides of the operator “=” are property variables.

Another equality constraint “Memories’ total capacity equals the unit capacity of a memory chip multiplied by the number of memory chips” is expressed as $Memory.TotalCapac = Memory.Capac \times Memory.No$ where the righthand side is a

mathematical formula that computes the value of property variable *Memory.TotalCapac* in the lefthand side.

Inequality constraint “The number of memory chips is not more than 5” is formulated as $Memory.No \leq 5$ where binary operator $\otimes$ is “ $\leq$ ”.

Moreover, equality and inequality constraints usually induce compatibility or incompatibility constraints on components (proposed in Section 3.3.3). For instance:

- There are two modules A and B in a product family, each of which has two components: $A\{A_1, A_2\}$, $B\{B_1, B_2\}$. Module A has property “Length” and Module B has property “Diameter”;
- $A_1.Length = 13$; $A_2.Length = 8$; $B_1.Diameter = 11$; $B_2.Diameter = 6$;
- Inequality constraint: $A.Length + B.Diameter \leq 19$.

Then, we can transform the inequality constraint on properties into compatibility or incompatibility constraints on components as follows:

- $(A = A_1) \rightarrow (B = B_2)$;
- $(A = A_2) \rightarrow (B = B_1)$;
- $(A = A_2) \rightarrow (B = B_2)$; and
- $(A = A_1) \rightarrow \neg(B = B_1)$.

3.3.5 Cost Constraints

As we have mentioned previously, within a product family, product variants are typically derived from a common base product, and hence are with strong commonalities in functions. The differences may be in capacity, additional functions, unique features, and just some simple aesthetic attributes such as colors and shapes. This has a possibility that a part is used in a number of product variants. Most of the time, the major constraint on part selection is the cost. Particularly over the past two decades,

manufacturers have raced to invest heavily to strengthen their competitiveness in cost, quality, and delivery lead-time. As the race continues, further improvements increasingly rely on achieving overall performance in these competitive factors over the product life cycle. PLCC has become an important criterion for both product family design and PLM. However the integration of product configuration and cost estimation has not been successfully addressed. To incorporate cost estimation into the configuration task tightly, we extend our DCSP-based configuration system by introducing *cost constraints* that make use of cost functions in the form of mathematical expressions to quantify the relationships among cost drivers and design parameters in product configuration tasks.

Definition 3.8 Cost Function. A cost function τ is a tuple $\langle \varphi, \delta \rangle$ where φ is a cost calculation function (such as $\sum$, $\prod$, *sigmoid*, *max*, *min*, $\int$), and $\delta \subseteq (V_{CT} \cup V_P)$ is a set of active cost variables and property variables for modules. Cost function τ is computed by applying φ to the set $\{m.t \cup m.p \mid m \in V_M, t \in V_{CT}, p \in V_P, (m.t \cup m.p) \in \delta \wedge \text{active}(m.t \cup m.p)\}$.

In the cost function, cost calculation function φ is required to be monotonic, and variables $m.t \cup m.p \in \delta$ must be active and are restricted to non-negative numeric values. If the values of property variables $m.p$ involved are symbolic, some approaches; e.g. fuzzy approach, must be applied to transform symbolic values into numeric ones.

Definition 3.9 Cost Constraint. A constraint that contains a literal with a cost function is called a cost constraint.

For example:

$$Cost_{\text{resource}} = \sum_{i=1}^n CR_i \times Qt_i \quad (3.9)$$

where $Cost_{\text{resource}}$ is the total resource cost consumed; n is the number of different resources consumed; CR_i is the consumption rate of the i_{th} resource; and Qt_i is the consumption quantity of the i_{th} resource. The right side of the cost constraint is a cost function where $\sum$ is the cost calculation function φ , CR_i and $Qt_i \in \delta$.

Cost constraints represent numeric relationships among diverse cost factors, design parameters, and manufacturing process variables. In PLCC, cost factors involved in cost constraints consist not only of traditional manufacturing costs, but also costs of product operation, maintenance, disposal, recycling, and even pollution control, such as decontamination of water and air. The latter part of the costs occupies more significant proportion of PLCC and yet is most difficult to predict. Usually, cost constraints are derived from empirical relationships based on historical data and experiences, engineering relationships including economic factors, statistical relationships [141], and can also be constructed by experts when innovative technologies are applied.

Many quantitative cost estimation techniques can be used to formulate cost constraints. As a popular method, Activity-Based Costing (ABC) calculates PLCC as the sum of costs incurred on performing activities in the life cycle of a product family. The activities can be classified as a four-level hierarchy [141]: *unit-level* activities consisting of direct labor, material, machine/tooling, and energy; *batch-level* activities including setup, material handling, inventory, purchase order, and inspection; *product-sustaining* activities including process engineering, production specifications, engineering change notices, and product enhancement; and *facility-sustaining* activities involving plant management, maintenance of the buildings and grounds, heating, and lighting. Another method Parametric Cost Estimation quantifies the unit cost of products. They apply statistical methodologies, establish the scaling factors of cost factors found through analysis of historical data, and express the cost as a function of its constituent variables [134].

In fact, cost constraints can be treated as a special kind of equality and inequality constraints (defined in Section 3.3.4) that define the quantity relationship among design parameters and cost variables, which can be viewed as a special kind of property variables of modules.

Similar to activation constraints C_a and structural constraints C_s , cost constraints on modules are inherited by their corresponding instance components; that is, in

cost function τ , cost calculation function φ is applied to the set $\{c.t \cup c.p \mid c \in \text{Domain}(m) \wedge (m.t \cup m.p) \in \delta \wedge \text{active}(c.t \cup c.p)\}$.

Two categories of cost constraints are distinguished: *Cost calculation constraints* and *cost variation constraints* for different implementation purposes.

Cost calculation constraints are used to compute the cost, or state the bound of the cost of specific components or activities, even for the whole product in the configuration process. The example in Equation (3.9) is a cost calculation constraint. Another example is $\min(C_{\text{material}})$ which means that material costs should be minimum.

In the product family-based configuration, impact analysis of the change of design parameters or cost drivers on the total PLCC is an important issue. A small change in a design parameter or cost driver may incur changes in numerous cost drivers throughout the entire PLC, and eventually a huge change in the total PLCC. Such changes can be positive or negative. Cost variation constraints are created to denote the cost impact relationships among cost drivers and design parameters.

For instance, traditionally, decision on the use of a part among parts of equivalent function in product variants is primarily based on its purchasing price. This leads to a large number of part varieties in a product family. Additionally, the use of purchasing price as the decision criterion also contributes to the frequent change of parts whenever a cheaper alternative becomes available. In product family design, the cost of whole product family $PLCC_{PF}$ includes two parts: static costs and dynamic costs, where the static cost of product family $Cost_{PF}^{\text{static}}$ is defined as

$$Cost_{PF}^{\text{static}} = \sum_{i=1}^N Cost_{\text{product}_i} \quad (3.10)$$

where $Cost_{\text{product}_i}$ is the sum of costs of components used in product variant i , N is the number of product variants in the product family. Although selecting lower priced component can reduce unit static cost of a product variant, when costs arising from operational activities over the product life cycle (i.e., the dynamic cost $Cost_{PF}^{\text{dynamic}}$) are considered, the characteristics and value of $Cost_{PF}$ becomes complex

Figure 3.4: Cost impact of the change of design parameters or cost drivers on PLCC.

and different. Lower price part selection in turn results in waste on inventory, and increases operational costs in product development, manufacturing and maintenance (e.g.; transportation and purchasing), resulting in further pressure on cost reduction. However, the impact in costs by increasing part variety and frequency of part change are ignored due to the lack of scientific and effective evaluation methods.

Figure 3.4 illustrates the cost impact on several PLC stages and activities when a part is changed in a product family. At the product design stage, a part change may result in possible changes in tooling, the needs for new prototypes and tests. If the product were already launched, the part changes would also impact the product life cycle processes. At the product production stage, it may result in the depletion of affected inventory parts, the shortage of required new parts, changes in production process, and trigger new efforts in manpower training, part manufacturing or procurements. At the product use and maintenance stage, it may lead to extra part manufacturing and procurements, and the shortages or over-inventory of affected service parts in the service inventory, and hence the extra costs or wastage respectively. At the product end-of-life (EoL) stage, the part change may lead to changes in the EoL process and requirements. All these contribute directly and indirectly to the change of the PLCC of a product family. The results for a narrowed scope of stages and activities with significant contributions are summarized in Table 3.1. For all the factors in consideration, only the costs associated with man-efforts for design change tends to become lower. This is mainly because more product variants need to be

Table 3.1: Impact of part changes on PLCC (partial) [198]

<i>Stage</i>	<i>Factor</i>	<i>Level of Part Variant</i>	<i>Cost Impact</i>
Product design	Manpower on design change	Up	Down
	Tooling	Up	Up
	Prototyping	Up	Up
	Testing	Up	Up
Production	Product part making	Up	Up
	Product part procurement	Up	Up
	Product part inventory	Up	Up
	Production/assembly	Up	Up
Service	Service part procurement	Up	Up
	Service part manufacturing	Up	Up
	Service part inventory	Up	Up

attended if a part to be changed is used by more product variants in the product family. The cost arising from all other factors tends to go up along with more part varieties as more efforts are required for part making and procurements, and more likely some inventory parts are to be depleted.

Here we construct cost variation constraints to indicate the quantitative impact analysis of the part change on PLCC. As we mentioned previously, $PLCC_{PF} = Cost_{PF}^{static} + Cost_{PF}^{dynamic}$, where $Cost_{PF}^{static}$ is relatively simple and defined by Equation 3.10. $Cost_{PF}^{dynamic}$ is complex, and can be defined as

$$Cost_{PF}^{dynamic} = \sum_{j=1}^M C_j(k_j, \omega_j) \quad (3.11)$$

where M is total number of cost factors in $Cost_{PF}^{dynamic}$, C_j is the j th cost factor, k_j is the number of parts affected by the part change at cost factor C_j , and ω_j is a function of γ_j^r ; i.e., $\omega_j = f(\gamma_j^1, \gamma_j^2, \dots, \gamma_j^R)$ where γ_j^r ($r=1,2,\dots,R$) are cost coefficients of C_j .

Four cost variation constraints representing cost factor C_j ($j = 1, 2, 3, 4$) are formulated as follows [198]: the cost impact of part changes on design is expressed as:

$$C_1(k_1, \omega_1) : Cost_{design} = \sum_{i=1}^{k_1} (T_i^{manday} \times C_i^{manday-rate} + C_i^{tooling} + C_i^{opportunity}) \quad (3.12)$$

where k_1 is the number of parts affected by the replacement, T_i^{manday} is the number of man-days required to change part i , $C_i^{manday-rate}$ is the cost per man-day, $C_i^{tooling}$

is the tooling cost due to changing part i , $C_i^{opportunity}$ is the opportunity cost for time T_i^{manday} , which designers might otherwise create value.

Cost variation constraint for part changes on purchasing can be expressed as:

$$C_2(k_2, \omega_2) : Cost_{purchase} = \sum_{i=1}^{k_2} (T_i^{manday} \times C_i^{manday_rate} + C_i^{opportunity}) \quad (3.13)$$

where k_2 is the number of parts affected by the replacement in the purchasing process, T_i^{manday} is the required man-days to procure affected part i , $C_i^{manday_rate}$ is the cost per man-day, $C_i^{opportunity}$ is the opportunity cost for time T_i^{manday} , which purchasers might otherwise create value.

Cost variation constraint on the cost/wastage of inventory is represented as:

$$C_3(k_3, \omega_3) : Cost_{inventory} = \sum_{i=1}^{k_3} \rho_i \times \theta_i \times \alpha_i \times Cost_{cm_i} \quad (3.14)$$

where $Cost_{cm_i}$ is the cost of affected component cm_i ; ρ_i is the life cycle time of cm_i ; $\alpha_i \in [0, 1]$ is a probability coefficient for likely depletion of cm_i ; and

$$\theta_i = f\{T^{procurement}(cm_i), T^{production}(cm_i), T^{delivery}(cm_i), Cost_{cm_i}\}$$

is the minimum inventory required to meet the order-to-delivery lead-time $T^{delivery}(cm_i)$. The factors that also play a major role in the determination of θ_i include the procurement lead-time $T^{procurement}(cm_i)$, the production/assembly lead-time $T^{production}(cm_i)$, and the cost of component cm_i $Cost_{cm_i}$.

Another example of cost variation constraints in regard to the cost of service part inventory of components cm_i is represented as:

$$C_4(k_4, \omega_4) : Cost_{serv-inv} = \sum_{i=1}^{k_4} \rho_i \times \mu_i \times \alpha_i \times Cost_{cm_i} \quad (3.15)$$

where ρ_i , α_i , $Cost_{cm_i}$ have the same meanings as in Equation 3.14, and μ_i , the minimum number of service parts of cm_i to be prepared in inventory, is determined by

$$\mu_i = f\{R^{failure}(cm_i), S^{sale}(cm_i), Cost^{mass}(cm_i), Cost^{small}(cm_i)\}$$

where $R^{failure}(cm_i)$ is the part-failure rate; $S^{sale}(cm_i)$ is the projected-units of sales, $Cost^{mass}(cm_i)$ is the mass-procurement cost; and $Cost^{small}(cm_i)$ is the small-batch procurement cost.

These two categories of cost constraints; i.e., cost calculation constraint and cost variation constraint, have different implementations in our DCSP model solving process. The implementations will be proposed in the next section.

3.3.6 Preference Constraints

Most constraints in a CSP can be recognized as *absolute constraints* that are used to prune potential solutions of a CSP. On the contrary, *preference constraints* are used to indicate which values or solutions in a CSP are preferred [199]. In a CSP, preference constraints have lower priority than absolute constraints in that absolute constraints must be satisfied while preference constraints may be sacrificed if they conflict with absolute constraints. We introduce three kinds of preference constraints.

- **Preference on value** sets the preferred value of a variable in its domain. It can be represented as:

$$CP : var_i \mid operator \mid value_j \quad (3.16)$$

where $var_i \in V_M \cup V_{CM} \cup V_P \cup V_{CT}$ in the configuration domain Φ ; $value_j$ refers to the value that var_i is preferred to be taken; *operator* specifies the relationship between var_i and $value_j$. It can be $=$, $\neq$, $>$, $<$, $\leq$, $\geq$, $\in$, *etc.*

For example, $Mouse.Color \neq Sliver$ or $Mouse.Color \in \{Silver, Red\}$.

- **Preference on solution** specifies which solutions of a CSP are preferred; e.g., the cost should be lowest. In fact this kind of preference constraints can be treated as objective functions in a CSP and be solved as an optimization problem of CSP. The representation of preference on solution is

$$Constraint \ Description \mid V_s \quad (3.17)$$

where V_s involves all the variables to which the preference is applied. It can be a list of variables or a description on the variables. *Constraint Description* states the preference on V_s .

For example, the preference “Computer weight should be lightest” is expressed as $\text{Min}(\text{Weight}_{\text{all}}) \mid \{\text{Weight}_{\text{Monitor}}, \text{Weight}_{\text{Host}}, \text{Weight}_{\text{Accessory}}\}$ or $\text{Min}(\text{Weight}_{\text{all}}) \mid \{\text{the weight of all constitutions of a computer}\}$.

- **Preference on constraint** is used to set the condition for a constraint to be applied. It is represented as:

$$\text{Conditional Constraint} \rightarrow \text{Sequential Constraint} \quad (3.18)$$

where both sides of the constraint can be variable assignment or any type of defined constraints in the configuration domain Φ . The conditional constraint sets a specified condition that triggers the sequential constraint.

For instance, the preference “If a computer is used for business, the total memory capacity should be at least 2G” is expressed as: $(\{\text{Usage} = \text{Business}\}) \rightarrow (\text{Memory.TotalCapac} \geq 2G)$.

3.4 DCSP-Based Configuration Problems

Based on the definition above, a product configuration problem can be described as a DCSP that involves a configuration domain referring to a knowledge base comprising various variables and constraints governing them, a set of initial variables with corresponding domains, and a set of constraints related to initial variables.

Definition 3.10 DCSP-Based Configuration Problem. *A DCSP-Based Configuration Problem is a quaternion $\langle V^I, D^I, C^I, \Phi \rangle$, where $\Phi = \langle V_M, V_{CM}, V_P, V_{CT}, D, C_a, C_s, C_g \rangle$ is a configuration domain, $V^I \subseteq V_M \cup V_{CM} \cup V_P \cup V_{CT}$ is the set of initial variables, $D^I \subseteq D$ is the corresponding domains of V^I , and $C^I \subseteq C_a \cup C_s \cup C_g$ is the set of constraints on V^I . A solution to the DCSP-based configuration problem is an*

assignment to a set of activated variables V^a ($V^a \subseteq V_M \cup V_{CM} \cup V_P \cup V_{CT}$, $V^I \subseteq V^a$), such that all active constraints on V^a are satisfied.

In this definition, initial variables in V^I with corresponding domains D^I and the constraints in C^I compose a partial constraint network as the initial status of a DCSP-based product configuration task, and then augment dynamically to achieve the final solution of the configuration problem.

Finding solutions S to a configuration problem $\langle V^I, D^I, C^I, \Phi \rangle$ refers to a solving process dynamically constructing a constraint network N starting from an initial network $\langle V^I, D^I, C^I \rangle$. The solving process can be viewed as a sequence of stable phases in each of which N represents a fixed classical constraint network where only active constraints on the active variables are propagated until a consistent state is reached. Constraint network N extends for the following reasons:

- When module m is active in N , all its direct common child modules $(m_1, \dots, m_n)$, which are not existing in N , are activated into N as required by structural constraints C_s of the form $m \Downarrow \{m_1, \dots, m_n, \sim m_{n+1}, \dots, \sim m_t\}$. Note that depending on customer needs, required optional child modules; e.g., m_t , are also added to N .
- When module m is active in N , all its property and cost variables are activated into N by activation constraints C_a of the form $m \implies m.p$ and $m \implies m.t$.
- When module m is selected for an assignment, an instance component $c \in \text{Domain}(m)$ is introduced into N as an active variable. Moreover, c inherits all property and cost variables of m . The names of the property and cost variables are changed to $c.p$ and $c.t$ respectively.
- Furthermore, constraints whose variables are all the active variables are added to N as active constraints.

3.5 Mapping Relationship between PDMs and DCSP-Based Configuration System

As mentioned before, a huge amount of data with great variety and complexity must be represented in product configuration tasks. In product configuration tasks, besides traditional production data (such as resource data, assembly/fabrication data, technique data, product features, transportation data, and inventory data), requirement specification, cost data, background knowledge for product configuration and cost estimation, and other PLM data (e.g., disposal and environment protection) must also be captured in the problem representation. The semantics and structure of these data are traditionally stored in PDMs. In a DCSP-based configuration system, all data are modeled by variables, domains, and constraints. Thus the formalisms of DCSP need to be derived from existing PDMs mastered by manufacturers. This process is illustrated in Figure 3.1.

In this section, some PDMs representing and managing the data involved in product configuration are described. The mapping relationships between these PDMs and the extended DCSP-based configuration model are also analyzed to facilitate the construction of the DCSP system.

3.5.1 PDM for Requirement

To satisfy target customers' needs and desires, manufacturers tailor product features within a product family. Hence, modeling requirements is fundamental for product configuration tasks. However, customer requirements are usually vague. In another words, customers do not exactly know what they really want. Manufacturers are required to extract customer's requirements from various resources, involving direct requirements, surveys, customer transaction records, customer profiles, and customer complaints or feedbacks, etc. [144], and then generate requirement specifications in technical terms. This process is complex and beyond the scope of this study. Generally speaking, customer requirements can be classified as *function requirement*, *restriction*

Figure 3.5: A data model of the requirement.

requirement, and *wish requirement* [104]. “Function requirement” states the required functionality for a product and focuses on the behavioral features, such as a mouse is required for a computer. “Restriction requirement” is defined on structural features, such as the color of a computer is red. “Wish requirement” refers to customer’s desires. It can be defined on either behavioral or structural features, and can be related on a constituent of a product at any level, such as a part, a subsystem, or the entire product (e.g., black color is not preferred, or computer’s weight should be lightest). However wish requirements have relatively lower priority than other two kinds of requirements. When a wish requirement conflicts with other mandatory requirements or limitations in PLC, such as functionality, processing, safety, or environment protection, it will be sacrificed. Among customer requirements, function and restriction requirements must be fulfilled while wish requirements may not be satisfied.

A PDM of requirements represented in UML is shown in Figure 3.5 where the multiplicity indicates that in customer requirements, there must be at least one function requirement, while there may be no restriction requirement or no wish requirement.

3.5.2 PDM for Product Configuration

A product can be recognized as an assembly comprising a set of sub-systems or components based on function decomposition at varying levels with certain structures. In product family, such structure is represented by a PFA that can be treated as an *assembly view* of product configuration [104]. There are two threads in a PFA: a *product family* and a set of *product variants*, as illustrated in Figure 3.6.

Figure 3.6: Product family architecture of desktop computer.

A product family is composed of *modules*. Corresponding to PFA, modules can be classified as primitive modules (which are leaf elements in a PFA) and compound modules (which include at least two child modules). Responding to the commonality and variety of product family design, modules can also be categorized into common and differentiation modules. During product generation, while common modules are mandatorily shared by all product variants, differentiation modules are the main source of variety and are optional. For example, in Figure 3.6, compound module “Host” includes three primitive modules: “CPU”, “Memory” (which are common modules) and “Floppy Disk” (which is a differentiation module). Each module possesses a set of attributes.

Product variants are generated as instantiations of the corresponding product family by selecting suitable modules and component alternatives, and assigning values to design parameters. A product variant consists of *components*, each of which is an instance of the corresponding module in the product family, and hence inherits all attributes of the module. In Figure 3.6, “product variant A” consists of six components corresponding to common modules in product family “Desktop PC”, and does not involve differentiation module “Floppy Disk”. Also, the structure of components in a product variant inherits that of modules in the corresponding product family. When a product configuration is completed, component attributes in a product variant are

Figure 3.7: Assembly view and cost view of product configuration.

filled with values so that the product variant matches specific requirements.

From another perspective, a product can be viewed as a combination of cost drivers that refer to major factors influencing the PLCC of a product. Such a view is recognized as the *cost view* of product configuration. Cost drivers directly or indirectly rely on various product information such as resource, functionality, fabrication/assembly processes, logistics, and other PLM activities. Generally, cost drivers are linked to certain activities associated with modules and can be treated as a type of attributes.

Figure 3.7 illustrates a combination of assembly and cost view of product configuration. The multiplicity between product variant and subsystem/component indicates that a product variant may not include any subsystem/component as its direct child. However, a product variant must involve at least one subsystem or component. A subsystem involves at least two components. Note that the link connecting subsystem to itself indicates that the subsystem may be decomposed into more than one levels. Each subsystem, component, and the product directly possesses at least one cost driver.

Based on the analysis above, a module is denoted as

$$Module : (\overrightarrow{IC}; \overrightarrow{A}; \overrightarrow{C}; \overrightarrow{CM}) \quad (3.19)$$

where $\overrightarrow{IC}$ refers to the vector of all instance components of the module; $\overrightarrow{A}$ refers to the attribute list of the module; $\overrightarrow{C}$ is the cost driver list of the module; $\overrightarrow{CM}$ is the vector of child modules of the module. Note that $\overrightarrow{A}$ and $\overrightarrow{C}$ of a module are all inherited by its instance components.

3.5.3 PDM for Design Knowledge

The core of product configuration is selecting and combining components based on predefined rules for component composition [180]. Design knowledge is used to define the predefined rules, and is in the form of relationships and constraints. It provides the background information, cooperating with configuration data and given requirements, for designers to tailor a valid product configuration. It includes the following information:

- (i) Function decomposition; i.e., module decision;
- (ii) Module classification; i.e., common or differentiation module;
- (iii) Connection between modules that can be used to determine the PFA;
- (iv) Requirement-module correspondence relationships that are used to retrieve module selection for new similar requirements, which is called design process reuse [104].
- (v) Regulations of component combinations and restrictions among attributes' values. These regulations can either be between two modules or components, or between two attributes' values of the same or different modules/components, or among more than two modules/components/attributes; i.e., n-ary constraints ($n > 2$). Note that a n-ary constraint is not always equivalent to two parallel binary constraints. For example, rule $\langle \text{Color}=\text{Red}, \text{Power}=220\text{V}, \text{Shape}=\text{square} \rangle$ is not equivalent to $\langle \text{Color}=\text{Red}, \text{Power}=220\text{V} \rangle$ and $\langle \text{Power}=220\text{V}, \text{Shape}=\text{square} \rangle$ as the relationship between “Color=Red” and “Shape = square” is ignored; this makes sense only when each of them is related to “Power = 220V”.
- (vi) Information on identification of design parameters, cost drivers, domains of them, and interrelationship among them.

Figure 3.8: Mapping relationships between PDM and DCSP.

3.5.4 Mapping Relationship

As proposed above, the concepts in DCSP-based configuration system can be directly derived from the data in PDMs. The mapping relationship between PDMs and DCSP, as illustrated in Figure 3.8, can be described as follows:

- Modules in a product family identified from PDM for configuration or PDM for design knowledge correspond to module variables $m \in V_M$ in DCSP model.
- In PDMs, instance component vector $\vec{IC}$ of a module corresponds to the domain of the corresponding module variable in DCSP; i.e., $Domain(m|m \in V_M)$. All elements in $\vec{IC}$ that are components in product variants map to component variables $c \in V_{CM}$ in DCSP.
- The elements in attribute vector $\vec{A}$, cost driver vector $\vec{C}$ of modules and their corresponding domains in PDMs correspond to property variables $p \in V_P$, cost

variables $t \in V_{CT}$, and domains in DCSP respectively.

- Activation constraints C_a in DCSP can be derived from $\vec{A}$ and $\vec{C}$; that is, when a module variable is active, all its related property and cost variables corresponding to the elements in $\vec{A}$ and $\vec{C}$ are also activated into DCSP.
- Child modules vector $\vec{CM}$ of a module in PDMs derives structural constraints C_s in DCSP.
- Product features required in requirement specifications convert to property variables $p \in V_P$ in DCSP model.
- Function and restriction requirements in PDMs map to equality and inequality constraints that directly restrict the value of property variables $p \in V_P$ in DCSP. Based on specific value settings, some preference on constraint in DCSP are generated.
- Wish requirements in PDMs maps to constraints on preference on value, or preference on solution in DCSP, all of which have relatively lower priority.
- From the cost view of PFA, cost constraints in DCSP that represent the relationships among cost drivers and design parameters can be derived.
- In PDMs for design knowledge, the classification of modules and connection among them can be used to derive structural constraints C_s in DCSP; i.e., common or differential child modules. The restriction on combinations of components and attribute values derive compatibility and incompatibility constraints in DCSP. Mathematical relationship among attribute and cost drivers convert to equality, inequality constraints and cost constraints in DCSP. According to specific variable settings in PDMs, the preference on constraint in DCSP is derived. In addition, some n-ary constraints convert to preference on constraint where each side only includes unary or binary constraints. For instance, $\langle \text{Color}=\text{Red}, \text{Power}=220\text{V}, \text{Shape}=\text{square} \rangle$ converts to $(\text{Color}=\text{Red}, \text{Power}=220\text{V}) \rightarrow (\text{Shape}=\text{square})$.

There is information loss in data conversion from PDMs to DCSP. Data representation in DCSP is at the symbol level; that is, variables and constraints. On the contrary, PDMs have semantic-rich expressiveness and provide background knowledge for DCSP [104]. For example, in the DCSP based on Figure 3.6, from constraint “ $MF_1.color = M_1.color$ ”, we only know that the property “color” of variables “ M_1 ” and “ MF_1 ” should be the same. However, for non-experts, it does not make any sense as it involves no semantics. When done in consideration of PDMs, we understand that it means that the color of monitor and host should be the same. Moreover, the background knowledge contained in PDMs provides the possibility of special solving and heuristic approaches for specific configuration tasks to improve the solving efficiency in DCSP. For example, variable ordering is based on the property importance of a module, which is provided by PDMs.

3.6 Constraint Acquisition

As previously mentioned, a huge amount and variety of data are mapped to formalisms in DCSP-based product configuration and cost estimation model. As an important engine of the DCSP solving process, constraints in the DCSP-based model are also derived from the data and imposed on those data inversely. Hence, efficient acquisition of constraints is a concern. Appropriate machine learning techniques can be applied to mine rules from a large set of complex and information-rich product configuration data. Constraint discovery targets mining possible relationships among components and attributes by evaluating the probability of co-existence in a configuration. In this section, we propose a constraint acquisition approach using the concept of association analysis. By applying the proposed approach, partial constraints are expected to be automatically generated from historical product configuration data.

3.6.1 Association Rule Mining Approach

Association rule mining [185] is a technique to discover interesting relationships (i.e., association rules) hidden in large data sets. Generally, the data set from which asso-

ciation rules are mined includes a large quantity of transactions each of which consists of several items. A set of zero and more items is termed as an itemset, and a k -itemset refers to an itemset including k items. An association rule can be viewed as a probabilistic statement to infer the co-occurrence of certain items in data sets, taking the form of $X \rightarrow Y$ where X and Y are disjoint itemsets; i.e., $X \cap Y = \emptyset$ [185, 211]. For instance,

Given itemsets : $A \{A_1, A_2, A_3\}, B \{B_1, B_2\}$

$(A = A_1) \rightarrow (B = B_2)$ with probability p

There are two issues in association rule mining [185]: First, mining association rules from a large transaction data set can be computationally expensive. Second, some mined rules may be spurious as they happen by chance. In association rule mining, *support* and *confidence* are two common indicators to assess the strength of rules and thus, address the above issues. *Support* is defined as $s(X \rightarrow Y) = \frac{|X \cup Y|}{N}$ where N is the total number of transactions in a data set, $|X \cup Y|$ is the support count of itemset $\{X, Y\}$ referring to the number of transactions containing itemset $\{X, Y\}$ in the data set. While *support* determines how often a rule is applicable to a given data set, and is often applied to eliminate uninteresting rules as a rule with low *support* may occur by chance, *confidence* $c(X \rightarrow Y) = \frac{|X \cup Y|}{|X|}$, where $|X|$ refers to the number of transactions containing itemset $\{X\}$ in the data set, determines how frequently item Y appears in transactions containing antecedent item X , and is used to measure the reliability of the inference made by an association rule [185]—the higher $c(X \rightarrow Y)$ is, the more likely it is for Y to be present in transactions involving X . Only rules with *support* and *confidence* that are not less than predefined thresholds (i.e., *minsup* and *minconf* respectively) are considered as interested rules.

Generally, a strategy adopted by association rule mining algorithm is to decompose the mining process into two major steps [185]:

- Frequent Itemset Generation: to find all itemsets with *support* $\geq$ *minsup*; these are called frequent itemsets.

- Rule Generation: to extract rules with high *confidence* (i.e., $confidence \geq minconf$) from frequent itemsets found in the previous step.

3.6.2 Constraint Acquisition

In our experiments, we apply association rule mining to acquire compatibility and incompatibility constraints in product configuration. A historical product configuration consisting of a set of components is treated as one transaction in association rule mining. Each component in a product configuration is treated as an item in a transaction. Based on the two steps of association rule mining theory described in the previous section, compatibility constraints $X \rightarrow Y$ can be derived from historical configuration transactions. Brute-force approach is a basic approach for association rule mining; it computes support and confidence of every possible rule, and is prohibitively computationally expensive [185]. To address this issue, the *Apriori* algorithm is adopted in both two steps of association rule mining.

For instance, Table 3.2 shows a small set of transactions including partial items in regard to a desktop computer configuration system. In this table, each row corresponds to a transaction (i.e., a historical computer configuration) in association rule mining data sets, each column represents a module in the computer family, and the value of each module in a transaction is a selected instance component of the corresponding module in a computer configuration; it refers to an item in a transaction. If the value of an item is 0, the corresponding module of the item represents an optional one, and is not selected in a product configuration (e.g., in Table 3.2, the value of item “Graphic Card” in transaction 5 is 0, which means that in the 5th computer configuration, optional module “Graphic Card” is not selected). In Table 3.2, there are 8 transactions and 27 items.

In frequent itemset generation, the *Apriori* algorithm is a support-based strategy whose principle is that if an itemset is frequent, then all of its subsets must also be frequent. Conversely, if an itemset is infrequent, then all of its supersets

Table 3.2: Transactions (With Partial Items) for Association Rule Mining

Tran	Monitor	Chassis	Mother-board	CPU	Memory	Mouse	Key-board	Graphic Card
1	cM_1	cCH_1	cMB_5	$cCPU_8$	cMM_7	cMS_9	cKB_1	cGC_1
2	cM_8	cCH_5	cMB_2	$cCPU_4$	cMM_7	cMS_2	cKB_5	cGC_1
3	cM_2	cCH_7	cMB_2	$cCPU_1$	cMM_2	cMS_9	cKB_1	cGC_1
4	cM_8	cCH_5	cMB_2	$cCPU_4$	cMM_2	cMS_2	cKB_5	cGC_1
5	cM_8	cCH_5	cMB_3	$cCPU_2$	cMM_6	cMS_2	cKB_5	0
6	cM_5	cCH_2	cMB_3	$cCPU_2$	cMM_6	cMS_3	cKB_1	cGC_2
7	cM_2	cCH_5	cMB_2	$cCPU_1$	cMM_7	cMS_2	cKB_5	cGC_1
8	cM_8	cCH_3	cMB_2	$cCPU_1$	cMM_7	cMS_2	cKB_5	cGC_2

must also be infrequent [185]. Using this property, the *Apriori* algorithm is able to prune the candidate itemset space and control the exponential growth of candidate itemsets; a critical issue in the brute-force approach. Specifically, provided that the threshold *minsup* is 50%, in Table 3.2, the support count of a frequent itemset must be at least 4. Thus, 1-itemsets, for example, $\{cGC_2\}$ and $\{cM_2\}$, both of which have support count of 2, are infrequent and are pruned from the candidate frequent itemset space. Any 2-itemset containing these two infrequent 1-itemsets, such as $\{cMB_2, cGC_2\}$, $\{cM_2, cCH_7\}$, are also pruned from the candidate space. Based on this principle, in Table 3.2 only 7 1-itemsets out of 27 items are frequent; i.e., support $\geq 50\%$. From these 7 frequent 1-itemsets, candidate frequent 2-itemsets can be generated, among which only 9 2-itemsets out of 21 candidates are frequent. Similarly, 2 frequent 3-itemsets out of 84 candidates can be generated. The total number of candidate itemsets generated by *Apriori* algorithm is $\binom{27}{1} + \binom{7}{2} + \binom{9}{3} = 27 + 21 + 84 = 132$. Compared with the number of candidates generated in brute-force approach; that is, $\binom{27}{1} + \binom{27}{2} + \binom{27}{3} = 27 + 351 + 2925 = 3303$, *Apriori* algorithm presents a 96% reduction in the number of candidate itemsets in frequent itemset generation process, even in a simple case, and thus, significantly improves the solving effectiveness.

Subsequently, inference rules are further generated from the derived frequent itemsets by computing the confidence of frequent itemsets. In rule generation, the *Apriori* algorithm performs confidence-based pruning, whose principle is that if a

Figure 3.9: Pruning association rules using the confidence measure.

rule $X \rightarrow Y - X$ does not satisfy the confidence threshold $minconf$, then any rule $X' \rightarrow Y - X'$, where X' is a subset of X , must not satisfy $minconf$ too [185]. Based on this principle, if any rule generated from frequent itemsets has low confidence, then all the rules generated from it are also pruned. For example, in the case of Table 3.2, based on the derived frequent 3-itemset $\{cMS_2, cKB_5, cCH_5\}$, rules that can be generated are illustrated in Figure 3.9. Given confidence threshold $minconf$ is 90%, the confidence of rule $\{cMS_2, cKB_5\} \rightarrow \{cCH_5\}$ is $80\% < minconf$. This rule and all the rules involving item cCH_5 in its consequent are all pruned, confined by dotted lines. Confidence also represents an estimate of the conditional probability of Y given X [185]. For example, association rule $\{cMS_2, cCH_5\} \rightarrow \{cKB_5\}$ is generated with probability (i.e., confidence) of 100%.

We are also interested in incompatibility constraints in the form of $X \rightarrow !Y$ among components, as they are stricter in terms of removing inconsistent components from modules' domains, and are thus, more effective in search space pruning. Association rule mining is also applied to mine incompatibility constraints. Figure 3.10 illustrates the mining process. In the first step, the methodology used to generate frequent itemsets is the same as the one in compatibility constraints mining process described above; that is, finding all frequent itemsets with $support \geq minsup$. However, in the rule generation step, the method applied is different from the one in compatibility constraints mining. While confidence (i.e., $p(X \rightarrow Y)$) is calculated in the compatibility

Figure 3.10: Mining process for compatibility and incompatibility constraint.

constraint mining process, in the incompatibility constraint mining, inverse confidence $p(X \rightarrow !Y)$ that equals to $1 - p(X \rightarrow Y)$ is measured. Rules contracted from frequent itemsets with low inverse confidence; i.e., $p(X \rightarrow !Y) \leq \text{mininvconf}$, will be considered as incompatibility constraints.

As constraints obtained by association rule mining are probabilistic implications, verification by designers is needed after the constraint discovery process.

3.6.3 Experiment Result

Experiments are performed on a set of data transactions consisting of 67 pieces of historical computer configurations to generate compatibility constraints $X \rightarrow Y$. The method presented in the previous section upon the data transactions is implemented in open source data mining software Weka, with the following settings:

- Class used: weka.associations.Apriori
- LowerBoundMinSupport: 0.1 with 0.05 increments

Table 3.3: Compatibility Constraints Generated from Experiment

No.	Compatibility Constraints	Support	Confidence
1	Keyboard= $cKB_1 \rightarrow$ Mouse= cMS_9	44.8%	100%
2	Mouse= $cMS_9 \rightarrow$ Keyboard= cKB_1	44.8%	93.8%
3	Mouse= $cMS_2 \rightarrow$ Keyboard= cKB_5	37.3%	92.6%
4	CPU= $cCPU_3 \rightarrow$ Motherboard= cMB_3	31.3%	100%
5	Motherboard= $cMB_3 \rightarrow$ Independent graphic card= cGC_2	31.3%	100%
6	CPU= $cCPU_3 \rightarrow$ Independent graphic card= cGC_2	31.3%	100%
7	Motherboard= $cMB_3 \rightarrow$ Memory= cMM_6	29.9%	95.2%
8	Motherboard= $cMB_2 \rightarrow$ Independent graphic card= cGC_1	28.4%	90.5%
9	CPU= $cCPU_6 \rightarrow$ Motherboard= cMB_2	26.9%	90%
10	CPU= $cCPU_6 \rightarrow$ Independent graphic card= cGC_1	26.9%	90%
11	Motherboard= $cMB_4 \rightarrow$ Chassis= cCH_4	17.9%	100%
12	Chassis= $cCH_3 \rightarrow$ Motherboard= cMB_5	16.4%	100%

Table 3.4: Incompatibility Constraints Generated from Experiment

No.	Incompatibility Constraints	Support	Inverse Confidence
1	CPU= $cCPU_9 \rightarrow !$ Motherboard= cMB_5	6.0%	25%
2	Motherboard= $cMB_5 \rightarrow !$ Independent graphic card= cGC_3	6.0%	22.2%
3	Motherboard= $cMB_1 \rightarrow !$ Memory= cMM_7	7.5%	27.8%
4	Floppy= $cF_1 \rightarrow !$ Chassis= cCH_1	7.5%	23.8%

- Minimum confidence: 0.9

As in the extended DCSP-based product configuration and cost estimation system proposed in this chapter, only binary constraints are considered. As such, in the association rule mining, we also only consider frequent 2-itemsets that generate binary association rules. From this experiment, 50% of the predefined compatibility constraints are discovered, among which the top 12 compatibility constraints are listed in Table 3.3. The accuracy is considered promising given the data size.

Experiments on mining incompatibility constraints by *Apriori* algorithm are also implemented. Experimental results are listed in Table 3.4 with the following settings:

- LowerBoundMinSupport: 0.05 with 0.02 increments
- Minimum confidence: 0.7, then minimum inverse confidence is 0.3.

3.7 Summary

The process of CSP-based configuration and cost estimation tasks consists of three constituents: PDMs, constraint model, and constraint solver. In this chapter, an extended DCSP-based product configuration and cost estimation model is presented in detail. This model extends the classical CSP by dynamically creating new active variables, captures the hierarchical nature of modules and components in a product configuration, and integrates cost estimation into product configuration tightly. In this model, all the concepts in product configuration and cost estimation tasks; e.g., modules, components, properties, and cost drivers, are specified as variables with associated domains of the extended DCSP-based model. Various constraints are also identified. Their impacts on the solving process of the extended DCSP-based model are discussed briefly. Using activation constraints, the extended DCSP-based scheme is therefore suitable for modeling problems whose variables involved in final solutions are not *a priori*. Structural constraints not only directly capture the hierarchy information of product family, but also improve the solving efficiency by a level-wise search. Cost constraints make cost a coherent criterion as functionality and requirements in product family design. And the extended DCSP-based system is extended dynamically by the activation of variables that can be triggered by structural constraints, activation constraints, and cost variation constraints.

However, the formalisms of the extended DCSP-based system, namely variables, domains and constraints, are derived from PDMs that store the semantics and structure of data for product configuration and cost estimation. To facilitate the derivation, mapping relationships between PDMs and the proposed DCSP-based configuration and cost estimation system are analyzed.

An automatical constraint generation method using the association rule mining approach is applied to discover compatibility and incompatibility constraints in the extended DCSP-based system that represent the co-existence relationship among components.

Chapter 4

Augmented Backtracking for DCSP-Based Product Configuration with Cost Estimation

After the extended DCSP-based product configuration and cost estimation model is formalized in Chapter 3, the extended DCSP-based system can be solved for possible configuration solutions with cost estimation in the constraint solver, as shown in the flowchart of Figure 3.1.

In this chapter, an augmented Backtracking solving process is proposed for the extended DCSP-based configuration and cost estimation model proposed in Chapter 3. Using Structural constraints and activation constraints, this extended backtracking process constructs a constraint network for the extended DCSP-based system incrementally. Following the internal hierarchy of a product family, only active variables with related active constraints whose variables are all active are introduced into the constraint network and are eventually assigned compatibility values in the final solution to satisfy specific customer requirements. The implementation of two types of cost constraints is also discussed. Moreover, the limitations and complexity of the proposed algorithms are discussed. Finally, the proposed algorithms are implemented on product configuration and cost estimation tasks; experimental results are discussed.

This chapter is organized as follows: In Section 4.1, an augmented Backtracking solving process for the extended DCSP-based system is proposed. Section 4.2 presents

the implementation of two categories of cost constraint in the augmented Backtracking solving process. Section 4.3 elaborates experiments implementing the extended DCSP-based system and the augmented Backtracking algorithms on a desktop personal computer configuration and cost estimation task. Finally, conclusion is drawn in Section 4.4.

4.1 Augmented Backtracking for DCSP-Based Configuration problem

CSP is generally solved via search paradigms. Specific techniques include generate-and-test paradigm (GT), backtracking, backjumping, consistency checking, constraint propagation, and forward checking [13, 99]. Among the diverse CSP solving algorithms, the backtracking (BT) paradigm [199] is most popular. Backtracking extends a partial assignment to a complete solution. Variables are instantiated sequentially, and the consistency of a constraint is checked once all its variables are instantiated. When a partial instantiation of variables violates a constraint, BT is performed backwards to re-evaluate the most recently instantiated variable with available unassigned values, thereby pruning a subspace from the entire search space; i.e., the Cartesian product of all variable domains.

In this section, we describe an augmented BT search process, as sketched in Figure 4.1 for our extended DCSP-based product configuration and cost estimation system proposed in Chapter 3 where structural constraint handler, activation constraint handler and cost variation constraint handler are embedded into a classical BT procedure for discrete DCSPs.

Generally, the structure of a CSP can be viewed as a *constraint network* where variables, domains, and constraints are represented as nodes, labels of nodes, and edges respectively. The BT searching process that we design for the extended DCSP-based configuration and cost estimation problem $\langle V^I, D^I, C^I, \Phi \rangle$ dynamically constructs a constraint network N starting from an initial network $\langle V^I, D^I, C^I \rangle$. The BT solving

Figure 4.1: Flowchart for BT algorithm on DCSP-based configuration.

process can be treated as a sequence of stable phases in which N refers to a fixed classical CSP consisting only of current active variables and associated activated constraints. A new phase is generated by the activation of variables with the subsequent introduction of new active constraints. The extension of a constraint network N can be triggered by the following reasons:

- (i) When module m is active in N , all its direct common child modules $(m_1, \dots, m_n)$, which do not exist in N , are activated into N as required by structural constraints C_s : $m \Downarrow \{m_1, \dots, m_n, \sim m_{n+1}, \dots, \sim m_t\}$. Note that depending on customer needs, the required optional child modules such as m_t are also added to N .
- (ii) When module m is active in N , all its property variables $m.p$ and cost variables $m.t$ are activated by activation constraints C_a of the form $m \implies m.p$ and $m \implies m.t$, and introduced into N .
- (iii) When module m is active in N , its instance component variables $c \in \text{Domain}(m)$

are introduced into N as active variables. Moreover, c inherits all property and cost variables of m . The names of the property and cost variables are changed to $c.p$ and $c.t$ respectively.

- (iv) Furthermore, constraints whose variables are all active variables are added to N as active constraints.

The main loop of the BT process in Figure 4.1 is described as follows:

- The solving process starts from an initial constraint network $N = \langle V^I, D^I, C^I \rangle$.
- Step 1: In the structural constraint C_s handling phase, when new module variables are to be activated (i.e., all active variables in DCSP are instantiated), all direct common child module variables of a module variable m are activated and added into N . According to specific requirements, some or all optional child modules of m are also added to the active variable set of N . Component variables in the domain of each new active module variable are also added to N .
- Step 2: Using activation constraints C_a , an appropriate set of property and cost variables associated with the activated module variables in Step 1 are activated into N , and are inherited by corresponding component variables. Subsequently, constraints (except cost variation constraints) whose variables are all activated are also introduced into N . By now, a new fixed classical constraint network (i.e., a new stable phase) is generated.
- Step 3: An unassigned active variable is chosen as current variable cv for checking.
- Step 4: Assuming an unassigned active variable exists, constraint propagation based on BT algorithm is performed on N . The consistency of constraint is checked between cv and instantiated variables. Then inconsistent values that violate any active constraint in N are temporally removed from cv 's domain,

such that all values in the remaining domain of cv are locally consistent with the assigned variables. Note that in the constraint propagating process, only the activated generic constraint C_g (such as compatibility constraints, equality constraints, cost calculation constraints and preference constraint) is checked for consistency. That is, the consistency of activation constraints C_a and structural constraints C_s are not checked again in the propagating process.

- Step 5: If cv 's domain becomes empty, BT is performed. In the case when BT is performed backwards to each previously assigned variable and a consistent partial assignment is still not found, there is no solution to the DCSP.
- Step 6: Otherwise, a value is assigned to cv from its remaining domain, which is consistent to the partial solution including all assigned variables. The assignment appends to the current partial solution.
- Steps 3-6, representing a classical BT searching procedure, iterate to assign a value to each active variable in N .
- Step 7: In Step 3, if no unassigned active variable exists (i.e., a partial consistent assignment to all active variables V^a is reached), the process returns to Step 1. New child module variables for an existing module variable in N , together with a set of corresponding property and cost variables, and associated active constraints, are introduced into N . New searching phase is formed and the solving process described above repeats.
- Step 8: If no new child module variable exists, the solving process terminates. The current consistent partial solution is just a solution to the DCSP.

The augmented BT solving process described above is also presented as Algorithm 4.1 to Algorithm 4.4. Generally speaking, the augmented BT process with our extended DCSP-base configuration and cost estimation model refers to a sequence of stable phases. A new phase is started by the creation of new active variables with

Algorithm 4.1: Augmented BT Solving Process on DCSP-based System

Input: Initial DCSP-based configuration task $CP = \langle V^I, D^I, C^I, \Phi \rangle$;

Output: configuration solution sol and cost estimation result $ctSol$ to CP .

```

1:  $V^a = V^I$ ;  $D^a = D^I$ ;  $C^a = C^I$ ;  $sol = \{\}$ ;  $ctSol = 0$ ;
2: do
3:    $newV^a = \{\}$ ;
4:    $C_s = \{c \mid c \in C^a, c \text{ is a structural constraint}\}$ 
5:   procedure ActivateModuleVar  $\{V^a, C_s, newV^a, D^a\}$ 
6:   if  $newV^a == \emptyset$  //no new child module exists, process terminates
7:     output  $sol, ctSol$ ;
8:    $C_{act} = \{c \mid c \in C^a, c \text{ is a activation constraint}\}$ 
9:   procedure ActivatePropertyCostVar( $newV^a, C_{act}, D^a$ );
10:   $V^a = V^a \cup newV^a$ ;
11:  procedure ActivateConstraint( $V^a, C^a$ ); //Activate related constraints.
    //By now, a new phase of  $N$  is built.
12:  procedure ConstraintPropagateBT( $V^a, D^a, C^a, sol, ctSol$ );
13:  if  $sol == \{\}$  //after backtracking there is no consistent partial solution
14:    output  $sol, ctSol$ ;
15:  while  $newV^a \neq \emptyset$  //while new child modules exist
16:  output  $sol, ctSol$ ;
```

Algorithm 4.2: Procedure ConstraintPropagateBT($V^a, D^a, C^a, partialSol, costResult$):
General constraint propagation process using backtracking.

```

1: while unassigned variable  $curV \in V^a$  exists
2:   for each  $val \in D(curV)$  //each value of  $curV$  in its domain
    // Assignment  $\langle curV, val \rangle$  is inconsistency with  $partialSol$ 
3:   if procedure CheckConsistency-BT( $curV, val, partialSol, C^a$ ) is false
4:      $D(curV) = D(curV) - val$ ; //remove inconsistent value  $val$ 
5:   else //Assign value to  $curV$ , add to partial solution, calculate total cost
6:     procedure Assign( $curV, val, partialSol$ );
7:     procedure CalculateCost( $curV, costResult$ );
8:     break;
9:   end for
10:  if  $D(curV) == \emptyset$  procedure Backtracking( $partialSol, D^a, C^a, costResult$ );
11: end while
12: return  $partialSol, costResult$ 
```

subsequent introduction of new active constraints. In each stable phase, the CSP solver performs a general backtracking procedure on a fixed constraint network. If a consistent partial solution to all active variables is reached without any new variables activated, the partial solution is a final solution. Otherwise, the CSP solver continues

Algorithm 4.3: Procedure Backtracking(*partialSol*, D^a , C^a , *costResult*):
Backtracking to the last assigned variable in the partial solution.

```

1: if partialSol ==  $\emptyset$ 
2:   return partialSol, costResult;
   //remove lv's last assignment from partial solution, related cost is also deleted
3: select the last assigned variable lv=val in partialSol
4:    $D(lv) = D(lv) - val$ ;
5:   partialSol = partialSol -  $\langle lv, val \rangle$ ;
6:   costResult = costResult - lv.cost;
7: for each value  $\in D(lv)$       //reassign lv from its remaining domain
   // Assignment  $\langle lv, value \rangle$  is inconsistency with partialSol
8:   if procedure CheckConsistency-BT(lv, value, partialSol,  $C^a$ ) == false
9:      $D(lv) = D(lv) - value$ ;      //remove inconsistent value value
10:  else      //Assign value to lv, add to partial solution, calculate total cost
11:    procedure Assign(lv, value, partialSol);
12:    procedure CalculateCost(lv, costResult);
13:    break;
14: end for
15: if  $D(lv) == \emptyset$       procedure Backtracking(partialSol,  $D^a$ ,  $C^a$ , costResult);
16: return partialSol, costResult

```

Algorithm 4.4: Procedure CheckConsistency-BT(*var*, *value*, *partialSol*, C^a):
Check consistency between an assignment with partial solution.

```

1: if partialSol ==  $\{\}$       return true;
2: for each constraint c  $\in C^a$  related to var
3:   if all variables in c are instantiated
4:     if assignment  $\langle var, value \rangle$  is inconsistent with partialSol
5:       return false;
6: end for      //constrain c
7: return true;      //consistent

```

with the new extended network.

Limitations. This scheme is more suitable for DCSP including variables with discrete values and binary constraints. Its efficiency drops considerably when n -ary constraints are involved. It is also difficult to achieve search efficiency for constraints over continuous variables which are often expressed in mathematical formulae. Under these conditions, constraint order checked in the solving process needs to be considered. Although this augmented BT scheme constructs the constraint network dynamically and then controls the number of variables in each searching phase, it still

Figure 4.2: Search trees of GT and BT for non-solution CSP problem.

has the disadvantages of ordinary BT algorithm. One of them is trashing. This is repeated failures for the same reason in different parts of search space as it does not identify the conflicting variables—the real reason of the conflict. Another drawback is that BT detects conflicts too late. It is not able to detect conflict before the conflict really happens as it checks the consistency of a constraint until all its variables are instantiated [14]. Many algorithms have been proposed to address these issues. This includes backjumping (BJ) and dependency-directed backtracking to address the issue of trashing; and look-ahead strategies to address the issue of late conflict detection.

Complexity Analysis. In this thesis, the time complexity of a CSP search algorithm is represented by the number of partial solutions explored by the algorithm, each of which corresponds to a node of the search tree visited by the algorithm assuming that visiting each node takes constant time.

Assuming that there are n variables and e constraints in the DCSP-based system, the maximum domain size of variables is d , the time complexity of procedure *ActivateModuleVar()* and *ActivatePropertyCostVar()* are $O(n)$, and $O(e)$ for procedure *ActivateConstraint()*, the time complexity of procedure *ConstraintPropagate()* is $O(d^n)$. The entire augmented BT algorithm for the extended DCSP-based product configuration and cost estimation system is $O(n + e + d^n)$.

Although BT algorithm has the time complexity of $O(d^n)$ in the worst case (i.e., there is no solution), which is the same as generate-and-test(GT) paradigm that has a search space as the Cartesian product of all variables' domains, its ability to prune a subspace from the search space makes it more efficient in most cases. Figure 4.2

sketches search trees when GT and BT are implemented on a non-solution problem respectively, where each edge refers to an assignment to a variable, and each level in a branch of the tree represents to values of a variable. In Figure 4.2(a), all combinations of values of variables are generated and tested before non-solution is detected. In Figure 4.2(b) all combinations in the left two branches are skipped since the first two assignments to the first variable are inconsistent. In addition, in our augmented BT solving process, the whole solving process may be implemented in more than one phase, in each of which the number of active variables is not more than the number of all variables in the entire DCSP; i.e., $n^{act} \leq n$. In any phase, if no consistent partial solution is detected, the solving process terminates, and inactive variables will not be checked. In this situation, the complexity of BT is $O(d^{n^{act}}) \leq O(d^n)$.

4.2 Implementation of Cost Constraints in the BT Solving Process

As we have mentioned in Section 3.3.5, there are two categories of cost constraints; cost calculation constraint and cost variation constraint. In the augmented BT solving process, these two categories are implemented in different phases. In the constraint propagation phase (Step 4 in Figure 4.1), *cost calculation constraints* are implemented as ordinary constraints (e.g., compatibility constraints). The phase checks constraint consistency in a constraint network N and calculates the result of cost estimation. So when a configuration solution is reached, cost estimation solution is also achieved. In addition, the results of cost calculation constraints can be used as a concurrent criterion to evaluate a configuration. Hence, cost estimation cooperates with the configuration task tightly. For instance, using cost calculation constraint “*monitor.price* $\leq$ \$300”, some monitor models that are more expensive than \$300 will be eliminated from the domain of module variable *monitor*. According to constraint “ $\sum(\textit{mouse.price}, \textit{keyboard.price}, \textit{speaker.price}, \textit{webcamera.price}) \leq \1000 ”, the instantiation of these module variables can be checked, in addition to other generic constraints; e.g., compatibility constraints.

Algorithm 4.5: Implementation of Cost Variation Constraints

Input: Initial DCSP-based configuration task $CP = \langle V^I, D^I, C^I, \Phi \rangle$, where $V^I = \{\text{all changed variables } chv \mid chv \in V_M \cup V_{CM} \cup V_P \cup V_{CT}\}$, $C^I = \{\text{cost variation constraints related to } V^I\}$, Note: C^a are only required to include at least one active variable in V^a , instead all variables are active.

Output: Cost variation result $costChange$ to CP .

```

1:  $V^a = V^I$ ;  $D^a = D^I$ ;  $C^a = C^I$ ;  $costChange=0$ ;
2:  $C_{var}^a = C^I$ ;  $C_{cal}^a = \emptyset$ ;      //the set for active cost variation and calculation
   constraints
3: for each  $cv \in V^a$ 
4:    $newV^a = \{\}$ ;
   //activate cost calculation constraints related to  $cv$  into  $C_{cal}^a$ .
5:   procedure ActivateCalculus( $cv, C_{cal}^a, \Phi$ );
   //get variables directly affected by  $cv$ , and assign the changed value to them
6:    $newV^a = \text{procedure}$  GetAffectedVar( $cv, C_{var}^a$ );
7:   if  $newV^a \neq \emptyset$ 
8:      $V^a = V^a \cup newV^a$ 
     //activate cost variation constraints related to variables in  $newV^a$ .
9:     procedure ActivateVariationCons( $newV^a, C_{var}^a, \Phi$ );
10:     $C^a = C^a \cup C_{var}^a \cup C_{cal}^a$ ;
11:  end for
12: return  $costChange = \text{procedure}$  CalculateChange( $V^a, C_{cal}^a$ );

```

In the general product configuration and cost estimation process described above, *cost variation constraints* are not activated into the constraint network N for constraint consistency check. In fact, they are implemented after a product configuration is finished and are only performed in the cost variation constraint handling phase (Step 9 marked by dotted frame in Figure 4.1) to estimate the cost impact of any change of variables on a product, an entire product family, or the total PLCC. With a given configuration, all changed variables in the extended DCSP-based system form the initial active variable set V^I of N . Then a changed variable is chosen as the current variable cv (Step 3). By cost variation constraints, variables that are directly affected by cv are introduced into N , the changed-value of those variables are calculated (Step 9), and assigned to them (Step 6). This process, marked by dotted arrows in Figure 4.1, continues iteratively until all affected variables, directly or indirectly, are identified and added into N . Finally, by cost calculation constraints, the cost impact

Figure 4.3: System architecture of DCSP-based desktop PC configuration task.

result of variable changes on a product family or PLCC can be achieved. Taken in this sense, cost variance constraints trigger new active variables as activation constraints to extend the constraint network. This process is presented in Algorithm 4.5.

4.3 Experiments

In the previous sections, we proposed an extended DCSP-based product configuration with cost estimation model, described the mapping relationship between PDMs and the extended DCSP-based system, and provided an augmented BT solving algorithm. In this section, we implement the extended DCSP-based system and the solving algorithm on a desktop personal computer (PC) configuration task, which is called “the extended DCSP-based PC system” for short in the following sections.

The structure of the extended DCSP-based PC system is sketched in Figure 4.3. In this system, the formalism of DCSP is generated from product family data and related design knowledge stored in a MSSQL database. The basic class relationship diagram for the database is illustrated in Figure 4.4. Due to their variability, customer requirements for a PC are parsed into XML files instead of a database. Figure 4.5 shows an example of the XML file. Based on the configuration domain Φ stored in the database and customer requirements in XML files, a DCSP-based configuration engine that deploys an open source Java-based CSP solving library Choco, which provides essential Java classes handling the constraint implement, is executed to generate valid configuration and cost estimation results.

Figure 4.4: Class diagram of DCSP-based desktop PC configuration task.

As mentioned previously, problem representation and solving are the two main issues in PF design. First, a desktop PC configuration task is modeled as an extended DCSP-based configuration and cost estimation system in Section 4.3.1. In Section 4.3.2, the augmented BT solving algorithm is deployed on the extended DCSP-based PC system to achieve the final configuration and cost estimation solution. Finally, experimental results are compared and discussed in Section 4.3.3.

4.3.1 Modeling DCSP-Based Desktop PC Configuration Problem from PDMs

Recall that in an extended DCSP-based configuration and cost estimation problem $\langle V^I, D^I, C^I, \Phi \rangle$, variables involved in the final solution to the extended DCSP-based system are activated from the configuration domain $\Phi = \langle V_M, V_{CM}, V_P, V_{CT}, D, C_s, C_a, C_g \rangle$, where the formulism can be derived from various PDMs. In this section, we

Figure 4.5: A example for requirement XML file.

Figure 4.6: PFA of Desktop PC configuration.

describe the formalism deriving process.

The PFA of a desktop PC, as shown in Figure 4.6, includes 16 modules, of which 4 underlined modules (independent graphics card, floppy disk, speaker and webcam) are differentiation modules, and the other 12 modules are mandatory. As defined in Section 3.5.4, from the denotation $Module : (\vec{IC}; \vec{A}; \vec{C}; \vec{CM})$, module variables V_M , component variables V_{CM} , property variables V_P , and cost variables V_{CT} in the DCSP are derived. All of these formalisms are listed in Table 4.1, where elements of V_M , V_{CM} , V_P and V_{CT} are prefixed with m , c , p , and ct respectively. Component variables in V_{CM} corresponding to each module variable m also construct m 's domain, as listed in the second column of Table 4.1. For example, module variable $mMonitor$ has eight instance component variables, then $Domain(mMonitor) = \{cM_1, ..., cM_8\}$. The name of variables from V_P and V_{CT} are constructed by both the original module variable and the name of property or cost variable; e.g., property “the brand of monitor” is represented as $mMonitor.pBrand$. Finally, these property and cost variables are inherited by the instance component of the module, such as $cM_1.pBrand$.

Activation constraints C_a represent the association between modules and properties and cost drivers; i.e., $\vec{A}$ and $\vec{C}$ of a module, can also be derived and are listed in

Table 4.1: Variables and activation constraints in DCSP.

$V_M (m)$	$V_{CM} (c)$ ($Domain(V_M)$)	$V_P (p)$	$V_{CT} (t)$	C_a $m \Rightarrow p, t$
mPC	$cFinalPC$		$ctCost,$ $ctAssm$ $ctProfitRate$	CA_1
$mHost$	$cFinalHost$		$ctCost$	CA_3
$mAcces$	$cFinalAcs$		$ctCost$	CA_4
$mMonitor$	$cM_1 \dots cM_8$	$pBrand, pType, pSize,$ $pColor, pResolution$	$ctUnit$ $ctProcure$ $ctStorage$ $ctTransp$ $ctPrice$ $ctManday$	CA_2
$mChassis$	$cCH_1 \dots cCH_5$	$pBrand, pType, p35No,$ $pColor, pMainBd$		CA_5
$mMotherBd$	$cMB_1 \dots cMB_7$	$pChip, pType, pInteg,$ $pCPUType, pMemNo$ $pMemCapac, pGraph$ $pCPUSlot, pMemTyp$		CA_6
$mCPU$	$cCPU_1 \dots cCPU_9$	$pBrand, pType, pSlot,$ $pMemTyp$		CA_7
$mHardDisk$	$cHD_1 \dots cHD_{10}$	$pBrand, pCapac$		CA_9
$mDVD$	$cDVD_1 \dots cDVD_6$	$pBrand, pType$		CA_{10}
$mGraphCd$	$cGC_1 \dots cGC_5$	$pSlot, pCardMem$ $pResolution$		CA_{11}
$mFloppy$	cF_1	$pSize$		CA_{12}
$mMouse$	$cMS_1 \dots cMS_9$	$pBrand, pType,$ $pWire, pColor$		CA_{13}
$mKeyBd$	$cKB_1 \dots cKB_{10}$	$pBrand, pWire, pErg,$ $pColor, pWaterpf$		CA_{14}
$mWebcam$	$cWC_1 \dots cWC_{11}$	$pBrand, pColor,$ $pResolution$		CA_{15}
$mSpeaker$	$cSP_1 \dots cSP_{10}$	$pBrand, pType, pColor$ $pPower, pFashion$		CA_{16}
$mMemory$	$cMM_1 \dots cMM_9$	$pBrand, pType, pNo,$ $pCapac, pTotalCapac$	$Same as$ $above, ctAll$	CA_8

the last column of Table 4.1.

Structural constraints C_s of DCSP are generated from parent-child relationships between modules; i.e., $\overrightarrow{CM}$ of a module, as listed below.

- CS_1 : $mPC \Downarrow \{mMonitor, mHost, mAcces\}$;
- CS_2 : $mHost \Downarrow \{mChassis, mMotherBd, mMemory, mCPU, mHardDisk, mDVD, \sim mGraphCd, \sim mFloppy\}$, where $mGraphCd$, and $mFloppy$ are

optional;

- CS_3 : $mAcces \Downarrow \{mMouse, mKeyBd, \sim mSpeaker, \sim mWebCam\}$, where $mSpeaker$ and $mWebCam$ are optional.

In Section 3.6, using the association rule mining approach, 12 compatibility constraints (Table 3.3) and 4 incompatibility constraints (Table 3.4) are acquired from historical data, which are denoted as $CM_1 \dots CM_{16}$ respectively. All compatibility constraints are included as part of the generic constraints $C_g \subset \Phi$.

As presented in Section 3.5.3, design knowledge provides background information for configuration, and can be mapped to various generic constraints C_g in the DCSP; e.g., compatibility constraints, equality constraints, cost constraints, and preference constraints. First, we provide some configuration rules generated from the design knowledge of desktop PC configuration as listed below. Corresponding equality and inequality constraints are also generated.

- The compatible motherboard type of module “chassis” must be the same as the type of module “motherboard”;

$$CE_1 : mChassis.pMainBd = mMotherBd.pType;$$

- The type of compatible CPU of “motherboard” must be the same as that of selected “CPU”;

$$CE_2 : mMotherBd.pCPUType = mCPU.pType;$$

- The type of compatible CPU slot of “motherboard” must be the same as the slot of selected “CPU”;

$$CE_3 : mMotherBd.pCPUSlot = mCPU.pSlot;$$

- The type of compatible memory of “motherboard”, compatible memory in “CPU” and selected “memory” must be the same;

$$CE_4 : mMotherBd.pMemTyp = mCPU.pMemTyp = mMemory.pType;$$

- The maximum capacity of memory in “motherboard” must be not less than the total capacity of all memories;

$$CE_5 : mMotherBd.pMemCapac \geq mMemory.pTotalCapac;$$

- The total capacity of “memory” is the sum of the capacity of all memories;

$$CE_6 : mMemory.pTotalCapac = mMemory.pCapac * mMemory.pNo;$$

- The maximum amount of memory slot in “motherboard” must be not less than the amount of used “memory”;

$$CE_7 : mMotherBd.pMemNo \geq mMemory.pNo;$$

- The slot type of “independent graphics card” must be the same as that in “motherboard”;

$$CE_8 : mGraghCd.pSlot = mMotherBd.pGraph;$$

- The color of “mouse” and “keyboard” should be similar; that is one cannot be white and another black;

$$CE_9 : mMouse.pColor \approx mKeyBd.pColor;$$

- Similarly, the color of webcam and monitor should be the similar.

$$CE_{10} : mWebcam.pColor \approx mMonitor.pColor.$$

- The resolution of webcam must be no more than that of monitor.

$$CE_{11} : mWebcam.pResolution \leq mMonitor.pResolution.$$

As stated previously, equality and inequality constraints usually induce compatibility or incompatibility constraints on components. For example, given $cMS_1.pColor = black$ and $cKB_4.pColor = dark\ blue$, according to CE_9 , $\{cMS_1, cKB_4\}$ is a compatible component tuple for the combination of $mMouse$ and $mKeyBd$.

In addition, cost constraints can be defined from design knowledge as follows:

- The unit cost $ctUnit_j$ of each leaf component j (i.e., the component listed in rows 5 to 17 of Table 4.1) is calculated as:

$$CC_i : ctUnit_j = \sum_j [ctProcure_j, ctStorage_j, ctTransp_j], \text{ where}$$

- Procurement cost $ctProcure_j = \sum[ctPrice_j, T_j^{man-day} \times ctManday_j]$, where $T_j^{man-day}$ is the number of man-days required to procure component j , $ctManday_j$ is the man-day rate in dollar terms.
 - Storage cost $ctStorage_j = f\{R_j^{failure}, ctP_j^{mass}, ctP_j^{samll}, Qt_j^{st}, T_j^{st}, ctManday_j\}$, where $R_j^{failure}$ is the failure rate of component j , ctP_j^{mass} is the mass procurement cost, ctP_j^{samll} is the small-batch procurement cost, Qt_j^{st} is the units of component j stored, T_j^{st} is the average storage time of j .
 - Transportation cost $ctTransp_j = f\{T_j^{leadtime}, Qt_j^{trans}, ctT^{mass}, ctT^{small}\}$, where $T_j^{leadtime}$ is the delivery lead-time, Qt_j^{trans} is the quantity for transportation, ctT^{mass} and ctT^{small} are transportation cost for mass and small-batch procurement respectively.
- The total cost of memories is calculated by the unit cost of memory multiplies the number of memory units;
 $CC_9 : mMemory.ctAll = mMemory.ctUnit \times mMemory.pNo;$
 - The cost of host and accessory are the sum of the cost of all their mandatory constituents respectively.
 $CC_3 : mHost.ctCost = mChassis.ctUnit + mMotherBd.ctUnit + mCPU.ctUnit + mHardDisk.ctUnit + mMemory.ctAll + mDVD.ctUnit;$
 $CC_4 : mAcces.ctCost = mMouse.ctUnit + mKeyBd.ctUnit;$
 - The cost of the entire PC is defined as:
 $CC_1 : mPC.ctCost = (\sum ctUnit_j + mPC.ctAssm) \times mPC.ctProfitRate$, where
 - $\sum ctUnit_j$ is the sum of the unit cost of all components; i.e., $\sum ctUnit_j = mMonitor.ctUnit + mHost.ctCost + mAcces.ctCost$.
 - $ctAssm = T^{assm} \times ctManday^{assm}$ is the assembly cost for a PC, where T^{assm} and $ctManday^{assm}$ are time and man-day rate for assembly respectively.

- $ctProfitRate = f\{ctOverhead, ctAntiProfit\}$ is the profit rate of a PC, which is calculated based on company's anticipant profit rate $ctAntiProfit$ and overhead costs $ctOverhead$, such as management costs, power, rental.

Also, six constraints of preference on constraints, which are related to optional modules in the extended DCSP-based PC system, may be derived as follows:

- If a floppy disk is selected, the amount of 3.5' slots of motherboard must be more than 1;

$$CP_1 : (mFloppy = true) \rightarrow (mChassis.p35No > 1);$$

- If an optional module is selected, its cost is added to total cost of their parent modules.

$$CP_2 : (mFloppy = true) \rightarrow (mHost.ctCost = mHost.ctCost + mFloppy.ctUnit);$$

$$CP_3 : (mGraphCd = true) \rightarrow (mHost.ctCost = mHost.ctCost + mGraphCd.ctUnit);$$

$$CP_4 : (mWebcam = true) \rightarrow (mAcces.ctCost = mAcces.ctCost + mWebcam.ctUnit);$$

$$CP_5 : (mSpeaker = true) \rightarrow (mAcces.ctCost = mAcces.ctCost + mSpeaker.ctUnit);$$

As an essential part in a configuration task, constraints C_g on requirements for each module are listed as follows:

$$CR_1 : mMonitor.pBrand = Philips;$$

$$CR_2 : mMonitor.pType = WLCD;$$

$$CR_3 : mMonitor.pColor \neq (White, Black);$$

$$CR_4 : mChassis.pType = MT, \text{ which means a mini-tower chassis is required;}$$

$$CR_5 : mChassis.pColor = White;$$

$$CR_6 : mMotherBd.pType = ATX;$$

$$CR_7 : mMotherBd.pChip = Intel;$$

$$CR_8 : mCPU.pType = IntelCore2Duo;$$

$$CR_9 : mMemory.pType = DDR2;$$

$$CR_{10} : mMemory.pTotalCapac = 8G;$$

$$CR_{11} : mHardDisk.pBrand = Seagate;$$

$CR_{12} : mHardDisk.pCapac = 500G;$

$CR_{13} : mGraphCd = true;$

$CR_{14} : mGraphCd.pCost \leq 1000;$

$CR_{15} : mDVD.pBrand = Sony;$

$CR_{16} : mDVD.pType = Build-in;$

$CR_{17} : mMouse.pWire = Wire$, which means a wire mouse is required;

$CR_{18} : mMouse.pColor = White;$

$CR_{19} : mKeyBd.pErg = Yes$, which means an ergonomic keyboard is desired;

$CP_6 : Min(mPC.ctCost) \mid \{mMonitor.ctUnit, mHost.ctCost, mAcces.ctCost\},$

which is a preference on solution.

By now, the configuration domain $\Phi = \langle V_M, V_{CM}, V_P, V_{CT}, D, C_s, C_a, C_g \rangle$ for the extended DCSP-based desktop PC configuration and cost estimation task is constructed, which consists of 16 module variables V_M with corresponding activation constraints C_a , 103 component variables V_{CM} , 53 property variables V_P , 84 cost variables V_{CT} , 3 structural constraints C_s , and 53 generic constraints C_g . Based on Φ , the desktop PC configuration task is initialized as $\langle V^I, D^I, C^I \rangle$, where $V^I = \{mPC\}$; D^I is $Domain(mPC) = cFinalPC$; and $C^I = \{CS_1, CA_1, CR_{13}\}$, where CS_1 and CA_1 are structural constraint and activation constraint of the initial variable mPC , CR_{13} is the requirement on the selection of optional module “ $mGraphCd$ ”.

4.3.2 Solving DCSP-Based PC Configuration by Augmented BT Algorithm

After modeling the extended DCSP-based PC system, we search its solutions using the augmented Backtracking searching process proposed in Section 4.1.

In DCSP, active variables V^a involved in the final solution are generated dynamically. Table 4.2 lists the activation process of variables and constraints in the extended DCSP-based PC system, where column “New V^a (Triggered by constraints)” lists new V^a in each step, and constraints in parentheses refer to the constraints triggering the

Table 4.2: The activation of V^a and C^a in the backtracking searching process.

Activation	New V^a (Triggered by constraints)	New C^a
(1)	mPC (Initial variable); cFinalPC; V_{CT} of mPC (CA_1)	$CS_1; CA_1;$ CR_{13}
(2)	mMonitor, mHost, mAcces (CS_1); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_2 - CA_4)	$CS_2, CS_3; CC_1;$ $CC_2; CA_2$ - $CA_4;$ $CP_6; CR_1$ - CR_3
(3)	mChassis, mMotherBd, mCPU, mHardDisk, mMemory, mDVD(CS_2), mGraphCd(CR_{13}); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_5 - CA_{11})	CA_5 - $CA_{11}; CE_1$ - $CE_8;$ CC_5 - $CC_{12}; CC_3;$ CM_4 - $CM_{16};$ $CP_3; CR_4$ - CR_{16}
(4)	mMouse, mKeyBd (CS_3); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_{13}, CA_{14})	$CA_{13}, CA_{14}; CE_9;$ $CC_4; CC_{14}; CC_{15};$ CM_1 - $CM_3; CR_{17}$ - CR_{19}

variables to become active; for example in “Activation (1)”, all the variables $\in V_{CT}$ of module mPC , such as $mPC.ctCost$, are activated by activation constraint CA_1 .

Figure 4.7 illustrates a simple process of the augmented BT solving process on the extended DCSP-based PC system modeled in the previous section. Table 4.3 describes the steps of the searching process sketched in Figure 4.7. Each step lists the current variable cv that is assigned. In the CSP solving process, consistency techniques are able to efficiently rule out inconsistent assignments at a relatively early stage, and thus, pruning the search space [14]. In our experiments, the consistency of unary constraints on cv , which are mainly derived from customer’s requirements, is first checked as they restrict feasible values of singular variables, (temporarily) remove inconsistent values from its domain which do not satisfy unary constraints, and thus, more directly prune the search space. It is called *node consistency*. Then subsequent constraint propagation and backtracking are only implemented within cv ’s remaining values that satisfy customer requirements. This can improve the searching efficiency due to a smaller value space. That is, *arc consistency* check temporarily removes inconsistent values from cv ’s left domain that conflict with binary constraints between cv and past instantiated variables. If cv ’s domain becomes empty, BT is implemented. The searching process is backward to the immediate previously assigned variable, and

Figure 4.7: A backtracking process for the example PC configuration.

the last assigned value of the immediate past variable is temporarily removed from its domain. The violated constraint that leads to *cv*'s empty domain is underlined in Table 4.3 (e.g., CE_5 in Step 7).

The searching process for the extended DCSP-based PC system starts from the initial DCSP $\langle \{mPC\}, \{cFinalPC\}, \{CS_1, CA_1, CR_{13}\} \rangle$ modeled in the last section. Three cost variables of *mPC* (*ctCost*, *ctAssm*, *ctProfitRate*) are activated by activation constraint CA_1 as summarized in "Activation (1)" in Table 4.2. Subsequently,

$mMonitor$, $mHost$, $mAcces$ are activated by structural constraint CS_1 . Corresponding variables in V_{CM} , V_P and V_{CT} of these three new active module variables are also activated into V^a (Activation (2) in Table 4.2). In this way, new V^a are introduced into the search space based on PFA and the solving process of DCSP is implemented following this hierarchy (depth-first search).

Then, each variable in V^a is assigned sequentially. In Step 5, $mMotherBd$ is checked. Pruned by requirement constraints CR_6 , CR_7 , equality constraint CE_1 , and compatibility constraint CM_{11} , CM_{12} , its remaining values are $\{cMB_1, cMB_2, cMB_3\}$. As the lowest cost is preferred by preference constraint CP_6 , cMB_1 is chosen due to its lowest price. Similarly, $cCPU_4$ is assigned to $mCPU$ in Step 6. In Step 7, the domain of $mMemory$ becomes empty due to constraint CE_5 , where $mMotherBd.pMemCapac$ refers to $cCH_8.pMemCapac = 4G$ and $mMemory.pTotalCapac = 8G$ according to CR_{10} . Thus, BT is performed and the solving process is tracked backward to $mCPU$ (Step 8). The last assignment $cCPU_4$ is removed and the only remaining value $cCPU_5$ is assigned to $mCPU$.

However, in Step 9, the assignment of $mMemory$ fails due to the same constraint CE_5 as in Step 7. The process is back to $mCPU$ again (Step 10), and the last assigned value $cCPU_5$ is removed. Consequently, the domain of $mCPU$ is empty. Hence, BT draws the searching process to $mMotherBd$ (Step 11) that is the previous assigned variable before $mCPU$, and its last assigned value cMB_1 is pruned. This time cMB_2 is assigned to $mMotherBd$. In the following step, the domain of $mCPU$ is recovered as the solving process is backtracked to its immediate past variable. Finally, $cCPU_4$ and cMM_7 are assigned to $mCPU$, $mMemory$ respectively.

When each variable has been assigned, its related cost is calculated by its corresponding cost constraints CC_1 - CC_{17} and preference constraint CP_3 related to the cost of the optional module. Then the cost can be implemented as a continuous criterion for a configuration task. For example, restricted by preference constraint CP_6 , the minimum cost of the whole product is desired. Thus, the compatible component

Table 4.3: Backtracking searching process of the example case.

	Module	Node Consistency	Left Components	Arc Consistency	Left Components	Cost & Preference	Selected Component
1	mPC	CR_{13}				CC_1	cFinalPC
2	mMonitor	$CR_1 - CR_3$	cM_8			CC_2, CP_6	cM₈
3	mHost					CC_3, CP_3	cFinalHost
4	mChassis	CR_4, CR_5	cCH_3, cCH_5			CC_5, CP_6	cCH₅
5	mMotherBd	CR_6, CR_7	cMB_1, cMB_2, cMB_3	CE_1, CM_{11}, CM_{12}	cMB_1, cMB_2, cMB_3	CC_6, CP_6	cMB_1
6	mCPU	CR_8	$cCPU_4, cCPU_5$	$CE_2, CE_3, CE_4, CM_9, CM_{13}$	$cCPU_4, cCPU_5$	CC_7, CP_6	$cCPU_4$
7	mMemory	CR_9, CR_{10}	$cMM_1, cMM_2, cMM_4, cMM_7, cMM_8, cMM_9$	$CE_4, CE_5, CE_6, CE_7, CM_7, CM_{15}$	$\{\}$		<i>Backtracking</i>
8	mCPU	CR_8	$cCPU_5$	$CE_2, CE_3, CE_4, CM_9, CM_{13}$	$cCPU_5$	CC_7, CP_6	$cCPU_5$
9	mMemory	CR_9, CR_{10}	$cMM_1, cMM_2, cMM_4, cMM_7, cMM_8, cMM_9$	$CE_4, CE_5, CE_6, CE_7, CM_7, CM_{15}$	$\{\}$		<i>Backtracking</i>
10	mCPU	CR_8	$\{\}$				<i>Backtracking</i>
11	mMotherBd	CR_6, CR_7	cMB_2, cMB_3	$CE_1, CM_4, CM_7, CM_9, CM_{11}, CM_{12}$	cMB_2, cMB_3	CC_6, CP_6	cMB₂
12	mCPU	CR_8	$cCPU_4, cCPU_5$	$CE_2 - CE_4, CM_4, CM_9, CM_{13}$	$cCPU_4, cCPU_5$	CC_7, CP_6	cCPU₄
13	mMemory	CR_9, CR_{10}	$cMM_1, cMM_2, cMM_4, cMM_7, cMM_8, cMM_9$	$CE_4, CE_5, CE_6, CE_7, CM_7, CM_{15}$	cMM_7	CC_8, CC_9, CP_6	cMM₇
14	mHardDisk	CR_{11}, CR_{12}	cHD_2			CC_{10}, CP_6	cHD₂
15	mDVD	CR_{15}, CR_{16}	$cDVD_3$			CC_{11}, CP_6	cDVD₃
16	mGraphCd	CR_{14}	cGC_1, cGC_2	$CE_8, CM_5, CM_6, CM_9, CM_{10}, CM_{14}$	cGC_1, cGC_2	CC_{12}, CP_6	cGC₁
17	mAcces					CC_4	cFinalAcs
18	mMouse	CR_{17}, CR_{18}	cMS_2, cMS_6			CC_{14}, CP_6	cMS₂
19	mKeyBd	CR_{19}	$cKB_1, cKB_4, cKB_5, cKB_6, cKB_8, cKB_{10}$	CE_9, CM_1, CM_2, CM_3	cKB_5	CC_{15}, CP_6	cKB₅

of each module with lowest *ctUnit* is selected. Finally, a valid solution to the extended DCSP-based PC system is obtained, denoted by the assignment flow in bold line reaching the end result in Figure 4.7. In Table 4.3 the solution is represented by selected components of each module in bold characters. The cost of each component and the entire PC are computed simultaneously.

The backtracking in Steps 7 and 9 are caused by the same constraint CE_5 . This situation is called “trashing” due to the failure of conflicting variable identification. For example, in Step 7, BT does not identify that the conflicting variable is *mMotherBd*, so it is backtracked to the immediate past variable *mCPU*, and repeats the failure in Step 9. Until the domain of *mCPU* is empty, the searching process finally returns to *mMotherBd*, the real reason of the failure, and continues until the final solution is obtained. Trashing is the main issue of BT. In our experiment, we apply node consistency to prune the search space to reduce the opportunity of trashing. As suitable variable ordering can also reduce the possibility of trashing, we activate and assign variable in depth-first order according to the hierarchy of the PFA of PC configuration. However, trashing is still unavoidable.

4.3.3 Experimental Results

To test the efficiency of the proposed extended DCSP-based configuration and cost estimation system with the augmented Backtracking algorithm, comparisons are made between the extended DCSP-based system with the augmented BT algorithm (denoted as “DCSP+BT” in the experimental results), and a classical CSP-based configuration system with standard backtracking algorithm (denoted as “CSP”); and between “DCSP+BT” and the case-based approach (denoted as “Case”). All experiments are conducted on a personal workstation with an Intel® Core™ 2 Quad Q9400 processor at 2.66 GHz with 4GB of RAM.

Figures 4.8 to 4.12 illustrate experimental results on solving efficiency. In the experiments, data storage and memory usage are not included for evaluating the solving

Figure 4.8: Experimental results for generating a feasible solution.

efficiency. Two comparison criteria are evaluated: the number of nodes visited and the CPU time consumed (in terms of clock clicks) in the searching process. The abscissa “number of selected modules” refers to the number of module variables involved in the final solution to a CSP/DCSP-based configuration system, and varies from 12 (including all mandatory modules) to 16 (including all optional modules). It also represents the number of module variables involved in the searching process of the extended DCSP-based PC system. Consequently, the number of component variables and constraints involved in the DCSP searching process also varies corresponding to the selected module variables. However, in classical CSP-based system, all 16 modules and all corresponding components and constraints (except structural and activation constraints) are involved in the searching process.

In the experiment for generating the first feasible solution, as shown in Figure 4.8, the extended DCSP-based PC system implementing the augmented BT algorithm outperformed the classical CSP system as it visits less nodes in the searching process.

Figure 4.9 illustrates experimental results between “DCSP+BT” and “CSP” for generating all solutions with respect to the number of visited nodes and CPU time respectively. The extended DCSP-based system with the augmented BT algorithms offers significant improvement compared with the CSP-based system, as the number of selected module variables, together with the number of components and constraints

4.9.a: Results on the no. of visited node.

4.9.b: Results on CPU time.

Figure 4.9: Experimental results for generating all configuration solutions (“DCSP+BT” vs. “CSP”).

increases. The more the active variable is involved, the greater performance improvement the extended DCSP-based system with the augmented BT algorithm is achieved.

Experiment results for “DCSP+BT” and “Case” for generating all solutions are shown in Figure 4.10. As the product family model and the rationale of case-based configuration approach are different from those of the DCSP-based configuration approach, the number of nodes visited in a searching process is not a meaningful criterion for comparison; CPU time in terms of clock clicks is used. It is observed that “DCSP+BT” outperforms the case-based configuration approach due to the less CPU time consumed. It is noted that in the result curve of “Case”, there is no data where the number of selected modules is 16. This is because when a new constituent that is never involved in historical configurations is added into the configuration tasks (e.g., “mWebcam” or a new module WLAN adapter), the case-based approach is infeasible when it comes to achieving a complete configuration for the new application. This scenario proves the advantage of DCSP-based configuration method; i.e., generality and flexibility. In the extended DCSP-based configuration system, the information of new constituents or new technologies, and their relationships with existing variables can be represented as variables and constraints in the DCSP, and then a full configuration result can be reached, if it exists.

Figure 4.10: Experimental results for generating all configuration solutions (“DCSP+BT” vs. “Case”)

As in the extended DCSP-based configuration system, cost constraints are integrated into DCSP as an instant criterion for configuration tasks. The performance of the extended DCSP-based system with the augmented BT algorithm is further checked for cost optimization (denoted as “DCSP+BT+Cost”) where minimum cost is preferred. First, its performance is compared with that of a classical CSP system with Branch-and-Bound (B&B) optimization (denoted as “CSP+BB”) (referring to Section 2.4.2, page 37). We also compare the performance of the augmented BT algorithm applied to the extended DCSP-based system integrating cost optimization with that of an extended DCSP-based system for yielding a feasible solution. Experimental results are illustrated in Figure 4.11 and Figure 4.12 respectively. In both comparisons, the extended DCSP-based system with cost constraints generally shows a better performance on optimization.

When incorporating cost optimization into the extended DCSP-based PC system, case-based approach is not a good choice and may yield unreliable results as costs of parts highly depend on market and are mutable. It is quite possible that the data of cost in historical configurations has no reference value. On the other hand, in DCSP-based configuration methodology, the data of cost can be updated whenever it is changed. This also highlights the advantage of DCSP-based configuration methodology; i.e., generality and flexibility.

Figure 4.11: Comparison of result optimization between “CSP+BB” and “DCSP+BT+Cost”.

Figure 4.12: Experimental results of DCSP with BT for the first feasible result (with vs. without cost optimization).

Hence, we conclude that the extended DCSP-based configuration system with augmented BT algorithm improves the search efficiency of configuration tasks. It outperforms the classical CSP-based configuration system in both solution searching and optimization, especially when multiple solutions are required. It also outperforms case-based configuration approach in solution searching. Moreover, it may overcome the inapplicability of case-based approach in both the application of new technologies and cost optimization.

4.4 Summary

Based on the extended DCSP-based product configuration and cost estimation model proposed in the previous chapter, various CSP algorithms can be applied to generate

solutions that assign a value to each active variable without violating all active constraints. In this chapter, we proposed an augmented Backtracking algorithm where structural constraint handler and activation constraint handler are embedded into a classical CPS solving procedure, both of which are able to trigger the extension of a DCSP-based system. Then in the solving process, the extended DCSP-based configuration and cost estimation system can be treated as a sequence of stable phases, each of which performs general backtracking algorithm on a fixed classical CSP. Details of the augmented BT algorithm is described, and its limitations and time complexity are discussed.

In the augmented Backtracking algorithm, cost calculation constraints are implemented as ordinary constraints that check constraint consistency in the DCSP solving process. Then the results of product configuration and cost estimation can be obtained simultaneously. The results of cost calculation constraints can be used as a concurrent criterion for product configuration. Hence, cost estimation is tightly integrated with product configuration. To study the impact of the change of cost drivers and design parameters on the cost of a product variant, or the entire product family, or even the whole product lifecycle cost, cost variation constraint handler is embedded into the extended DCSP-based system.

In the experiments, the procedures of modeling and solving an extended DCSP-based desktop PC configuration task using the augmented BT algorithm have been elaborated. Finally, experimental results are compared. We conclude that the extended DCSP-based configuration and cost estimation system with the augmented BT algorithm outperforms classical CSP-based configuration system in terms of both solution searching and optimization, especially when multiple solutions are required. It also outperforms case-based approach in solution searching, and may overcome the infeasibility of case-based approach on both the application of new technologies and cost optimization.

Chapter 5

Hybrid Algorithms for DCSP-Based Product Configuration with Cost Estimation

In previous chapters, an extended DCSP-based product configuration with cost estimation system was formalized (Chapter 3), and an augmented backtracking algorithm was proposed to solve it (Chapter 4). However, the augmented BT algorithm still suffers from trashing (i.e., repeated failures), and late conflict detection.

In this chapter, a hybrid solving algorithm is proposed for the extended DCSP-based configuration and cost estimation system to address two issues of the augmented BT algorithm—trashing and late conflict detection. Specifically, based on the same constraint network extension principles of the augmented BT algorithm (i.e., using structural and activation constraints, the constraint network of the DCSP-based system is incrementally constructed following the hierarchy of product family), forward checking-backjumping algorithm is applied to the DCSP constraint network for constraint propagation and assigns active variables with compatibility values to satisfy specific customer requirements. Moreover, a variable ordering heuristic—fail-first heuristic—is also adopted together with forward checking-backjumping algorithm to improve solving efficiency. Furthermore, to address the backtracking issue in forward checking-backjumping algorithm with fail-first heuristic, another variable ordering

heuristic is applied to the forward checking-backjumping algorithm to reduce backtracking occurrence. Limitations and complexity of the proposed algorithms are discussed. Finally, the proposed solving algorithms are implemented on the DCSP-based PC system, and experimental results are discussed.

This chapter is organized as follows: In Section 5.1, for better solving efficiency, an augmented forward checking-backjumping algorithm with fail-first heuristic is applied to the extended DCSP-based scheme to address the issues of the augmented backtracking algorithm proposed in Section 4.1. In Section 5.2, a hybrid heuristic—a combination of an amended most constraint heuristic and fail-first heuristic—is integrated with the forward checking-backjumping algorithm to improve the solving efficiency. Section 5.3 elaborates experiments implementing these two solving algorithms on the DCSP-based PC configuration and cost estimation system modeled in Section 4.3.1. Finally, conclusions are drawn in Section 5.4.

5.1 FF-BJ Algorithms for DCSP-Based Configuration

As discussed in Section 4.1, the augmented BT algorithm suffers from trashing, repeated failures, and late conflict detection. In this section, a hybrid CSP solving algorithm is implemented on the extended DCSP-based configuration and cost estimation system to address some of these issues.

A Forward Checking (FC) algorithm, the easiest of look-ahead strategies, addresses the issue of late detection of conflict in the BT algorithm. It checks arc consistency between an instantiated variable (the current variable) and an uninstantiated variable (a future variable) that has direct connection with the current variable [13], and thereby is able to prevent future conflicts. FC removes values from the future variable's domain which conflict with an assignment of the current variable. Whenever the domain of the future variable becomes empty, it is known that the current partial solution is inconsistent with future variables. Then another available unassigned value

is assigned to the current variable. If the domain of the current variable becomes empty, backtracking is performed to the last assigned variable (i.e., 1 step backwards). In FC, whenever a variable is assigned, all its available values left are guaranteed to be consistent with the current partial solution, so it is not necessary to check the assignment against the past assignments.

Algorithm backjumping (BJ) is proposed to address the issue of trashing in BT algorithm. It identifies the conflicting variable according to the violated constraint. The procedure of BJ is exactly the same as that of BT, except when backtracking occurs. That is when all the values of the current variable are discarded due to inconsistency, BJ backtracks to the most recent conflicting variable, while BT backtracks to the immediate recent assigned variable [14].

In this section, we apply a combination of these two algorithms—Forward Checking with Backjumping (FC-BJ)—which can backtrack more than one step backwards; i.e., backjump [147]. In FC-BJ, each variable v corresponds to a *conflict set* $conf(v)$ that records all variables whose value in the current partial solution is the reason for discarding values from v 's domain. The conflict set is updated to guide backjumping in the search process. Note that in the original FC, there is no consistency checking between the current variable and past variables. However, in the proposed extended DCSP-based configuration and cost estimation system, variables are activated into the search process dynamically. So when a new active variable is introduced into the DCSP, constraints between the variable and instantiated variables will be missed. Hence, in the FC-BJ algorithm, these constraints must first be checked; this can be viewed as filling the vacancy of previous variables' FC performance in the new phase. If the new active variable is inconsistent with the current partial solution, the FC-BJ solving process backjumps to the instantiated variable that leads to the inconsistency.

In a CSP solving process, CSP searching algorithms select variables in certain order to be assigned. The variable ordering can affect the efficiency of searching algorithms substantially [13, 14]. In the best case, good variable ordering can lead to a backtrack-free search. In the extended DCSP-based configuration and cost estimation system,

Figure 5.1: Flowchart for FC-BJ with FF algorithm on DCSP-based configuration.

as the active variable set is extended dynamically (that is, V^a is not fixed), static variable ordering is not feasible. Here we apply a dynamic variable ordering heuristic called “Fail-First (FF)” with the FC-BJ algorithm proposed above, which is one of the simplest and the most common variable ordering heuristics. In the solving process, FF selects a variable with the smallest current domain as the current variable to be instantiated as it seems that the variable with the fewest remaining values is most likely to fail. The FC-BJ algorithm is then applied to the current variable for removing values from the domain of its future variables, which are inconsistent with the current partial solution.

Specifically, Steps 3–6 in the augmented BT solving process for the extended DCSP-based configuration and cost estimation system (described in Section 4.1) are

updated to FC-BJ with FF algorithm as follows:

- Steps 1 and 2 are the same as that in the augmented BT solving process in Section 4.1 (page 86). Suitable variables and constraints are activated into the DCSP-based system, and initially, the conflict set of each active variable $conf(v)$ is empty.
- Step 3: An unassigned variable with the minimum domain size is chosen as current variable cv for checking.
- Step 4: Assuming that such an unassigned variable exists, if there are constraints between cv and instantiated variables pv in the current partial solution, these constraints must first be checked. If cv 's domain becomes empty, this means that cv is inconsistent with the current partial solution. The FC-BJ solving process backjumps to pv . The process jumps to Step 6, where cv refers to pv in Step 4 and fv refers to cv in Step 4.
- Step 5: A value val is assigned to cv , and assignment $\langle cv, val \rangle$ appends to the current partial solution.
- Step 6: FC is performed between cv and future variables fv . Values of fv , which are incompatible with the assignment $\langle cv, val \rangle$, are (temporarily) discarded from fv 's current domain; cv is added into the conflict set of fv ; i.e., $conf(fv) \cup \{cv\}$.
- Step 7: When backtracking occurs due to the empty domain of fv , FC-BJ backtracks to the last conflict variable stored in $conf(fv)$ (i.e., the current variable cv), discards its previous assignment val , and re-assigns cv from its remaining values. The conflict set $conf(cv)$ is updated to $conf(cv) \cup conf(fv) \setminus \{cv\}$ [147]. The values of fv removed by assignment $\langle cv, val \rangle$ are all restored, and cv is removed from $conf(fv)$. Similarly, if cv 's domain becomes empty, FC-BJ backjumps to the last conflict variable pv in $conf(cv)$, re-assigns pv , and updates $conf(pv)$ to $conf(pv) \cup conf(cv) \setminus \{pv\}$. Moreover, the assignment of variables

Algorithm 5.1: Augmented FC-BJ with FF Algorithm on DCSP-based System

Input: Initial DCSP-based configuration task $CP = \langle V^I, D^I, C^I, \Phi \rangle$;

Output: configuration solution sol and cost estimation result $ctSol$ to CP .

```

1:  $V^a = V^I$ ;  $D^a = D^I$ ;  $C^a = C^I$ ;  $sol = \{\}$ ;  $ctSol = 0$ ;
2: do
3:   Same procedures to active module, property and cost variables and
   constraints as in Algorithm 4.1 (Section 4.1). New active variables are stored in
    $newV^a$ ;
4:   for each variable  $v \in newV^a$ 
5:      $conf(v) = \{\}$ ; //initial conflict set
6:   end for
7:   procedure ConstrPropagateFC( $V^a, D^a, C^a, sol, ctSol$ ); //FF-BJ-FF
8:   if  $sol == \{\}$  output  $sol, ctSol$ ; //no solution
9: while  $newV^a \neq \emptyset$  //while new child modules exist
10: output  $sol, ctSol$ ;
```

that were performed between the previous assignment of pv and the assignment of cv are removed from the current partial solution. The values removed by these assignments are all restored. These variables are removed from the corresponding conflict sets. In the case when FC-BJ is performed backwards to each conflict variable, a consistent partial solution is still not found. We conclude that there is no solution to the DCSP.

- Steps 3–7, representing the FC-BJ searching procedure, iterate to assign a value to each active variable in N .
- Steps 8 and 9 are the same as the last two steps in the augmented BT solving process in Section 4.1 (page 87). The solving process extends to a new phase by the activation of new module variables, or terminates with the consistent partial solution as the final solution.

The procedure of the augmented FC-BJ algorithm with Fail-First variable ordering heuristic is described in Algorithm 5.1 to Algorithm 5.7, and illustrated in Figure 5.1.

Advantages and Limitations. Generally speaking, forward checking has almost always better performance than ordinary Backtracking [14]. As the forward checking

Algorithm 5.2: Procedure ConstrPropagateFC($V^a, D^a, C^a, partialSol, costResult$):
Constraint propagation process using FC-BJ with FF heuristic.

```

1:  $sol = partialSol$ ;      //store original partial solution
2: while  $unassignedVar \neq \emptyset$ 
3:   select current variable  $curV \in unassignedVar$  with min domain  $D(curV)$ 
   //FF
4:   if there constraints between  $curV$  and past variables in  $sol$ 
5:     procedure CheckBack( $curV, sol, ctSol, C^a$ );
6:   if  $sol == \{\}$ 
7:     return  $partialSol = \{\}$ ;      //no solution
8:    $fv = \text{procedure FC}(curV, unassignedVar, partialSol, costResult, C^a)$ 
9:   if  $D(curV) == \emptyset$ 
10:    procedure BJ( $curV, fv, partialSol, costResult$ );
11: end while
12: return  $partialSol, costResult$ 

```

Algorithm 5.3: Procedure CheckBack($cv, partialSol, costResult, C^a$): Check consistency current variable cv and past variables in $partialSol$.

```

1: for each  $\langle pv, value \rangle \in partialSol$  that has direct connection with  $cv$ 
2:   if procedure CheckConsistency-FC( $pv, value, cv, C^a$ ) ==  $\{cv\}$ 
   //inconsistent
3:     BJRestore( $cv, pv, partialSol, costResult$ );
4:   return;
5: end for      //pv

```

Algorithm 5.4: Procedure BJRestore($cv, pv, partialSol, costResult$): when backjumping occurs, solving process restores related data.

```

1: for each assignment  $asv = val$  from  $pv$  to  $cv$  in  $partialSol$ 
2:    $partialSol = partialSol - \langle asv, val \rangle$ ;
3:    $costResult = costResult - asv.cost$ ;
4:    $unassignedVar = unassignedVar + asv$ ;
5:   procedure RestoreValue( $asv, val$ ); //restore values removed by  $\langle asv, val \rangle$ 
6:   procedure RemoveVarFromConf( $asv, val$ ); //remove  $asv$  from conflict set
7: end for //asv

```

algorithm checks the consistency between the current variable and a future variable, consequently, it is able to detect inconsistency and thus prune branches of the search tree that will result in inconsistency earlier than the original BT algorithm, and hopefully reduce the overall amount of work done. Furthermore, as the remaining values in the current domain of each future variable are consistent with the current partial

Algorithm 5.5: Procedure $FC(cv, unassignedVar, partialSol, costResult, C^a)$:
Check consistency between the current variable cv and future variables.

```

1: for each  $value \in D(cv)$            //each value of  $cv$  in its domain
    // Forward check assignment  $\langle cv, value \rangle$  with unassigned variables
2:    $fv = \text{procedure CheckConsistency-FC}(cv, value, unassignedVar - cv, C^a)$ ;
3:   if  $fv == \{\}$            // consistent
4:     procedure  $Assign(cv, value, partialSol)$ ;
5:     procedure  $CalculateCost(cv, costResult)$ ;
6:      $unassignedVar = unassignedVar - cv$ ;
7:     break;           //jump out  $cv$ , assign next variable
8:   else           // inconsistent
9:      $D(cv) = D(cv) - value$ ;
10:  end for           //each  $value \in D(cv)$ 
11: return  $fv$ ;

```

Algorithm 5.6: Procedure $CheckConsistency-FC(cv, val, unassignedVar, C^a)$: Check consistency based on FC. If inconsistent, return future variable fv , else $\{\}$.

```

1: for each  $fv \in unassignedVar$  that has direct connection with  $cv$  //future
   variable
2:   for each constraint  $c \in C^a$  related to both  $cv$  and  $fv$ 
3:     for each  $value \in D(fv)$ 
4:       if  $\langle fv, value \rangle$  is inconsistency with  $\langle cv, val \rangle$ 
5:          $D(fv) = D(fv) - value$ ;           //remove inconsistent value from future
         var.
6:          $conf(fv) = conf(fv) + \langle cv, val \rangle$ ;           //add  $cv$  to  $fv$ 's conflict set
7:         break; //jump out for(value)
8:       end for           //value
9:     end for           //constrain  $c$ 
10:    if  $D(fv) == \emptyset$            return  $fv$ ;           //inconsistent
11:  end for           //fv
12: return  $\{\}$ ;           //consistent

```

solution, its size can be computed without any additional constraint checks [8]. Then Fail-First heuristic can be performed conveniently for dynamic variable reordering. In addition, the Backjumping algorithm backtracks to conflicting variables directly and, hence, addresses the issue of trashing in BT algorithm.

However, BJ still suffers from the issue of having to perform redundant work, as it does not remember the conflicting values of variables for detection of the same con-

Algorithm 5.7: Procedure $\text{BJ}(cv, fv, \text{partialSol}, \text{costResult})$: Solving process backjumps to the last conflict variable of a future variable fv , remove corresponding assignment from partial solution, related cost is also deleted.

```

1: if  $\text{partialSol} == \emptyset$       return  $\text{partialSol}, \text{costResult}$ ;
2: select the last assigned variable  $pv=val$  in  $\text{conf}(fv)$ 
3:  $D(pv) = D(pv) - val$ ;
4:  $\text{conf}(pv) = \text{conf}(pv) + \text{conf}(fv) - \{pv\}$ ;
5: procedure  $\text{BJRestore}(cv, pv, \text{partialSol}, \text{costResult})$ ;
6: procedure  $\text{FC}(pv, \text{unassignedVar}, \text{partialSol}, \text{costResult}, C^a)$ 
7: if  $D(pv) == \emptyset$ 
8:   procedure  $\text{BJ}(pv, pv, \text{partialSol}, \text{costResult})$ 
9: return  $\text{partialSol}, \text{costResult}$ 

```

flict in subsequent computations. It also seems that FC may lead to many redundant checks, as it checks against future variables that may never be assigned. For example, after the assignment $\langle v_1, val \rangle$, checking against v_7 is wasted if the solving process ends with v_3 and v_4 being incompatible with each other. Moreover, in the proposed extended DCSP-based system, variables are activated and introduced into the searching process dynamically. When a new active variable is assigned, constraint checking between the variable and instantiated variables may lead to backjumping and then more work. Hence, in the extended DCSP-based system, the augmented FC-BJ with FF is more suitable for the problem where there are few constraints between the variables in different levels of the searching tree; i.e., different phases in the searching process.

Complexity Analysis. Assuming that there are n variables with a maximum domain size of d and e constraints in the DCSP-based system, the time complexity of the activation procedure of module, property, and cost variables are $O(n)$, and that of constraint activation procedure is $O(e)$ (referring to complexity analysis of the augmented BT algorithm on page 90).

Generally, the time complexity of FC-BJ with the FF heuristics is $O(dn^2(d-1)^{n-2})$ [147], where for a constant $d > 1$, $O((d-1)^{n-2})$ is smaller than $O(d^{n-2})$ (then definitely smaller than $O(d^n)$), as $O(d^{n-2}) = O((\frac{d}{d-1})^{n-2} \times (d-1)^{n-2})$, where $(\frac{d}{d-1})^{n-2}$ is growing exponentially and cannot be ignored. As in each stable phase

i of the DCSP solving process, only active variables and constraints are involved in the searching process. The time complexity of the augmented FC-BJ with FF algorithm is amended as $O(d(n^{act_i})^2(d-1)^{n^{act_i}-2})$, where n^{act_i} refers to the number of active variables in the searching phase i , and $n^{act_{i+1}} > n^{act_i}$. Moreover, when a new stable phase i is formed (new variables and constraints are activated into the DCSP), constraint back-checking is required. In the worst case, in each new stable phase, all assigned variables are backtracked; the solving process backjumped to the first variable of the whole solving process. Thus, the time complexity of the FC-BJ with FF heuristic is $\sum_i d(n^{act_i})^2(d-1)^{n^{act_i}-2}$, where i refers to the number of searching phases.

As in the extended DCSP-based system, the activation of module variables, together with other variables, is based on product family architecture, where a parent module must have at least two child modules, the number of phases in the solving process is $i \leq \lceil \frac{n}{2} \rceil$. Thus, $\sum_i d(n^{act_i})^2(d-1)^{n^{act_i}-2} \leq \lceil \frac{n}{2} \rceil d(n^{act})^2(d-1)^{n^{act}-2} \leq (\frac{n}{2} + 1)d(n^{act})^2(d-1)^{n^{act}-2}$, where n^{act} refers to the number of active variables involved in the final solution. Then the time complexity of the augmented algorithm FC-BJ with FF is $O((\frac{n}{2} + 1)d(n^{act})^2(d-1)^{n^{act}-2})$. Taking the activation of variables and constraints into consideration, the time complexity of the whole solving process is $O(n^{act} + e^{act} + (\frac{n}{2} + 1)d(n^{act})^2(d-1)^{n^{act}-2})$, where $n^{act} \leq n$, and $e^{act} \leq e$.

5.2 A Hybrid Heuristic for DCSP-Based Configuration

As mentioned in the last section, the FC-BJ with FF heuristic may lead to a lot of backtracking when there are many constraints between variables activated in different phases, which reduces the search efficiency. In this section, a *hybrid heuristic* is proposed. The hybrid heuristic integrates an amended *most-constrained heuristic* with the augmented FC-BJ with FF algorithm for the extended DCSP-based product configuration and cost estimation system proposed in the last section to handle this condition for better solving efficiency.

In a solving process, a general *most-constrained heuristic* assigns a variable that is involved with the largest number of constraints with other *unassigned variables*. To control backtracking in the solving process of the augmented FC-BJ with FF algorithm, the *hybrid heuristic* activates modules and selects the variable to be assigned; both of which are involved with the largest number of constraints on *assigned variables* that are activated in previous search phases. The rationale is that in an extended DCSP-based configuration system, module variables, together with associated property variables and cost variables, are activated into the DCSP system incrementally. In the FC-BJ algorithm, arc consistency check is only performed between the current variable and its direct future variables, whenever a set of variables are activated into DCSP. Constraint consistency checks between the new activated variables and all assigned variables; i.e., variables in the current partial solution, are missed in previous solving process and must be performed first. Conflicts happening in the complementary consistency check process may lead to a lot backtracking. In the worst case, it may backtrack to the entire current partial assignment. Thus, selecting the variable involving in the largest number of constraints on assigned variables is able to detect conflicts as early as possible, and thus, shorten the possible backtracking process.

In product configuration, modules are identified in such a way that between-module interactions are minimal while within-module interactions are high (referring to Section 2.1.4). For instance, in the DCSP-based PC configuration example (Figure 4.6 in Section 4.3.1, page 95), interactions among the child modules of module “Mainframe” are stronger than interactions between these child modules with those of module “Accessory”. According to this nature of product configuration, in the extended DCSP-based configuration system, the activation of module variables is triggered according to the hierarchy of a product family; i.e., PFA. That is, a set of the child module variables of an assigned module variables are activated into DCSP to form a stable phase step by step, together with associated property variables and cost variables. Therefore, when inactive module variables are to be activated, the hybrid

heuristic activates the set of child modules of an assigned module that has the largest total number of constraints on assigned variables, instead of a single variable. In this set, the total number of constraints between each child module and assigned variables is the largest. Note that here only active generic constraints C_g in the DCSP-based configuration systems are considered; but structural constraints C_s and activation constraints C_a are excluded.

The reason to activate a set of child modules with the largest sum of constraints instead of a single child module with the largest number of constraints on assigned variables, together with its sibling module variables can be explained as follows. First, in the stable phase of the extended DCSP-based system, more constraints between new active variables and assigned variables are more likely to result in more steps of backtracking in the augmented FC-BJ algorithm. Second, in each step, the DCSP-based system activates a set of child modules according to PFA. However, even if a child module is involved in the largest number of constraints on assigned variables, it does not mean that the total number of constraints between it and its sibling module variables and assigned variables is the largest.

The hybrid heuristic integrating the augmented FC-BJ algorithm proposed in the last section (Section 5.1) is described as follows:

- The solving process starts from an initial constraint network $N = \langle V^I, D^I, C^I \rangle$.
- Step 1: In the structural constraint C_s handling phase, when new module variables are needed to be activated; i.e., all active variables in DCSP are instantiated, a set of child module variables of an assigned module variable will be activated into V^a according to the following rules:
 - For each assigned module variable m , the solving process decides which child modules will be activated, including all of m 's direct common child module variables, and some of its optional child modules according to specific requirements.

- Count the number of generic constraints C_g between each of these qualified child modules and all assigned variables pvs in the current partial solution. Note that only generic constraints C_g between those variables are considered, except structural constraints C_s and activation constraints C_a that are not checked in the searching process.
 - Then the set of child module variables of m that have the largest total number of constraints on pvs are activated into constraint network N .
 - If there is no constraint, activation of new child modules follows the order of the last activation phase. That is, the child modules of the first activated module variable in the last phase are activated first.
 - Component variables in the domain of each new active module variable are also added to N .
- Step 2: Using activation constraints C_a , an appropriate set of property and cost variables associated with the new active module variables in Step 1 are activated into N . Subsequently, constraints (except cost variation constraints) whose variables are all activated are also introduced into N . By now, a new fixed classical constraint network; i.e., a new stable phase, is generated. Initially, the conflict set of each active variable $conf(v)$ is empty.
 - Step 3: An unassigned variable is chosen as the current variable cv for checking according to a hybrid heuristic that integrates heuristics of amended most-constrained and Fail-First, and is described as follows:
 - Unassigned variables nv that are involved in constraints on pvs are selected in a descending order in terms of the number of constraints involved; that is, the variable involved in the largest number of constraints on pvs are selected as cv first, and the least the latest.

Algorithm 5.8: Augmented FC-BJ algorithm with hybrid heuristic on DCSP-based System

Input: Initial DCSP-based configuration task $CP = \langle V^I, D^I, C^I, \Phi \rangle$;
Output: configuration solution sol and cost estimation result $ctSol$ to CP .

```

1:  $V^a = V^I$ ;  $D^a = D^I$ ;  $C^a = C^I$ ;  $sol = \{\}$ ;  $ctSol = 0$ ;
2: do
3:    $newV^a = \{\}$ ; //storing new active variables
4:    $C_s = \{c \mid c \in C^a, c \text{ is a structural constraint}\}$ 
5:   procedure ActivateModuleVar_MostCons  $\{V^a, C_s, newV^a, D^a\}$ 
      //revised
6:   if  $newV^a == \emptyset$  //no new child module exists, process terminates
7:     output  $sol, ctSol$ ;
8:   procedure ActivatePropertyCostVar( $newV^a, C_{act}, D^a$ );
9:   procedure ActivateConstraint( $V^a, C^a$ );
      //By now, a new phase of  $N$  is built.
10:  for each variable  $v \in newV^a$ 
11:     $conf(v) = \{\}$ ; //initial conflict set
12:  end for
13:  procedure
    ConstrPropagateFC_MostCons( $V^a, D^a, C^a, sol, ctSol$ ); //revised
14:  if  $sol == \{\}$  output  $sol, ctSol$ ; //no solution
15: while  $newV^a \neq \emptyset$  //while new child modules exist
```

- If there are more than one nv with the same number of constraints on pvs , including no constraint, then the variable ordering follows the rule of Fail-First heuristic; i.e., the variable within them with the minimum domain size is selected, or the variable is selected according to the lexicographic ordering if more than one variables with the same domain size.
- The process that is the same as steps 4 - 9 in the procedure of FC-BJ with FF algorithm proposed in Section 5.1 (page 116) is performed to check back the consistency between cv and instantiated variables, and apply FC-BJ on cv and its direct future variables for a final solution.

The FC-BJ with FF algorithm proposed in Section 5.1 is revised for the hybrid heuristic by Algorithm 5.8 to Algorithm 5.10:

Complexity Analysis. Generally, when there is no constraint between variables activated in different phases, algorithm FC-BJ with the hybrid heuristic has the same

Algorithm 5.9: Procedure `ActivateModuleVar_MostCons` $\{activeVar, C_{str}, newActiveV, D^{act}\}$: Activate new child modules and domains with the largest number of constraints on assigned variables into N by structural constraints C_{str} .

```

1: for each module variable  $m \in activeVar$  whose child modules are inactive
2:    $chm = \text{procedure ChildModule}(m, C_{str});$ 
3:   if  $chm == \emptyset$  then break;
      //Count the total number of  $C_g$  between child modules and assigned variables
4:   procedure CountConstraints( $chm, activeVar$ );
5: end for
6:  $chm\_most = chm$  with the largest total number of  $C_g$  on instantiated variables
7: procedure ActivateVar( $chm\_most, newActiveV, D^{act}$ );
8: return  $newActiveV, D^{act}$ 

```

Algorithm 5.10: Procedure `ConstrPropagateFC_MostCons`($V^a, D^a, C^a, partialSol, costResult$): Constraint propagation process using FC-BJ with hybrid heuristic.

```

1:  $sol = partialSol;$  //store original partial solution
2: while  $unassignedVar \neq \emptyset$ 
      //select current variable  $curV \in unassignedVar$ 
3:   if there are constraints between  $nv \in unassignedVar$  and  $pv \in sol$ 
4:      $curV = nv$  with the largest no. of constraints on  $pv$  //most constraints
5:     if there are more than one  $nv$ 
6:        $curV = nv$  with min domain  $D(nv)$  //FF
7:       procedure CheckBack( $curV, sol, costResult, C^a$ ); //Algorithm 5.3
8:       if  $sol == \{\}$  return  $partialSol = \{\}$ ; //no solution
9:     else //no constraint with past variables
10:       $curV = nv$  with min domain  $D(nv)$  //FF
11:    end if
12:     $fv = \text{procedure FC}(curV, unassignedVar, partialSol, costResult, C^a)$ 
      //Algorithm 5.5
13:    if  $D(curV) == \emptyset$ 
14:      procedure BJ( $curV, fv, partialSol, costResult$ ); //Algorithm 5.7
15:    end while
16: return  $partialSol, costResult$ 

```

complexity as FC-BJ with FF (page 120), as in this case, the variable ordering in FC-BJ with the hybrid heuristic is also Fail-First. When there are such constraints, FC-BJ with the hybrid heuristic has better efficiency than FC-BJ with FF heuristic, as it first handles variables that may lead to more backtracking, and therefore reduces the total amount of backtracking. Although FC-BJ with the hybrid heuristic can

Table 5.1: The activation of V^a and C^a in the FC-BJ+FF searching process.

Activation	New V^a (Triggered by constraints)	New C^a
(1)	mPC (Initial variable); cFinalPC; V_{CT} of mPC (CA_1)	$CS_1; CA_1;$ CR_{13}
(2)	mAcces, mHost, mMonitor (CS_1); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_2 - CA_4)	$CS_2, CS_3; CC_1;$ $CC_2; CA_2$ - $CA_4;$ $CP_6; CR_1$ - CR_3
(3)	mMouse, mKeyBd (CS_3); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_{13}, CA_{14})	$CA_{13}, CA_{14}; CE_9;$ $CC_4; CC_{14}; CC_{15};$ CM_1 - $CM_3;$ CR_{17} - CR_{19}
(4)	mChassis, mMotherBd, mCPU, mHardDisk, mMemory, mDVD, mGraphCd (CS_2, CR_{13}); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_5 - CA_{11})	CA_5 - $CA_{11}; CE_1$ - $CE_8; CC_5$ - $CC_{12};$ $CC_3; CM_4$ - $CM_{12};$ $CP_3; CR_4$ - CR_{16}

help to relieve the backtracking issue in algorithm FC-BJ with FF on the extended DCSP-based product configuration and cost estimation system, it still provides poor efficiency when a lot of constraints between variables activated in different phases in the solving process exist.

5.3 Experiments

In the last chapter, we modeled the extended DCSP-based PC system (Section 4.3.1) and implemented the augmented BT solving process on the system (Section 4.3.2). In this section, we implement the augmented FC-BJ with FF heuristic (Section 5.1) and with the hybrid heuristic (Section 5.2) on this model. Experimental results are compared and discussed.

5.3.1 DCSP-Based PC Configuration Using FC-BJ with FF

In this section, the FC-BJ algorithm with FF heuristic proposed in Section 5.1 is implemented on the DCSP-based PC system modeled in Section 4.3.1 (page 94).

The use of the Fail-First (FF) heuristic requires the variable ordering to be dynamic. In the searching process, a variable with the smallest domain size is selected as the current variable to be assigned. If there are more than one variables with the

same smallest domain size, the selection is performed according to the lexicographic ordering. In the DCSP-based PC system, variable activation ordering in the solving process of FC-BJ with FF, listed in Table 5.1, is different from that in the augmented Backtracking solving process described in Table 4.2 (page 102). The first two activation steps are the same. However, in the FC-BJ with FF process, after “Activation (2)”, both variable $mAcces$ and $mHost$ have the smallest domain size 1, and according to the lexicographic ordering, $mAcces$ is selected to be assigned firstly. Then in “Activation (3)”, the child modules of $mAcces$ are activated, and the child modules of $mHost$ are activated into DCSP in “Activation (4)”.

Table 5.2 lists the steps of the FC-BJ with FF solving process on the DCSP-based PC system, in which each of the variable is selected as the current variable cv to be assigned. In column 3, the current domain of cv refers to its remaining values after forward checks (FC) performed by previous assigned variables; i.e., the domain of cv after the (temporary) removal of values inconsistent with the current partial solution. The conflict set of the current variable $conf(cv)$ (column 4) involves the conflicting variables of cv to which the solving process backjumps when cv 's domain becomes empty and backtracking occurs. After an assignment of cv , FC is performed on future variables that have direct connections with cv . Constraints checked in FC are listed in column 9 (“Constraints in FC”), and if some variables' values are removed due to the assignment, cv is added to the conflict set of the variables $conf(x)$ listed in the last column (“Added to $conf(x)$ ”).

For instance, in Step 10, $mMotherBd$ is selected as the current variable for its minimum domain size cut by FC performed by $mChassis$ and $mGraphCd$, and these two variables are added into $conf(mMotherBd)$. After the assignment $\langle mMotherBd, cMB_2 \rangle$, FC is performed on $mMotherBd$'s direct future variables $mCPU$ and $mMemory$, and some of their values are (temporarily) removed from their corresponding domains. Then $mMotherBd$ is involved into $conf(mCPU)$ and $conf(mMemory)$. However, in Step 11, although $mCPU$ performs forward check on $mMemory$ using constraint

Table 5.2: FC-BJ with FF searching process for the example case.

Step	Current Variable (cv)	Current Domain	conf(cv)	Node Consistency	Current Domain	Cost & Prefer. Constr.	Selected Comp.	Constraints in FC	Added to conf(x)
1	mPC	cFinalPC		CR_{13}	cFinalPC	CC_1	cFinalPC		
2	mAcces	cFinalAcs			cFinalAcs	CC_4	cFinalAcs		
3	mHost	cFinalHost			cFinalMF	CC_3, CP_3	cFinalMF		
4	mMonitor	$cM_1 - cM_8$		$CR_1 - CR_3$	cM_8	CC_2, CP_6	cM₈		
5	mMouse	$cMS_1 - cMS_9$		CR_{17}, CR_{18}	cMS_2, cMS_6	CC_{14}, CP_6	cMS₂	CE_9, CM_2, CM_3	KeyBd
6	mKeyBd	cKB_5	mMouse	CR_{19}	cKB_5	CC_{15}, CP_6	cKB₅		
7	mChassis	$cCH_1 - cCH_5$	mMonitor	CR_4, CR_5	cCH_3, cCH_5	CC_5, CP_6	cCH₅	CE_1, CM_{12}	mMotherBd
8	mGraphCd	$cGc_1 - cGc_5$		CR_{14}	cGC_1, cGC_2	CC_{12}, CP_6	cGC₁	CE_8, CM_{14}	mMotherBd
9	mDVD	$cDVD_1 - cDVD_6$		CR_{15}, CR_{16}	$cDVD_3$	CC_{11}, CP_6	cDVD₃		
10	mMotherBd	$cMB_2, cMB_5, cMB_6, cMB_7$	mChassis, mGraphCd	CR_6, CR_7	cMB_2	CC_6, CP_6	cMB₂	$CE_2 - CE_5, CE_7, CM_7, CM_{13}, CM_{15}$	mCPU, mMemory
11	mCPU	$cCPU_1, cCPU_4 - cCPU_6$	mMotherBd	CR_8	$cCPU_4, cCPU_5, cCPU_6$	CC_7, CP_6	cCPU₄	CE_4	
12	mMemory	$cMM_1, cMM_2, cMM_4, cMM_7, cMM_8, cMM_9$	mMotherBd	CR_9, CR_{10}, CE_6	cMM_7	CC_8, CC_9, CP_6	cMM₇		
13	mHardDisk	$cHD_1 - cHD_{10}$		CR_{11}, CR_{12}	cHD_2	CC_{10}, CP_6	cHD₂

Figure 5.2: A FC-BJ with FF solving process for the example PC configuration.

CE_4 , $mCPU$ is not added into $conf(mMemory)$ as there is no value of $mMemory$ removed by the assignment of $mCPU$.

In Table 5.2, the solving process of FC-BJ with FF for the example DCSP-based PC system (modeled in Section 4.3.1) is backtrack-free. In the rest of this section, we change requirement CR_{10} to $mMemory.pTotalCapac = 16G$ that leads to back-

Table 5.3: FC-BJ with FF searching process for the example case with new requirement (partial).

	Current Variable	Current Domain	conf(cv)	Node Consistency	Left Domain	Cost & Prefer.	Selected Comp.	Constraints in FC	Added to conf(x)
7	mChassis	cCH_1-cCH_5	mMonitor	CR_4, CR_5	cCH_3, cCH_5	CC_5, CP_6	cCH₅	CE_1, CM_{12}	mMotherBd
8	mGraphCd	cGc_1-cGc_5		CR_{14}	cGC_1, cGC_2	CC_{12}, CP_6	cGC₁	CE_8, CM_{14}	mMotherBd
9	mDVD	$cDVD_1 - cDVD_6$		CR_{15}, CR_{16}	$cDVD_3$	CC_{11}, CP_6	cDVD₃		
10	mMotherBd	$cMB_2, cMB_5, cMB_6, cMB_7$	mChassis, mGraphCd	CR_6, CR_7	cMB_2	CC_6, CP_6	cMB₂ <i>/BJ</i>	$CE_2-CE_5, CE_7, CM_7, CM_{13}, CM_{15}$	mCPU, mMemory
11	mMotherBd		mChassis mGraphCd				<i>BJ</i>		
12	mGraphCd	cGc_2, cGc_3	mChassis	CR_{14}	cGC_2	CC_{12}, CP_6	cGC₂	CE_8, CM_{14}	mMotherBd
13	mMotherBd	cMB_3, cMB_4	mChassis, mGraphCd	CR_6, CR_7	cMB_3, cMB_4	CC_6, CP_6	cMB₃	$CE_2-CE_5, CE_7, CM_7, CM_{13}, CM_{15}$	mCPU, mMemory
14	mCPU	$cCPU_2, cCPU_3$	mMotherBd	CR_8	$cCPU_2, cCPU_3$	CC_7, CP_6	cCPU₃	CE_4	
15	mMemory	cMM_3, cMM_5, cMM_6	mMotherBd	CE_6, CR_9, CR_{10}	cMM_6	CC_8, CC_9, CP_6	cMM₆		
16	mDVD	$cDVD_1 - cDVD_4$		CR_{15}, CR_{16}	$cDVD_3$	CC_{11}, CP_6	cDVD₃		
17	mHardDk	cHD_1-cHD_{10}		CR_{11}, CR_{12}	cHD_2	CC_{10}, CP_6	cHD₂		

BJ: Backjumping.

jumping in a solving process (shown in Table 5.3), where the first nine steps are the same as those in Table 5.2. In Step 10, assignment $\langle mMotherBd, cMB_2 \rangle$ removes some values of the future variable $mMemory$, and $mMotherBd$ is added into $conf(mMemory)$. However, the assignment results in an empty domain of $mMemory$ (denoted by underlined conflict set $mMemory$ in the last column of Table 5.3). Hence, the solving process backjumps to the last conflict variable in $conf(mMemory)$; i.e., $mMotherBd$ (Step 11), and discards its last assignment cMB_2 . $mMotherBd$ is removed from $conf(mMemory)$ and $conf(mCPU)$. Variable $conf(mMotherBd)$ is updated to $conf(mMotherBd) \cup conf(mMemory) \setminus \{mMotherBd\} = \{mChassis, mGraphCd\}$. Consequently, $mMotherBd$ is re-assigned. However, its current domain becomes empty after the removal of cMB_2 , then the solving process backjumps to the last conflict variable in $conf(mMotherBd)$, $mGraphCd$, instead of the last assigned variable $mDVD$ (Step 9) to which BT algorithm is backwards. In Step 12, $conf(mGraphCd)$ is reset from $\{\}$ to $\{mChassis\}$ ($\{\} \cup conf(mMotherBd) \setminus \{mGraphCd\}$). Its last assignment cGC_1 is discarded from its current domain, and $mGraphCd$ is removed from $conf(mMotherBd)$. The values of $mMotherBd$ removed by the assignment $\langle mGraphCd, cGC_1 \rangle$ are also restored. Subsequently, an available remaining value is instantiated to $mGraphCd$, and the solving process continues to reach the final solution. The solving process in Table 5.3 is also illustrated in Figure 5.2.

Note that variable ordering in the solving process is dynamic. For example, in Table 5.3, after the assignment of $mGraphCd$ in Step 8, both $mDVD$ and $mMotherBd$ have the same smallest domain size 4. According to lexicographic ordering, $mDVD$ is assigned before $mMotherBd$. However, when $mGraphCd$ is re-assigned in Step 12, the domain size of $mMotherBd$ becomes 2, which is smaller than that of $mDVD$. It is selected to be assigned. After $mMotherBd$ implements forward checking, the domain size of $mCPU$ and $mMemory$ is cut to 2 and 3 respectively. Hence, $mDVD$ that has 4 values in its domain is assigned after $mCPU$ and $mMemory$.

5.3.2 DCSP-Based PC Configuration Using Algorithm FC-BJ with Hybrid Heuristic

In this section, FC-BJ algorithm with the hybrid heuristic proposed in Section 5.2 is implemented on the example of DCSP-based PC system modeled in Section 4.3.1 (page 94).

For a better explanation of this algorithm, the DCSP-based PC system is extended by adding more constraints and requirements as follows:

- The color of “mouse” and “monitor” should be similar; that is one cannot be white and another black;

$$CE_{12} : mMouse.pColor \approx mMonitor.pColor;$$

- The color of “keyboard” and “monitor” should be similar.

$$CE_{13} : mKeyBd.pColor \approx mMonitor.pColor;$$

- Similarly, the color of “speaker” and “monitor” should be similar.

$$CE_{14} : mSpeaker.pColor \approx mMonitor.pColor.$$

- The resolution of “monitor” is larger than that of “graphics card”.

$$CE_{15} : mMonitor.pResolution \geq mGraphCD.pResolution.$$

- $CR_{20} : mWebcam. = true;$

- $CR_{21} : mWebcamBrand = Logitech;$

- $CR_{22} : mSpeaker = true;$

- $CR_{23} : mSpeaker.pPower = External;$

As defined in Section 4.1, in the searching process of the extended DCSP-based product configuration and cost estimation system, a new stable phase is formed when new module variables, together with instance components in their domain, and associated property and cost variables, are activated into the DCSP system. The variable

Table 5.4: The activation of V^a and C^a in the FC-BJ+hybrid searching process.

Activa- tion	New V^a (Triggered by constraints)	New C^a
(1)	mPC (Initial variable); cFinalPC; V_{CT} of mPC (CA_1)	$CS_1; CA_1; CR_{13},$ CR_{20}, CR_{22}
(2)	mAcces, mHost, mMonitor (CS_1); V_{CM} in their domain; all V_P & V_{CT} of new active modules (CA_2 - CA_4)	$CS_2, CS_3; CC_1;$ $CC_2; CA_2$ - $CA_4;$ $CP_6; CR_1$ - CR_3
(3)	mMouse, mKeyBd, mSpeaker, mWebcam (CS_3, CR_{20}, CR_{22}); V_{CM} in their domain; all V_P & V_{CT} of new active modules(CA_{13} - CA_{16})	CA_{13} - $CA_{16};$ CM_1 - $CM_3; CE_9$ - CE_{14} $CC_4; CC_{14}$ - $CC_{17};$ CR_{17} - $CR_{23}; CP_4, CP_5$
(4)	mChassis, mMotherBd, mCPU, mHardDisk, mMemory, mDVD, mGraphCd (CS_2, CR_{13}); V_{CM} in their domain; all V_P & V_{CT} of new active modules(CA_5 - CA_{11})	CA_5 - $CA_{11}; CE_1$ - CE_8 $CE_{15}; CC_5$ - $CC_{12};$ $CC_3; CM_4$ - $CM_{12};$ $CP_3; CR_4$ - CR_{16}

activation ordering in the solving process of FC-BJ with hybrid heuristic is listed in Table 5.4. Although the activation order of variables is the same as that in FC-BJ with FF (Table 5.1, page 127), the reason is different. In Activation (3) of Table 5.4, the child modules of module $mAccess$ are activated before those of $mHost$, because they have the largest total number 8 of constraints on assigned variables that are activated in Activation (1) and (2), while in algorithm FC-BJ with FF (Table 5.1, page 127), the activation order is based on the assignment order of their parent module variables, that is, $mAccess$ is assigned earlier than $mHost$ according to the lexicographic ordering as both of them have the smallest domain size.

To explain the efficiency of algorithm FC-BJ with the hybrid heuristic, the solving processes of the DCSP-based PC system with new constraints and requirements, using both FC-BJ with FF and FC-BJ with hybrid heuristic (denoted as FC-BJ-FF, FC-BJ-hybrid respectively), are compared. A part of the solving processes are listed in Table 5.5 and Table 5.6 respectively. In both solving processes, when a new active variable is assigned, the solving process first checks constraints on past variables activated in previous phases, and then, if there are still value(s) available, checks for its direct future variables. In column “Constraints in FC/checkback” of each table,

Table 5.5: FC-BJ-FF searching process for the augmented example case (partial).

Step	Current Variable (cv)	Current Domain	conf(cv)	Node Consistency	Current Domain	Cost & Prefer. Constr.	Selected Comp.	Constraints in FC/ checkback	Added to conf(x)
1	mPC	cFinalPC		CR_{13}	cFinalPC	CC_1	cFinalPC		
2	mAcces	cFinalAcs			cFinalAcs	CC_4	cFinalAcs		
3	mHost	cFinalHost			cFinalHost	CC_3, CP_3	cFinalMF		
4	mMonitor	cM_1-cM_8		CR_1-CR_2	cM_3, cM_8	CC_2, CP_6	cM_3		
5	<i>checkback</i>	cMS_1-cMS_9	mMonitor		cMS_3, cMS_6			CE_{12}	
	mMouse	cMS_3, cMS_6	mMonitor	CR_{17}, CR_{18}	cMS_6	CC_{14}, CP_6	cMS_6	CE_9, CM_2, CM_3	mKeyBd
6	<i>checkback</i>	cKB_1, cKB_7	mMouse		cKB_1, cKB_7			CE_{13}	
	mKeyBd	cKB_1, cKB_7	mMouse	CR_{19}	cKB_7	CC_{15}, CP_6	cKB_7		
7	<i>checkback</i>	cSP_1-cSP_{11}	mMonitor		cSP_3, cSP_9			CE_{14}	
	mSpeaker	cSP_3, cSP_9	mMonitor	CR_{22}, CR_{23}	cSP_3, cSP_9	CC_{17}, CP_5	cSP_3		
8	<i>checkback</i>	cWC_1-cWC_{11}	mMonitor		cWC_1, cWC_4			CE_{10}	
	mWebcam	cWC_1, cWC_4	mMonitor		cWC_5, cWC_6			CE_{11}	
9	mMonitor	cM_8	mMonitor	CR_{20}, CR_{21}	$\{\}$		Backjumping		
10	<i>checkback</i>	cMS_1-cMS_9	mMonitor		cMS_2, cMS_3		cM₈		
	mMouse	cMS_2, cMS_3	mMonitor	CR_{17}, CR_{18}	cMS_6, cMS_8			CE_{12}	
11	<i>checkback</i>	cKB_5	mMouse		cMS_2, cMS_6	CC_{14}, CP_6	cMS₂	CE_9, CM_2, CM_3	mKeyBd
	mKeyBd	cKB_5	mMouse	CR_{19}	cKB_5	CC_{15}, CP_6	cKB₅	CE_{13}	
12	<i>checkback</i>	cSP_1-cSP_{10}	mMonitor		cSP_1, cSP_6			CE_{14}	
	mSpeaker	cSP_1, cSP_6	mMonitor	CR_{22}, CR_{23}	cSP_6, cSP_9	CC_{17}, CP_5	cSP₉		
13	<i>checkback</i>	cWC_1-cWC_{11}	mMonitor		cWC_2, cWC_3			CE_{10}	
	mWebcam	cWC_2, cWC_3	mMonitor	CR_{20}, CR_{21}	cWC_6	CC_{16}, CP_4	cWC₂	CE_{11}	
14	...		mMonitor						

constraints for back checking and forward checking are listed respectively for different operations. For example, in Table 5.5, *mMouse* is selected as the current variable in Step 5. As there is a constraint between *mMouse* and a past variable *mMonitor*; i.e., CE_{12} listed in column “Constraints in FC/checkback”, the constraint must first be checked (first row in Step 5), and the domain of *mMouse* after back checking is stored in the sixth column “Current Domain”. If the current domain of *mMouse* is not empty, then FC-BJ is performed on it (second row in Step 5), and constraints checked in FC are also listed in column “Constraints in FC/checkback”. Otherwise, Backjumping occurs.

In these two processes, the first 4 steps are the same, in which after the activation of module variable *mAccess*, *mHost* and *mMonitor*, as there are no constraints between them and the previous active variable *mPC*, the order of them for assignment is based on FF heuristic, referring to Section 5.3.1. Subsequently, child modules of module variable *mAccess* are activated into the DCSP-based PC system. The following variable orders are different in these two algorithm solving processes.

Based on Fail-First heuristic, in Table 5.5, FC-BJ-FF instantiated these new active variables according to their domain sizes, that is, *mMouse*, *mkeyBd*, *mSpeaker*, and *mWebcam* are instantiated sequentially from Step 5 to Step 8. However, after back checking and requirement checking (i.e., node consistency) operations, the domain of *mWebcam* becomes empty (Step 8), and backjumping occurs. The solving process backjumps to *mWebcam*’s last conflict variable *mMonitor* (assigned in Step 4), and re-assigns it (Step 9) and all these four child module variables (Step 10 – 13).

In the solving process of algorithm FC-BJ-hybrid (Table 5.6), *mWebcam* (Step 5) is first checked as it is involved in the largest number of constraints on past variables, then the conflict is detected. In Step 6, FC-BJ-hybrid solving process backjumps to its last conflict variable *mMonitor* (assigned in Step 4), and re-assigned *mMonitor* (Step 6) and *mWebcam* (Step 7). Subsequently, as the other three child module variables are involved in the same number of constraints on past variables, they are instantiated from Step 8 to Step 10, according to their domain size.

Table 5.6: The searching process of FC-BJ-hybrid for the augmented example case (partial).

Step	Current Variable (cv)	Current Domain	conf(cv)	Node Consistency	Current Domain	Cost & Prefer. Constr.	Selected Comp.	Constraints in FC/checkbox	Added to conf(x)
1	mPC	cFinalPC		CR_{13}	cFinalPC	CC_1	cFinalPC		
2	mAccess	cFinalAcs			cFinalAcs	CC_4	cFinalAcs		
3	mHost	cFinalHost			cFinalHost	CC_3, CP_3	cFinalHost		
4	mMonitor	cM_1-cM_8		CR_1-CR_2	cM_3, cM_8	CC_2, CP_6	cM_3		
5	<i>checkboxack</i>	cWC_1-cWC_{11}	mMonitor		cWC_1, cWC_4 cWC_5, cWC_6			CE_{10} CE_{11}	
	mWebcam	cWC_1, cWC_4 cWC_5, cWC_6	mMonitor	CR_{20}, CR_{21}	$\{\}$		Backjumping		
6	mMonitor	cM_8					cM₈		
7	<i>checkboxack</i>	cWC_1-cWC_{11}	mMonitor		cWC_2, cWC_3 cWC_6			CE_{10} CE_{11}	
	mWebcam	cWC_2, cWC_3 cWC_6	mMonitor	CR_{20}, CR_{21}	cWC_2, cWC_6	CC_{16}, CP_4	cWC₂		
8	<i>checkboxack</i>	cMS_1-cMS_9	mMonitor		cMS_2, cMS_3 cMS_6, cMS_8			CE_{12}	
	mMouse	cMS_2, cMS_3 cMS_6, cMS_8	mMonitor	CR_{17}, CR_{18}	cMS_2, cMS_6	CC_{14}, CP_6	cMS₂	CE_9, CM_2, CM_3	mKeyBd
9	<i>checkboxack</i>	cKB_5	mMouse		cKB_5			CE_{13}	
	mKeyBd	cKB_5	mMouse	CR_{19}	cKB_5	CC_{15}, CP_6	cKB₅		
10	<i>checkboxack</i>	cSP_1-cSP_{10}	mMonitor		cSP_1, cSP_6 cSP_9, cSP_{10}			CE_{14}	
	mSpeaker	cSP_1, cSP_6 cSP_9, cSP_{10}	mMonitor	CR_{22}, CR_{23}	cSP_6, cSP_9 cSP_{10}	CC_{17}, CP_5	cSP₉		
11	...

Figure 5.3: Experimental results for generating a feasible solution.

From this simple comparison, it is observed that FC-BJ-hybrid can reduce the backtracking occurrence, comparing with FC-BJ-FF.

5.3.3 Experimental Results

To test the solving efficiency of the augmented FC-BJ with both FF heuristic and the hybrid heuristic implemented on the extended DCSP-based PC configuration with cost estimation system (modeled in Section 4.3.1), comparisons are conducted between algorithm FC-BJ with FF heuristic (denoted as “DCSP+FC-BJ-FF” in experimental results) and a classical CSP-based configuration system with standard backtracking algorithm (denoted as “CSP”); between “DCSP+FC-BJ-FF” and the augmented BT algorithm proposed in Section 4.1 (denoted as “DCSP+BT”); between “DCSP+FC-BJ-FF” and algorithm FC-BJ with the hybrid heuristic (denoted as “DCSP+FC-BJ-hybrid”); and among “DCSP+FC-BJ-FF”, “DCSP+FC-BJ-hybrid” and case-based configuration approach (denoted as “Case”). The experimental environment and the coordinate setting are the same as the experiment in the last chapter (page 106). Experimental results are illustrated in Figure 5.3 to Figure 5.9.

In experiments for generating the first feasible solution as shown in Figure 5.3, the augmented algorithm FC-BJ with FF applied to the DCSP-based PC system outperforms both the augmented BT algorithm and the classical CSP system, and has the best performance. In the performance curve, when the number of module

5.4.a: Results on the no. of visited node.

5.4.b: Results on CPU time.

Figure 5.4: Experimental results for generating all configuration solutions (“CSP” vs. “DCSP+FC-BJ-FF”).

variables ranges from 15 to 16, there is a relatively sharp increase in the number of visited nodes as when new variables are activated into the DCSP, there are new active constraints defined between assigned variables and new active variables that are not yet assigned. The consistency of these constraints needs to be checked before the performance of FC to guarantee that there is at least one value in the domain of new active variables consistent with the current partial solution. Otherwise, the solving process backtracks to the assigned variable in the constraint and re-assigns it and all variables after the assigned variable in the current partial solution must also be re-evaluated. This results in more work.

Figure 5.4 illustrates the experimental results implementing comparison between “CSP” and “DCSP+FC-BJ-FF” for generating all solutions with respect to the number of visited nodes in the searching process and CPU time respectively. The algorithm FC-BJ with FF heuristic on the DCSP-based PC system offers significant improvement compared with the CSP-based system, as the number of selected module variables, together with the number of components, property and cost variables and constraints increases. Similar experiments are conducted on the implementation of two algorithms; i.e., the augmented BT algorithm and the augmented FC-BJ with FF heuristic, on the DCSP-based PC system, and are illustrated in Figure 5.5. FC-

5.5.a: Results on the no. of visited node.

5.5.b: Results on CPU time.

Figure 5.5: Experimental results for generating all configuration solutions (“DCSP+BT” vs. “DCSP+FC-BJ-FF”).

Figure 5.6: Experimental results with cost optimization among “CSP” and “DCSP+BT” and “DCSP+FC-BJ-FF”.

BJ with FF heuristic also achieves better performance in terms of both aspects. It is observed that its outperformance becomes greater as the number of active variables increases, but the outperformance may be counteracted due to the increase of constraints between variables activated in different solving phases.

The performance of algorithm FC-BJ with FF on the extended DCSP-based PC system is further checked, taking into account cost optimization, such as minimum cost (denoted as “DCSP+FC+Cost”). First, its performance is compared with that of classical CSP system with Branch-and-Bound (B&B) optimization (page 37) (denoted as “CSP+BB”), and that of the augmented BT algorithm on the DCSP-based PC

Figure 5.7: Experimental results of FC-BJ-FF on DCSP for the first feasible result (with vs. without cost optimization).

system (denoted as “DCSP+BT+Cost”). Experimental results are illustrated in Figure 5.6. It is observed that FC-BJ with FF achieves the best performance although, compared with the augmented BT algorithm, counteraction of performance occurs when there are constraints between variables that are activated in different searching phases; that is, in Figure 5.6, for example, when the number of active module variables changes from 15 to 16.

The performance of the extended DCSP-based system integrating cost optimization is also compared with that of the extended DCSP-based system for yielding the first feasible solution. The comparison is conducted on FC-BJ with FF algorithm. As illustrated in Figure 5.7, the DCSP-based system with cost optimization shows a somewhat better performance for generating the first solution.

In addition, the performance of algorithm FC-BJ with FF heuristic is compared with that of FC-BJ with the hybrid heuristic (“DCSP+FC-BJ-hybrid”). Figure 5.8 illustrates experimental results in terms of both the total number of visited nodes in the searching process and the CPU time. When there is no constraint between variables activated in different searching phases, for example, when there are 12 and 13 active variables in the searching process as illustrated in Figure 5.8, these two solving algorithms are the same and thus, there is no difference in their performance. However, as constraints increase between variables activated in different searching

5.8.a: Results on the no. of visited node.

5.8.b: Results on CPU time.

Figure 5.8: Experimental results for generating all configuration solutions (“DCSP+FC-BJ-FF” vs. “DCSP+FC-BJ-hybrid”).

phases, algorithm FC-BJ with the hybrid heuristic achieves a better performance than FC-BJ with FF heuristic. Specifically, as active variables involved in the searching process change from 13 to 16 in Figure 5.8, the performance curve of algorithm FC-BJ with the hybrid heuristic presents a smaller curvature than that of FC-BJ with FF; i.e., FC-BJ with the hybrid heuristic visits less nodes and spends less CPU time in the searching process. Moreover, the performance improvement is monotonically increased, since the difference of the number of visited nodes and CPU time between these two algorithms becomes larger with the increase of constraints between variables activated in different searching phases.

Finally, comparisons among the performance of “DCSP+FC-BJ-FF”, “DCSP+FC-BJ-hybrid”, and “Case” are performed. Figure 5.9 illustrates the experimental results measured by CPU time in clock clicks. Both “DCSP+FC-BJ-FF” and “DCSP+FC-BJ-hybrid” exhibit better performance than “Case”, and may overcome the limitation of the case-based approach when new variables and new technologies are added to the configuration task (e.g., when the number of selected modules is 16). While the case-based approach does not include cost optimization, these two algorithms for DCSP achieve good performance on cost optimization (Figures 5.6 and 5.7). Detailed explanations can be found in Section 4.3.3 (pages 109 and 110).

Figure 5.9: Experimental results for generating all configuration solutions (“DCSP+FC-BJ-FF” vs. “DCSP+FC-BJ-hybrid” vs. “Case”)

Hence, it is observed that comparing with CSP-based system solved by classical BT algorithm and the case-based approach, FC-BJ with FF heuristic applied to the extended DCSP-based system significantly improves the searching efficiency of configuration tasks. It also generally outperforms the augmented BT algorithm, especially when multiple solutions are required, although it needs to handle more work when a new active variable is added into a constraint network. Furthermore, algorithm FC-BJ with the hybrid heuristic is proved to improve the performance of FC-BJ with FF by reducing the backtracking that may occur in the searching process when there are constraints between variables activated in different searching phases. In addition, these two proposed algorithms overcome a shortcoming of the case-based approach—application of new technologies and cost optimization.

5.4 Summary

Based on the augmented solving procedure for the extended DCSP-based configuration and cost estimation system presented in the last chapter, structural and activation constraint handlers are embedded into a classical CPS solving procedure, both of which are able to trigger the extension of the DCSP to form stable phases in the searching process. In this chapter, a hybrid solving algorithm for the DCSP-based

configuration with cost estimation system was proposed. The combination of algorithm forward checking and backjumping addresses issues of augmented Backtracking algorithm proposed in the last chapter (trashing and late conflict detection), and enhances the solving efficiency. In the forward checking-backjumping algorithm, as remaining values in the current domain of each future variable are consistent with the current partial solution, its size can be computed without any additional constraint checks. Thus, Fail-First heuristic can be performed conveniently, together with forward checking-backjumping, to improve the solving efficiency.

However, the augmented algorithm forward checking-backjumping with Fail-First heuristic applied to the extended DCSP-based system suffers limitation of a lot of constraint back checking when new active variables that have constraints with assigned variables are added into the DCSP. To address this limitation, a hybrid heuristic with algorithm forward checking-backjumping—a combination of an amended most-constrained heuristic with Fail-First heuristic—was proposed. The hybrid heuristic guarantees the activation of a set of new variables and the selection of current variable to be assigned are performed in a descending order of the number of constraints involved on assigned variables, since such variables are more likely to lead to backtracking.

Finally, procedures of both proposed algorithms for solving an extended DCSP-based desktop PC configuration and cost estimation task are elaborated in experiments. Experimental results of both algorithms are also compared and discussed.

Chapter 6

Experiments on Elevator System Configuration

In this chapter, the extended DCSP-based product configuration and cost estimation system (Chapter 4) and three proposed solving algorithms (Chapter 4 and Chapter 5) are implemented on a cable-operated elevator [216] configuration task, which is called “the extended DCSP-based elevator system” for short in this chapter. The architecture of the cable-operated elevator system is sketched in Figure 6.1, which visualizes major components comprising the elevator system.

Similar to the extended DCSP-based PC system (Figure 4.3, page 93), the configuration domain Φ of the extended DCSP-based elevator system, where the formalism of DCSP is generated from product family data and related design knowledge, is stored in a MSSQL database; customer requirements for an elevator are parsed into XML files; and a DCSP-based configuration engine deploys Java-based CSP solving library *Choco*, and is executed to generate valid configuration and cost estimation results.

This chapter is organized as follows: In Section 6.1, an elevator system configuration task is modeled as an extended DCSP-based configuration and cost estimation system. In Section 6.2, the augmented Backtracking solving algorithm (proposed in Section 4.1) is conducted on the extended DCSP-based elevator system to generate configuration and cost estimation results. In Section 6.3, the algorithm Forward Checking-Backjumping with Fail-First heuristic (proposed in Section 5.1) is implemented on the extended DCSP-based elevator system. Section 6.4 elaborates the im-

Figure 6.1: Vertical section of a cable-operated elevator system [216].

plementation of algorithm Forward Checking-Backjumping with the hybrid heuristic (proposed in Section 5.2) on the the extended DCSP-based elevator system. Experimental results are analyzed in Section 6.5. Finally, conclusions are made in Section 6.6.

6.1 Modeling DCSP-Based Elevator Configuration System from PDMs

As proposed in Chapter 4, the extended DCSP-based product configuration and cost estimation system is defined as $\langle V^I, D^I, C^I, \Phi \rangle$, in which the configuration domain $\Phi = \langle V_M, V_{CM}, V_P, V_{CT}, D, C_s, C_a, C_g \rangle$ stores all variables and constraints (i.e., active and inactive) in the DCSP, and is generated from various PDMs. In the extended DCSP-based system, all active variables involved in the final solutions and active constraints checked in the searching process are activated from Φ . In this section, the formalism deriving process of Φ is described.

Corresponding to Figure 6.1, Figure 6.2 illustrates a PFA of an elevator system

Figure 6.2: PFA of an elevator configuration system.

(due to the complexity of the elevator system, Figure 6.2 only shows only the first 5 levels of the PFA). The elevator system includes 45 modules, among which 6 underlined modules (car phone, car lantern, car intercommunication, car position indicator, counterweight guard, machine beam bearing plate) are optional modules, and the other 39 modules are mandatory. Based on module donation $Module : (\vec{IC}; \vec{A}; \vec{C}; \vec{CM})$ (defined in Section 3.5.4), formalisms of DCSP (i.e., module variables V_M , component variables V_{CM} , property variables V_P , and cost variables V_{CT}) are derived from PDMs. The size of the elevator dataset is large, and Table 6.1 lists a part of these formalisms and corresponding activation constraints, where elements of V_M , V_{CM} , V_P and V_{CT} are prefixed with m , c , p , and ct respectively.

In the extended DCSP-based elevator system, 45 activation constraints C_a corresponding to each module variable m can be derived from $\vec{A}$ and $\vec{C}$ of m , which represent the association between modules and their properties and cost drivers. As the number of property variables and cost variables is large, only a part of C_a are listed in the last column of Table 6.1. For example,

$$CA_{44}: mCarBuf \Rightarrow \{pModel, pHgt, PStroke, pBlkHgt, pFootHgt, PLoad, PLoadMin, PLoadMax, pClear, pQt, pHdWgt, pStkSpdMax, pStkSpdMin, ctCostA, ctCostU, ctAssm\}.$$

Corresponding to the elevator PFA shown in Figure 6.2, structural constraints C_s

Table 6.1: Variables and C_a in the extended DCSP-based elevator system (partial).

V_M (m)	V_{CM} (c) ($Domain(V_M)$)	V_P (p)	V_{CT} (t)	C_a $m \Rightarrow p/t$
mDoor	cSSSO c2SSO cSSCO c2SCO	pModel,pOpenType,pSpeed pOpenSide,pOprtWgt pEngWgt,pHdWgt,pSpace pOprtConst,pSTEdgeWgt pOpenStrik,pOpenWdt	ctDoor ctFace,ctOprt ctOprtEng ctOprtHd ctAssm	CA_8
mHCb	c(N)-D (where N:pNo. D:pDiam)	pModel,pLgt,pWdt,pNo., pLdCarCarB,pLdCarCarT, pLdCwtCarB,pLdCwtCarT pAbvCwtCarB,pTractRt, pTractRtBE,pTractRtBF, pTractRtTE,pSafetyMin,pQt pTractRtTF,pWgtU,pDiamt, pTractFactor,pStrgthMax	ctCostA ctCostU ctAssm	CA_{30}
mMchBm	cS10-25.4 cS10-35.0 cS12-31.8 cS12-35.0 cS12-40.8 cS12-50.0 cS15-42.9 cS15-50.0 cS18-54.7 cS18-70.0	pSectModulus,pPlateThick, pBendMtRMax,pWgtU pBendMtLMax,pWgt pLgt,pImpLdAll,pImpLdL pSafetyFac,pTotalLdL, pTotalLdR,pLdFL,pLdFR, pLdRL,pLdRR,pSptType pRctFL,pRctFR,pRctRL, pRctRR,pSpt2Ct,pSptDist pSptB2RoomT,pModel, pSptF2Hoist,pCt2ShvCt, pSpt2ShvCt,pSptB2RoomF	ctCostA ctCostU ctMBBear ctAssm	CA_{35}
mCarBuf	cOH-1 cOM-14	pModel,pHgt,pBlkHgt,pLoad pFootHgt,pStroke,pLoadMin pLoadMax,pClear,pQt pStkSpdMax,pStkSpdMin	ctCostA ctCostU ctAssm	CA_{44}

of the extended DCSP-based elevator system are generated from parent-child relationships between modules; i.e., $\overrightarrow{CM}$ of a module, and are listed below.

- CS_1 : $mElevator \Downarrow \{mHoist, mCar, mCwt, mSuspend, mCable\}$;
- CS_2 : $mHoist \Downarrow \{mOverHd, mTravel, mPit\}$;
- CS_3 : $mCar \Downarrow \{mCab, mSupport, mGov, mCarBuf\}$;
- CS_4 : $mCwt \Downarrow \{mCwtFrame, mCwtPlate, mCwtGdshoe, mCwtGdrail, mCwtU,$

$mCwtBuf, \sim mCwtGuard\}$, where $mCwtGuard$ is optional;

- CS_5 : $mSuspend \Downarrow \{mHCb, mMch, mMchBm\}$;
- CS_6 : $mCable \Downarrow \{mContCb, mCompCb\}$;
- CS_7 : $mCab \Downarrow \{mDoor, mCarFrame, \sim mPhone, \sim mLantern, \sim mInterCom, \sim mPosInd\}$, where the last four module variables with $\sim$ are optional;
- CS_8 : $mSupprot \Downarrow \{mPlatfm, mSling\}$;
- CS_9 : $mGov \Downarrow \{mGovCb, mGovLoop, mSafBm\}$;
- CS_{10} : $mSling \Downarrow \{mCrossHd, mStile, mCarGdshoe, mCarGdrail\}$;
- CS_{11} : $mMch \Downarrow \{mMotor, mShv, mDefShv, mSensor\}$;
- CS_{12} : $mMchBm \Downarrow \{\sim mMchBmBear\}$, where $mMchBmBear$ is optional.

As presented in Section 3.5.3, design knowledge provides background information for product configuration, and can be mapped into various generic constraints C_g in the DCSP (e.g., compatibility constraints CM_i , equality constraints CE_j , cost constraints CC_k , and preference constraints CP_l). As there are many configuration rules [216] generated from the design knowledge of elevator configuration, in the rest of this section, only a small subset of them are summarized. Corresponding compatibility constraints (CM_i), and equality/inequality constraints (CE_j) are generated. For certain constraints when violated, there is no fix for the violation; however for others, specific design modifications that may potentially fix the violation are implemented. Each fix is represented as a preference constraint on constraint (CP_k) with one of codes D1-D10 [216] to identify its desirability. The codes are listed below, and codes with lower numbers are more desirable.

D1: No problem.

D2: Increases maintenance requirements.

D3: Makes installation inconvenient.

D4: Changes minor equipment selection or sizing.

D5: Violates minor equipment constraint.

D6: Changes minor contract specifications.

D7: Requires special part design.

D8: Changes major equipment selection or sizing.

D9: Changes the building dimensions.

D10: Changes major contract specifications.

- (1) The car overtravel equals the distance from the hoistway top landing to underside machine beam minus the sum of the sling underbeam distance and the crosshead height. It must be at least $R+1.5S+24$ (R: the counterweight runby, S: the counterweight buffer stroke). 5 fixes are possible: Decrease the counterweight bottom reference by the amount the overtravel falls short of its minimum (D1). Decrease the sling underbeam space by the amount the overtravel falls short of its minimum (D3). Increase the counterweight buffer blocking height by steps of 1 inch (D4). Increase the hoistway overhead by the amount the overtravel falls short of its minimum (D9). Decrease the hoistway pit depth by steps of 1 inch (D9).

$$CE_8: mCar.pOverTrav = mHoist.pT2MchBmD - (mSling.pUB + mCrossHd.pHgt) \\ \geq mCwt.pRunby + 1.5 \times mCwtBuf.pStroke + 24 \text{ inches.}$$

$$CP_5: CE_8 = False \rightarrow mCwt.pBtmRef = \\ mCwt.pBtmRef - |mCar.pOverTrav - mCar.pOverTravMin|. \text{ (D1)}$$

$$CP_6: CE_8 = False \rightarrow mSling.pUB = \\ mSling.pUB - |mCar.pOverTrav - mCar.pOverTravMin|. \text{ (D3)}$$

$$CP_7: CE_8 = False \rightarrow mCwtBuf.pBlkHgtStep = 1 \text{ inch. (D4)}$$

$$CP_8: CE_8 = False \rightarrow mOverHd.pDpt = \\ mOverHd.pDpt + |mCar.pOverTrav - mCar.pOverTravMin|. \text{ (D9)}$$

$$CP_9: CE_8 = False \rightarrow mPit.pDptStep = -1 \text{ inch. (D9)}$$

- (2) The counterweight between guiderails distance must be at least 38 inches when there are 2 counterweight buffers. If it is not, upgrade the distance to the next

higher available distance (D4).

CM_7 : IF $mCwtBuf.pQt = 2$, THEN $mCwt.pBG \geq 38$ inches.

CP_{14} : $CM_7 = False \rightarrow mCwt.pBG = 54$ inches. (D4)

- (3) The counterweight runby must be between 6 and 36 inches, inclusive. For the maximum, 3 fixes are possible: decrease the counterweight bottom reference by the amount the runby exceeds its maximum (D1), or increase the counterweight buffer blocking height by steps of one inch (D4), or decrease the hoistway pit depth by steps of one inch (D9).

CE_{21} : $6 \text{ inches} \leq mCwt.pRunby \leq 36 \text{ inches}$.

CP_{22} : $mCwt.pRunby > 36 \text{ inches} \rightarrow$

$$mCwt.pBmRef = mCwt.pBmRef - (mCwt.pRunby - 36). \text{ (D1)}$$

CP_{23} : $mCwt.pRunby > 36 \text{ inches} \rightarrow mCwtBuf.pBlkHgtStep = 1 \text{ inch}$. (D4)

CP_{24} : $mCwt.pRunby > 36 \text{ inches} \rightarrow mPit.pDptStep = -1 \text{ inch}$. (D9)

- (4) Machine model 18 is compatible with motor model 10 HP and 15 HP. Machine model 28 is compatible with motor models 15 HP and 20 HP. Machine model 38 is compatible with motor model 20 HP, 25 HP, 30 HP, and 40 HP. Machine model 58 is compatible with motor model 40 HP. If machine and motor are incompatible, upgrade the machine model if the machine model is 18, or if the machine model is 28 and the motor model is 25 HP, 30 HP, or 40 HP (D8); upgrade the motor model if the machine model is 28 or 38, or the machine model is 58 and the motor model is 10 HP (D8).

CM_{29} : IF $mMch.pModel = 18$, THEN $mMotor.pModel \in (10HP, 15HP)$.

CM_{30} : IF $mMch.pModel = 28$, THEN $mMotor.pModel \in (15HP, 20HP)$.

CM_{31} : IF $mMch.pModel = 38$, THEN $mMotor.pModel \in (20HP, 25HP, 30HP, 40HP)$.

CM_{32} : IF $mMch.pModel = 58$, THEN $mMotor.pModel = 40HP$.

CP_{40} : $CM_{29}=False$ or $(mMch.pModel=28 \ \& \ mMch.pModel \in (25HP, 30HP, 40HP))$
 $\rightarrow mMch.pModelUpgd = True$. (D8)

CP_{41} : ($mMch.pModel = 28$ & $mMotor.pModel = 10HP$) or

$$CM_{31} = False \text{ or } CM_{32} = False \rightarrow mMotor.pModelUpd = True. \quad (D8)$$

- (5) The opening width is at most 72 inches. If platform model is 2.58, the minimum opening width is half the platform width minus 12 inches for center-opening doors, and half the platform width minus 6 inches for side-opening doors. If it is violated, increase opening width by the amount by which it falls short of its minimum (D1).

CE_{34} : $mHoist.pOpenWdt \leq 72 \text{ inches}$.

CM_{33} : IF $mPlatfm.pModle = 2.58$ & $mDoor.pOpenType = center-opening$,

THEN $mHoist.pOpenWdtMin = 1/2 \text{ } mPlatfm.pWdt - 12 \text{ inches}$.

CM_{34} : IF $mPlatfm.pModle = 2.58$ & $mDoor.pOpenType = side-opening$,

THEN $mHoist.pOpenWdtMin = 1/2 \text{ } mPlatfm.pWdt - 6 \text{ inches}$.

CP_{44} : $mHoist.pOpenWdt < mHoist.pOpenWdtMin \rightarrow$

$$mHoist.pOpenWdt = mHoist.pOpenWdtMin. (D1)$$

- (6) The hoist cable traction ratio is computed separately under 4 conditions: when the car is at the top and full, at the top and empty, at the bottom and full, and at the bottom and empty. 4 formulas define the 4 ratios, where the numerator must be at least as large as the denominator ($T_{side,position}$: the total load on the *side* of the sheave when the car is in the *position*; e.g., $T_{cwt,top}$: cable load total countweight side car top, F: the hoist cable traction factor). The overall hoist cable traction ratio is the maximum of these 4 ratios. It is at most $0.00788Q+0.675$ when the machine groove model is K3140, and $0.006555Q+0.755$ when the model is K3269 (Q: the machine angle of contact of the hoist cable). There are 4 fixes: Decrease the distance counterweight to platform rear by half-inch steps (D3); Increase the car supplement weight by 100 pound steps (D4); Upgrade the compensation cable to the next higher model or, if no compensation cable is currently in use, use one with the smallest unit weight (D6); Upgrade the machine groove model (D8).

$$CE_{42} : mHCb.pTractFactor F = \frac{32.2 + mMch.pAccMax}{32.2 - mMch.pAccMax}.$$

$$CE_{43} : mHCb.pTractRtBE = \frac{T_{cwt,bottom}}{T_{car,bottom} - mCar.pCapac} \cdot F \geq 1.$$

$$CE_{44} : mHCb.pTractRtBF = \frac{T_{car,bottom}}{T_{cwt,bottom}} \cdot F \geq 1.$$

$$CE_{45} : mHCb.pTractRtTE = \frac{T_{cwt,top}}{T_{car,top} - mCar.pCapac} \cdot F \geq 1.$$

$$CE_{46} : mHCb.pTractRtTF = \frac{T_{car,top}}{T_{cwt,top}} \cdot F \geq 1.$$

$$CE_{47} : mHCb.pTractRt = \max(mHCb.pTractRtBE, mHCb.pTractRtBF, \\ mHCb.pTractRtTE, mHCb.pTractRtTF).$$

$$CM_{46} : \text{IF } mMch.pGrvModel = K3140,$$

$$\text{THEN } mHCb.pTractRt \leq 0.00788 \cdot mMch.pContactAngle + 0.675.$$

$$CM_{47} : \text{IF } mMch.pGrvModel = K3269,$$

$$\text{THEN } mHCb.pTractRt \leq 0.006555 \cdot mMch.pContactAngle + 0.755.$$

$$CP_{54} : CM_{46} = \text{False} \text{ or } CM_{47} = \text{False} \rightarrow mCwt.p2PlatfmRStep = -1/2 \text{ inches. (D3)}$$

$$CP_{55} : CM_{46} = \text{False} \text{ or } CM_{47} = \text{False} \rightarrow mCar.pSupWgtStep = 100 \text{ pounds. (D4)}$$

$$CP_{56} : CM_{46} = \text{False} \text{ or } CM_{47} = \text{False} \rightarrow$$

$$\text{IF } mCompCb.pQt \neq 0, \text{ THEN } mCompCb.pModelUpGd = \text{True}.$$

$$\text{ELSE } mCompCb.pModel = 3/16\text{-Chain \& } mCompCb.pQt = 1. \text{ (D6)}$$

$$CP_{57} : CM_{46} = \text{False} \text{ or } CM_{47} = \text{False} \rightarrow mMch.pGrvModelUpgd = \text{True. (D8)}$$

- (7) The maximum vertical rail force of car guiderrails depends on the rail unit weight and the hoistway bracket spacing (denoted by B). The actual car guiderail vertical rail force equals the sum of car capacity, car weight, compensation cable load car side car top and control cable load car side car top. The rail's maximum force must be at least as great as the actual car guiderail vertical rail force. If it is not, upgrade the car guiderail unit weight to the next higher available value (D4).

$$CM_{48} : \text{IF } mCarGdrail.pWgtU = 8,$$

$$\text{THEN IF } B \leq 72, \text{ THEN } mCarGdrail.pVForceMax = 5500,$$

$$\text{ELSE } mCarGdrail.pVForceMax = 7750 - 31.25B.$$

$$CM_{49} : \text{IF } mCarGdrail.pWgtU = 11,$$

THEN IF $B \leq 126$, THEN $mCarGdrail.pVForceMax = 10000$,

ELSE $mCarGdrail.pVForceMax = 16000 - 47.619B$.

CM_{50} : IF $mCarGdrail.pWgtU = 16$,

THEN IF $B \leq 120$, THEN $mCarGdrail.pVForceMax = 15000$,

ELSE $mCarGdrail.pVForceMax = 22500 - 62.5B$.

CM_{51} : IF $mCarGdrail.pWgtU = 18.5$,

THEN IF $B \leq 108$, THEN $mCarGdrail.pVForceMax = 21000$,

ELSE $mCarGdrail.pVForceMax = 26400 - 50B$.

CM_{52} : IF $mCarGdrail.pWgtU = 32$,

THEN IF $B \leq 126$, THEN $mCarGdrail.pVForceMax = 41000$,

ELSE $mCarGdrail.pVForceMax = 45500 - 35.714B$.

CE_{49} : $mCarGdrail.pVForce = mCar.pCapac + mCar.pWgt + mCompCb.pLdCarCarTop$
 $+ mContCb.pLdCarCarTop \leq mCarGdrail.pVForceMax$.

CP_{59} : $CE_{49} = False \rightarrow mCarGdrail.pWgtUUpd = True$. (D4)

- (8) The machine beam maximum load is defined separately for 4 contact points of the beam (front (F) and rear (R), left (L) and right (R)), where O_L and O_R : the left and right machine beam offsets, respectively ($mMch.pOffsetL$, $mMch.pOffsetR$), D_1 : the distance from the machine beam support to machine beam center ($mMchBm.pSpr2Ct$), D_2 : the horizontal distance from the machine sheave center to front machine beam support ($mMchBm.pSpt2ShvCt$), and D_3 : the machine beam support distance ($mMchBm.pSptDist$). The maximum building loads at the 4 corners are equal to the machine beam reaction at the same corner times the machine beam safety factor. In donations $mMchBm.pLD\{FR, LR\}$ and $mMchBm.pRct\{FR, LR\}$, “FR” represents front or rear corner, “LR” represents left or right corner; e.g., $mMchBm.pLdFL$ refers to machine beam load front left.

CE_{92} : $mMchBm.pImpLdAll = mMch.pWgtA + mDefShv.pWgt + 2 \times mMch.pSptLd$.

CE_{93} : $mMchBm.pImpLdL = mMchBm.pImpLdAll \cdot \frac{O_R}{O_L + O_R}$.

CE_{94} : $mMchBm.pImpLdR = mMchBm.pImpLdAll \cdot \frac{O_L}{O_L + O_R}$.

$$CE_{95}: mMchBm.pTotalLdL = mMchBm.pWgt + mMchBm.pImpLdL.$$

$$CE_{96}: mMchBm.pTotalLdR = mMchBm.pWgt + mMchBm.pImpLdR.$$

$$CE_{97}: mMchBm.pRctRL = \frac{D_1 \cdot mMchBm.pWgt + D_2 \cdot mMchBm.pTotalLdL}{D_3}.$$

$$CE_{98}: mMchBm.pRctRR = \frac{D_1 \cdot mMchBm.pWgt + D_2 \cdot mMchBm.pTotalLdR}{D_3}.$$

$$CE_{99}: mMchBm.pRctFL = mMchBm.pTotalLdL - mMchBm.pRctRL.$$

$$CE_{100}: mMchBm.pRctFR = mMchBm.pTotalLdR - mMchBm.pRctRR.$$

$$CE_{101}: mMchBm.pLd\{FR, LR\} = mMchBm.pRct\{FR, LR\} \times mMchBm.pSaftyFactor.$$

- (9) The machine angle of contact refers to the angle subtended by the arc of hoist cable in contact with the machine sheave. It is defined by the following formulas:

$$CE_{104}: mShv.p2DefShvHorz =$$

$$mCar.pCbHitch2CwtCbHitch^{-1/2} mShv.pDiam^{-1/2} mDefShv.pDiam.$$

$$CE_{105}: mShv.p2DefShvDiag = +\sqrt{mShv.p2DefShvHorz^2 + mShv.p2DefShv^2}.$$

$$CE_{106}: mMch.pAngleContact =$$

$$90 + \sin^{-1} \frac{mShv.p2DefShv}{mShv.p2DefShvDiag} + \sin^{-1} \frac{1/2(mShv.pDiam - mDefShv.pDiam)}{mShv.p2DefShvDiag}.$$

- (10) The motor horsepower required is a function of car capacity, car speed, and the mechanical efficiency of the system as defined in the following formula, where the motor overall system efficiency equals the machine efficiency times the motor hoistway efficiency (which is estimated as 0.95).

$$CE_{107}: mMotor.pHpReq =$$

$$\frac{mCar.pCapac \times mCar.pSpeed(1 - mCwt.pPerc)}{33000} \times mMch.pEffc \times 0.95.$$

- (11) The platform weight is computed using a complex function of platform width (W) and platform depth (D), car capacity, door opening width (Q), door speed, and platform model. There are 4 platform weight factors AP , S , X , and Z .

$$CM_{76}: \text{IF } mPlatfm.pModle = 4B, \text{ THEN } AP = \left\lceil \frac{W-0.125}{9.5} \right\rceil.$$

ELSE IF $mPlatfm.pModle=2.5B$, THEN IF $W \leq 60$, THEN $AP=3$,
 ELSE IF $60 < W \leq 76$, THEN $AP=4$,
 ELSE IF $W > 76$, THEN $AP=5$.
 ELSE IF $mPlatfm.pModle=6B$, THEN IF $W \leq 67$, THEN $AP=5$,
 ELSE IF $67 < W \leq 76.5$, THEN $AP=6$,
 ELSE IF $76.5 < W \leq 86$, THEN $AP=7$,
 ELSE IF $86 < W \leq 96.5$, THEN $AP=8$,
 ELSE IF $96.5 < W \leq 105$, THEN $AP=9$,
 ELSE IF $W > 105$, THEN $AP=10$.

CM_{77} : IF $mPlatfm.pModle = 2.5B$, THEN

IF $mCar.pCapac = 2000 \text{ pounds} \ \& \ mPlatfm.pDpt \leq 53 \text{ inches}$,
 THEN $S = 63$, ELSE $S = 71$.

CM_{78} : IF $mPlatfm.pModle=4B$, THEN IF $mDoor.pSpeed=SS$, THEN $X=2.26$,
 ELSE IF $mDoor.pSpeed = 2S$, THEN $X = 2.354$.

ELSE IF $mPlatfm.pModle = 6B$,
 THEN IF $mDoor.pSpeed = SS$, THEN $X = 2.441$,
 ELSE IF $mDoor.pSpeed = 2S$, THEN $X = 2.394$.

CM_{79} : IF $mPlatfm.pModle = 2.5B$, THEN $mPlatfm.pWgt =$
 $S + 0.318(5.06W + 0.11WD) + (D - 7.6)(3.14 + 0.8AP)$,
 ELSE IF $mPlatfm.pModle = 4B$, THEN $mPlatfm.pWgt =$
 $35 + XW + 2.774D + 0.03WD + 0.226Q + AP(0.226D - Z)$,
 ELSE IF $mPlatfm.pModle = 6B$, THEN $mPlatfm.pWgt =$
 $35 + XW + 3.228D + 0.34Q + AP(0.226D - Z)$.

In addition, based on relationships among cost variables and property variables of modules, cost constraints (CC_l) can be defined from design knowledge as follows:

- (1) The total cost of miscellaneous car components is the sum of the cost of door operator, car guideshoes, car saf-t-edge, car limit switch cam, car cable hanger, car duct and wiring, car landing switch, and car maintenance station.

- CC_4 : $mCar.ctMisc = \sum(mDoor.ctOprrt, mCarGdshoe.ctCostA, mCar.ctLimSwtCam, mCar.ctSTEG, mCar.ctHanger, mCar.ctDuctWir, mCar.ctLandSwt, mCar.ctMaint)$.
- (2) The cost of the counterweight stack is the sum of the cost of counterweight plates.
- CC_6 : $mCwt.ctStack = mCwtPlate.ctCostU \times mCwtPlate.pQt$.
- (3) Initially the total cost of car (counterweight) buffers is the unit cost of a buffer. If car (counterweight) buffer quantity is more than one, the total cost of car (counterweight) buffers is buffer's unit cost multiplies the quantity of buffers.
- CC_{16} : $mCarBuf.ctCostA = mCarBuf.ctCostU$.
- CP_{60} : $mCarBuf.pQt > 1 \rightarrow mCarBuf.ctCostA = mCarBuf.ctCostU \times mCarBuf.pQt$.
- CC_{34} : $mCwtBuf.ctCostA = mCwtBuf.ctCostU$.
- CP_{61} : $mCwtBuf.pQt > 1 \rightarrow mCwtBuf.ctCostA = mCwtBuf.ctCostU \times mCwtBuf.pQt$.
- (4) The total cost of counterweight guideshoes (car guideshoes) are the sum of the cost of 4 guideshoes on the counterweight (on the car).
- CC_{20} : $mCwtGdshoe.ctCostA = 4 \times mCwtGdshoe.ctCostU$.
- CC_{63} : $mCarGdshoe.ctCostA = 4 \times mCarGdshoe.ctCostU$.
- (5) The cost of control cable and governor cable are calculated by the product of the unit cost (dollars/foot) and the length (inches) of corresponding cable.
- CC_{42} : $mContCb.ctCostA = mContCb.ctCostU \times mContCb.pLgt/12$.
- CC_{51} : $mGovCb.ctCostA = mGovCb.ctCostU \times mGovCb.pLgt/12$.
- (6) The cost of hoist cable and compensation cable equal the product of the unit cost (dollars/foot), the length (inches), and the quantity of corresponding cable.
- CC_{37} : $mHCb.ctCostA = mHCb.ctCostU \times mHCb.pLgt/12 \times mHCb.pQt$.
- CC_{44} : $mCompCb.ctCostA = mCompCb.ctCostU \times mCompCb.pLgt/12 \times mCompCb.pQt$.
- (7) The cost of the platform is its unit cost (dollars/pound) times its weight. Similar calculations are performed on the cost of the machine beam, and the safety beam.
- CC_{48} : $mPlatfm.ctCostA = mPlatfm.ctCostU \times mPlatfm.pWgt$.

$$CC_{54}: mSafBm.ctCostA = mSafBm.ctCostU \times mSafBm.pWgt.$$

$$CC_{40}: mMchBm.ctCostA = mMchBm.ctCostU \times mMchBm.pWgt.$$

(8) The unit cost $ctLeafUnit_j$ of each leaf component j in an elevator PFA is calculated as: $CC_l : ctLeafUnit_j = \sum_j [ctProcure_j, ctStorage_j, ctTransp_j]$, where

- Procurement cost $ctProcure_j = \sum [ctPrice_j, T_j^{manday} \times ctManday_j]$, where T_j^{manday} is the number of man-days required to procure component j , $ctManday_j$ is the man-day rate in dollar terms.
- Storage cost $ctStorage_j = f\{R_j^{fail}, ctP_j^{mass}, ctP_j^{small}, Qt_j^{st}, T_j^{st}, ctManday_j\}$, where R_j^{fail} is the failure rate of component j , ctP_j^{mass} is the mass procurement cost, ctP_j^{small} is the small-batch procurement cost, Qt_j^{st} is the units of component j stored, T_j^{st} is the average storage time of component j .
- Transportation cost $ctTransp_j = f\{T_j^{LT}, Qt_j^{trans}, ctT^{mass}, ctT^{small}\}$, where T_j^{LT} is the delivery lead-time, Qt_j^{trans} is the transportation quantity, ctT^{mass} and ctT^{small} are transportation cost for mass and small-batch procurement.

(9) The cost of a non-leaf component g in the elevator PFA (i.e., parent component, such as *Machine Unit* and *Car Assembly* in Figure 6.2) is the sum of the total cost of g 's mandatory constituents and corresponding assembly cost:

$$CC_k : ctParentCost_g = \sum ctChildCost_f + ctAssm_g, \text{ where}$$

- $\sum ctChildCost_f$ is the total cost of all mandatory child components of g ; e.g., $\sum mCab.ctCost = mDoor.ctCost + mCarFrame.ctCost$.
- $ctAssm_g = T_g^{asm} \times ctManday_g^{asm}$ is the assembly cost for g , where T_g^{asm} and $ctManday_g^{asm}$ are time and man-day rate for assembly respectively.

(10) The cost of the entire elevator is defined as: $CC_1 : mElevator.ctCost = (\sum ctChild_h + mElevator.ctAssm) \times mElevator.ctProfitRate$, where

- Assembly cost $ctAssm = T^{asm} \times ctManday^{asm}$, where T^{asm} and $ctManday^{asm}$ are time and man-day rate for elevator assembly respectively.

- $\sum ctChild_h$ is the total cost of all direct mandatory constituents of an elevator; i.e., $\sum ctChild_h = mHoist.ctCost + mCar.ctCost + mCwt.ctCost + mSuspend.ctCost + mCable.ctCost$.
- $ctProfitRate = f\{ctOverhead, ctAntiProfit\}$ is the profit rate of an elevator, which is calculated based on company's anticipant profit rate $ctAntiProfit$ and overhead costs $ctOverhead$, such as management costs, power, rental.

There are also 15 constraints of preference on constraint (page 66) corresponding to optional modules in the extended DCSP-based elevator system.

- (1) A car cab may include intercom, lantern, phone, and position indicator. Its weight includes the weight of fixtures that the customer requests. Similarly, its total cost includes the cost of requested fixtures.

$$CP_{62}: mPhone.pRequ = True \rightarrow mCab.pWgt = mCab.pWgt + mPhone.pWgt.$$

$$CP_{63}: mLantern.pReq = True \rightarrow mCab.pWgt = mCab.pWgt + mLantern.pWgt.$$

$$CP_{64}: mIntercom.pReq = True \rightarrow mCab.pWgt = mCab.pWgt + mIntercom.pWgt.$$

$$CP_{65}: mPosInd.pReq = True \rightarrow mCab.pWgt = mCab.pWgt + mPosInd.pWgt.$$

$$CP_{66}: mPhone.pRequ = True \rightarrow mCab.ctCost = mCab.ctCost + mPhone.ctCost.$$

$$CP_{67}: mLantern.pReq = True \rightarrow mCab.ctCost = mCab.ctCost + mLantern.ctCost.$$

$$CP_{68}: mIntercom.pReq = True \rightarrow mCab.ctCost = mCab.ctCost + mIntercom.ctCost.$$

$$CP_{69}: mPosInd.pReq = True \rightarrow mCab.ctCost = mCab.ctCost + mPosInd.ctCost.$$

- (2) If and only if no compensation cable is used, a counterweight guard is required, and the counterweight guard thickness is 0.75 inches. Its cost is counted into the total cost of counterweight assembly.

$$CP_{70}: mCompCb.pQt = 0 \rightarrow mCwtGuard.pReq = True.$$

$$CP_{71}: mCwtGuard.pReq = True \rightarrow mCwtGuard.pThick = 0.75 \text{ inches}.$$

$$CP_{72}: mCwtGuard.pReq = True \rightarrow mCwt.ctCost = mCwt.ctCost + mCwtGuard.ctCost.$$

- (3) The car supplement weight may be attached to the car to increase the hoist cable traction on the machine sheave. Initially, no supplement weight is added.

$$CP_{73}: mCar.pSuppleReq = Ture \rightarrow mCar.pWgt = mCar.pWgt + mCar.pSuppleWgt.$$

$$CP_{74}: mCar.pSuppleReq = Ture \rightarrow mCar.ctCost = mCar.ctCost + mCar.ctSupple.$$

- (4) If the machine beam support type is the pocket type, a machine beam bearing plate is required. The bearing plate is 1 inch thick. Its cost is also added to the total cost of machine beam module.

$$CP_{75}: mMchBm.pSptType = pocket \rightarrow mMchBmBear.pReq = True.$$

$$CP_{76}: mMchBmBear.pReq = True \rightarrow mMchBmBear.pThick = 1 \text{ inch, \&} \\ mMchBm.ctCostA = mMchBm.ctCostA + mMchBmBear.ctCost.$$

As an essential part in a configuration task, constraints C_g derived from requirements on elevator features and relevant building dimensions are listed as follows:

$$CR_1 : mCar.mCapac = 3000 \text{ pounds};$$

$$CR_2 : mCar.mSpeed = 250 \text{ feet/minute};$$

$$CR_3 : mCab.pHgt = 96 \text{ inches};$$

$$CR_4 : mLantern.pReq = Ture, \text{ which means a lantern in the cab is required};$$

$$CR_5 : mPhone.pReq = Ture, \text{ which means a phone in the cab is required};$$

$$CR_6 : mPosInd.pReq = Ture, \text{ which means a position indicator is required};$$

$$CR_7 : mDoor.mOpenType = side.$$

$$CR_8 : mDoor.mSpeed = double.$$

$$CR_9 : mDoor.mOpenStrik = left.$$

$$CR_{10} : mPlatfm.mWdt = 70 \text{ inches}.$$

$$CR_{11} : mPlatfm.mDpt = 84 \text{ inches}.$$

$$CR_{12} : mOverhd.mDpt = 192 \text{ inches}.$$

$$CR_{13} : mTravel.mDpt = 729 \text{ inches}.$$

$$CR_{14} : mPit.mDpt = 72 \text{ inches}.$$

$$CR_{15} : mHoist.mFloorHgt = 165 \text{ inches}.$$

$$CR_{16} : mHoist.mWdt = 90 \text{ inches}.$$

$$CR_{17} : mHoist.mDpt = 110 \text{ inches}.$$

$CR_{18} : mHoist.mOpenNo = 6.$

$CR_{19} : mHoist.mOpenWdt = 42 \text{ inches}.$

$CR_{20} : mHoist.mOpenHgt = 84 \text{ inches}.$

$CR_{21} : mHoist.mOpen2L = 32 \text{ inches}.$

$CR_{22} : mMchBm.mSptType = \text{pocket}.$

$CR_{23} : mMchBm.mSptDist = 118 \text{ inches}.$

$CR_{24} : mMchBm.mSptF2Hoist = 3 \text{ inches}.$

$CR_{25} : mMchBm.mSptB2RoomT = 16 \text{ inches}.$

$CP_{77} : \text{Min}(mElevator.ctCost) \mid \{mHoist.ctCost, mCar.ctCost, mCwt.ctCost, mSuspend.ctCost, mCable.ctCost\}$, which is a preference on solution.

Based on the formalisms above, the configuration domain Φ for the extended DCSP-based elevator system is constructed, which consists of 45 module variables V_M with corresponding 45 activation constraints C_a , many component variables V_{CM} (as some module variables hold consecutive values, V_{CM} is uncountable), 264 property variables V_P , 96 cost variables V_{CT} , 12 structural constraints C_s , and 362 generic constraints C_g . Based on Φ , the extended DCSP-based elevator system is initialized as $\langle V^I, D^I, C^I \rangle$, where $V^I = \{mElevator\}$; D^I is $\text{Domain}(mElevator) = cFinalElv$; $C^I = \{CS_1, CA_1, CR_4, CR_5, CR_6\}$, where CS_1 and CA_1 are structural constraint and activation constraint for the initial variable $mElevator$, CR_4, CR_5 and CR_6 are requirements on the selection of optional modules.

6.2 Solving DCSP-Based Elevator Configuration by Augmented BT Algorithm

After the extended DCSP-based elevator system is modeled, its solutions can be searched by the augmented Backtracking algorithm proposed in Section 4.1.

Starting from the initial DCSP-based elevator system $\langle \{mElevator\}, \{cFinalElv\}, \{CS_1, CA_1, CR_4, CR_5, CR_6\} \rangle$ modeled in the last section, the searching process of the augmented BT algorithm dynamically and systemically activates variables and constraints into the DCSP. The activation process is summarized in Table 6.2, where

Table 6.2: The activation of V^a and C^a in the extended DCSP-based elevator system using the augmented BT algorithm.

Activation	New V^a (Triggered by constraints)	New C^a
(1)	mElevator (Initial variable); cFinalElv; V_{CT} of mElevator (CA_1)	$CS_1; CA_1$ CR_4, CR_5, CR_6
(2)	mHoist, mCar, mCwt, mSuspend, mCable (CS_1); all V_{CM}, V_P, V_{CT} of new active modules(CA_2-CA_7)	$CS_2-CS_6; CA_2-CA_6$; $CR_1, CR_2, CR_{15}-CR_{21}$; $CC_1; 9 CE_j; 2 CP_k$
(3)	mOverhd, mTravel, mPit (CS_2); all $V_{CM},$ V_P, V_{CT} of new active modules (CA_7-CA_9)	$CA_7-CA_9; CR_{12}-CR_{14}$; $CC_2, CC_9-CC_{11}; 2 CE_j$
(4)	mCab, mSupport, mGov, mCarBuf(CS_3); all V_{CM}, V_P, V_{CT} of new active modules ($CA_{10}-CA_{13}$)	$CS_7-CS_9; CA_3, CA_{13}$; $CC_3, CC_{15}, CC_{16}; CR_3$; $9 CE_j; 7 CP_k$
(5)	mDoor, mCarFrame (CS_7); mLantern, mPhone, mPosInd (CR_4-CR_6); all $V_{CM}, V_P,$ V_{CT} of new active modules($CA_{14}-CA_{17}, CA_{19}$)	$CA_{14}-CA_{17}, CA_{19}; CC_{12},$ $CC_{45}, CC_{46}; CR_7-CR_9$; $8 CM_i; 2 CE_j; 6 CP_i$
(6)	mPlatfm, mSling (CS_8); all V_{CM}, V_P, V_{CT} of new active modules (CA_{20}, CA_{21})	$CS_{10}; CA_{20}, CA_{21}; CR_{10}$; $CC_{13}, CC_{47}, CC_{48}; CR_{11}$; $16 CM_i; 10 CE_j; 7 CP_k$
(7)	mCrossHd, mCarGdshoe, mCarGdrail, mStile(CS_{10}); all V_{CM}, V_P, V_{CT} of new active modules($CA_{39}-CA_{42}$)	$CA_{22}-CA_{25}$; $CC_4, CC_{60}-CC_{64}$; $5 CM_i; 3 CE_j; 3 CP_k$
(8)	mGovCb, mGovLoop, mSafBm(CS_9); all $V_{CM},$ V_P, V_{CT} of new active modules($CA_{26}-CA_{28}$)	$CA_{26}-CA_{28}; CC_{50}-CC_{54}$; $15 CM_i; 8 CE_j; 7 CP_k$
(9)	mCwtFrame, mCwtPlate, mCwtGdshoe, mCwtGdrail, mCwtU, mCwtBuf(CS_4); $V_{CM},$ V_P, V_{CT} of new active modules($CA_{29}-CA_{34}$)	$CA_{29}-CA_{34}$; $CC_6, CC_{17}-CC_{34}$; $4 CM_i; 20 CE_j; 14 CP_k$
(10)	mHCb, mMch, mMchBm (CS_5); all $V_{CM},$ V_P, V_{CT} of new active modules ($CA_{36}-CA_{38}$)	$CS_{11}; CA_{12}; CA_{36}-CA_{38}$; $CC_7, CC_{36}, CC_{37}, CC_{39},$ $CC_{40}; CR_{22}-CR_{25}$; $22 CM_i; 16 CE_j; 9 CP_k$
(11)	mMotor, mShv, mDefshv, mSensor(CS_{11}); all V_{CM}, V_P, V_{CT} of new active modules ($CA_{39}-CA_{42}$)	$CA_{39}-CA_{42}$; $CC_{38}, CC_{55}-CC_{58}$; $11 CM_i; 24 CE_j; 9 CP_k$
(12)	mMchBmBear (CS_{12}); all V_{CM}, V_P, V_{CT} of the new active module(CA_{43})	$CA_{43}; CC_{59}$; $CM_{74}; CP_{76}$
(13)	mContCb, mCompCb (CS_6); all $V_{CM}, V_P,$ V_{CT} of new active modules (CA_{45}, CA_{46})	$CA_{44}, CA_{45}; CC_{41}-CC_{44}$; $8 CM_i; 14 CE_j; 8 CP_k$

each row “Activation X” represents a new search phase in the DCSP solving process. In each phase, variables activated into the DCSP are listed in column “New V^a (Triggered by constraints)”, where constraints in parentheses refer to the constraints activating the variables into the DCSP. For example, in Activation (5), module vari-

ables $mDoor$ and $mCarFrame$ are activated by structure constraint CS_7 as their parent module $mCab$ is active in Activation (4), and $mCab$'s optional child modules $mLantern$, $mPhone$, and $mPosInd$ are activated due to customer requirements CR_4 – CR_6 . Their corresponding property variables $p \in V_P$ and cost variables $t \in V_{CT}$ (such as property and cost variables of $mDoor$ listed in Table 6.1) are activated by activation constraint CA_{14} – CA_{17} and CA_{19} . As in the augmented BT algorithm, there is no heuristic restricting the variable ordering, activation of variables is operated according to an elevator PFA, and the DCSP solving process is performed depth-first following the PFA. Subsequently, new active constraints C^a related to the new V^a are introduced into the DCSP searching process, as listed in column “New C^a ” of Table 6.2. It is noted that in the extended DCSP-based elevator system, some constraints are tied to certain potential fixes when they are violated (page 149). Then in addition to the condition that all variables in C^a are active, one more restriction on the activation of C^a is required; that is, all variables in its potential fixes (CP_i) are also required to be active. For example, if we only consider variables in a constraint, compatibility constraints CM_{46} and CM_{47} (page 153) are activated in Activation (10). But if also considering variables in their potential fixes, they are activated in Activation (13), together with their potential fixes CP_{54} – CP_{57} , since variable $mCompCb$ in CP_{56} is not activated until Activation (13).

Similar to the solving process of the augmented BT algorithm on the extended DCSP-based PC system (Section 4.3.2), in the solving process of the elevator system, unary constraint consistency (i.e., *node consistency*) is first checked on the current variable cv in each step. As restricting consistent values of singular variable, unary constraints remove infeasible values from cv 's domain, and thus, more directly prune the search space. As customer requirements are modeled as unary constraints, subsequent *arc consistency* checking is only implemented within cv 's remaining values meeting customer needs, and removes cv 's values conflicting with instantiated variables. This can improve searching efficiency due to a smaller search space. If cv 's domain becomes empty, BT occurs.

In the DCSP solving process, the consistency of C^a is checked whenever all its variables are instantiated, which consist of those both in the C^a and in its potential fixes (CP_i). Once a constraint violation occurs, modifications are immediately implemented to remedy it. There are two scenarios for the modification. The first one is that if there is no potential fix for the constraint violation, the solving process is implemented as same as that defined in Section 4.1. That is, the solving process discards the previous assignment of the current variable cv , and reassigns cv . If cv 's domain becomes empty, the solving process backtracks to the last assigned variable. In the second scenario, certain potential fixes correspond to the constraint violation. The fixes are implemented in ascending order of desirability (page 149). That is, a potential fix with the lowest desirability is tried first. If it fails, then a potential fix with the next lowest desirability is tried. If there are more than one fixes with the same desirability, the selection is arbitrary. Thereafter, the solving process backtracks or backjumps (i.e., more than 1 step) to corresponding variable in the fix.

For example, variable $mMotor$ is activated in search phase Activation (11) (Table 6.2), and is selected as the current variable. In Step 197, its property variable $mMotor.pModel$ is initialized as $c20HP$, as it holds the smallest horsepower that is not less than $mMotor.pHpReq$ calculated by constraint CE_{107} (page 155). But according to constraint CM_{29} (page 151), when $mMch.pModel$ (activated in Activation (10)) is assigned as $c18$ (Step 155), $mMotor.pModel$ is either $c10HP$ or $c15HP$. That is, CM_{29} is violated. Thus, corresponding to its potential fix CP_{40} (page 151), $mMch.pModel$ is upgrade. The solving process backjumps to $mMch.pModel$ (Step 155), discards its previous assignment $c18$, and reassigns it as $c28$ (Step 198). Previous assignments between Step 155 and Step 197 are all discarded, and corresponding variables are all re-evaluated.

There is an example for multiple potential fixes for a constraint violation. In Activation (7), after the assignment of $mCrossHd.phgt$ in Step 87, constraint CE_8 (page 150) is checked, and a violation is detected. As there are five potential fixes

for the violation, the one with the lowest desirability; i.e., CP_5 with code D1, is first implemented. Consequently, the solving process backjumps to $mCwt.pBtmRef$ (activated in Activation (2), and assigned in Step 36), reassigns it, and then re-evaluates all variables assigned between Step 36 and Step 87.

Unfortunately, compared with the extended DCSP-based PC system (proposed in Section 4.3), in the extended DCSP-based elevator system, some constraints and their potential fixes are defined on variables that are activated in different search phases. This may lead to more BT. One case is that the check of some variables is “delayed”, as constraints on the variables are activated in later search phases. For example, variable $mCwt.pBG$ (activated in Activation (2)) is not checked until Activation (9), in which constraint CM_7 (page 151) related to $mCwt.pBG$ is activated. If CM_7 is violated, BT happens. Guided by CM_7 ’s potential fix CP_{14} , the solving process backjumps to $mCwt.pBG$ and assignments between $mCwt.pBG$ (Activation (2)) and $mCwtBuf.pQt$ (Activation (9)) are all discarded.

The second case is that the activation and consistency check of some constraints are “delayed”, as a constraint is not activated until all variables both in the constraint and in its potential fixes are assigned. For example, although all variables in constraints CM_{46} and CM_{47} (page 153) are activated in Activation (10), CM_{46} and CM_{47} are delayed to be activated and checked in Activation (13), due to the activation of variable $mCompCb$ in their potential fix CP_{56} . Another example is that although the only variable $mCwt.pRunby$ in CE_{21} (page 151) is activated in Activation (2), CE_{21} is not activated and checked until Activation (9), in which variable $mCwtBuf.pBlkHgt$ in its potential fix CP_{23} is activated.

The third case is that potential fixes may result in more BT. For example, in constraint CM_{33} (page 152), modules $mHiost$, $mDoor$, and $mPlatfm$ are activated in Activation (2), (5) and (6) respectively. Thus CM_{33} is checked in Activation (6). Once it is violated, according to its potential fix CP_{44} , the solving process directly backjumps to $mHiost$ (Activation (2)), and all assignments between $mHiost$ and

$mPlatfm$ (Activation (6)) are discarded. However, assuming there is no potential fix for CM_{33} 's violation, the solving process reassigns $mPlatfm.pWdt$ (Activation (6)) to remedy the violation. If the domain of $mPlatfm.pWdt$ becomes empty, the solving process backtracks to the last assigned variable step by step, such as backtracking to $mDoor.pOpenType$ (Activation (5)) to reassign it. During this BT process, if the constraint violation is fixed, then the solving process continues to the next yet-to-be assigned variable, and does not necessarily backtrack to $mHlost$ in Activation (2).

However, potential fixes for constraint violations also may reduce the occurrence of BT, because they may guide the solving process directly to the real reason for violations. For example, after assignments of variables $mCompCb.pLdCarCarTop$ and $mContCb.pLdCarCarTop$ in Activation (13), constraint CE_{49} (page 154) is checked. When a violation of CE_{49} is detected, CP_{59} (that is CE_{49} 's potential fix) guides the solving process to directly backjump to $mCarGdrail$ that is the real conflicting reason and is activated in Activation (7). Assuming that there is no potential fix for the violation, the solving process backtracks from assigned variables in Activation (13) to $mCarGdrail$ in Activation (7). In the BT process, all assignments, constraint checking, and backtracking on these variables are repeated. It is time-consuming. Similarly, in the case of CM_{33} described in the last paragraph, if $mHlost$ is the real conflicting reason, then the potential fix may greatly improve the solving efficiency than that in the case of no fix.

In a more complex case, trade-off of BT occurrence is caused by one constraint and its corresponding potential fixes. For example, constraint CM_{46} (page 153), together with its potential fixes CP_{54} - CP_{57} , are delayed to be checked in Activation (13) as the late activation of $mCompCb$ in CP_{56} . This may lead to more BT (referring to the second case of BT increase, page 165). On the other hand, potential fixes may directly guide the solving process to the real conflicting reason (such as $mCwt$ in CP_{54} , which is activated in Activation (2)), so as to skip wasted assignments and BTs.

Such trade-off of BT occurrence due to potential fixes and its effect on the solving efficiency are difficult to be concluded briefly and exactly, and are case-dependent.

As the solving process of the augmented BT algorithm on the extended DCSP-based elevator system is complex and relatively time-consuming, it is unpractical to list all its steps. The basic searching process is similar to that on the extended DCSP-based PC system proposed in Section 4.3.2, except the process for handling constraint violations with potential fixes described above. Then, detailed steps of the solving process are not discussed here. Readers can refer to Section 4.3.2.

6.3 Solving DCSP-Based Elevator Configuration Using FC-BJ with FF

In this section, the algorithm FC-BJ with FF heuristic proposed in Section 5.1 is performed on the extended DCSP-based elevator system.

As defined in Section 5.1, FC-BJ with FF heuristic selects a current variable to be assigned in an ascending order of domain size. If there are more than one variables with the same smallest domain size, the selection is performed according to the lexicographic ordering. In the algorithm, although FF heuristic is implemented only on the selected current variable, and does not directly restrict the variable activation ordering, the activation process of variables and constraints (summarized in Table 6.3) is still different from that in the augmented BT algorithm (Table 6.2), since in these algorithms, the ordering of variables to be assigned in each phase may be different, based on which new child variables are activated into the DCSP.

The first two activation phases in the solving process of FC-BJ with FF heuristic (Table 6.3) are the same as those in the augmented BT algorithm (Table 6.2). However, in Activation (2) of FC-BJ with FF heuristic (Table 6.3), all the five variables have the same domain size 1. Then according to the lexicographic ordering, the selection order of current variables is $mCable$, $mCar$, $mCwt$, $mHoist$, and $mSuspend$. Subsequently, based on this order, all mandatory child modules of $mCable$ and $mCar$ with corresponding V_{CM} , V_P , and V_{CT} are activated into the DCSP in Activation (3) and Activation (4) respectively.

Table 6.3: The activation of V^a and C^a in the extended DCSP-based elevator system using FC-BJ with FF.

Activation	New V^a (Triggered by constraints)	New C^a
(1)	mElevator (Initial variable); cFinalElv; V_{CT} of mElevator (CA_1)	$CS_1; CA_1$ CR_4, CR_5, CR_6
(2)	mCable, mCar, mCwt, mHoist, mSuspend (CS_1); all V_{CM}, V_P, V_{CT} of new active modules(CA_2 - CA_7)	CS_2 - $CS_6; CA_2$ - CA_6 ; CR_1, CR_2, CR_{15} - CR_{21} ; CC_1 ; 9 CE_j ; 2 CP_k
(3)	mContCb, mCompCb (CS_6); all V_{CM}, V_P , V_{CT} of new active modules (CA_{45}, CA_{46})	$CA_{44}, CA_{45}; CC_{41}$ - CC_{44} ; 1 CM_i ; 1 CE_j ; 1 CP_k
(4)	mCarBuf, mCab, mGov, mSupport (CS_3); all V_{CM}, V_P, V_{CT} of new active modules (CA_{10} - CA_{13})	CS_7 - $CS_9; CA_3, CA_{13}$; $CC_3, CC_{15}, CC_{16}; CR_3$; 10 CE_j ; 7 CP_k
(5)	mDoor, mCarFrame (CS_7); mLantern, mPhone, mPosInd (CR_4 - CR_6); all V_{CM}, V_{CT} , V_P of new active modules(CA_{14} - CA_{17}, CA_{19})	CA_{14} - $CA_{17}, CA_{19}; CC_{12}$, $CC_{45}, CC_{46}; CR_7$ - CR_9 ; 8 CM_i ; 2 CE_j ; 6 CP_i
(6)	mGovCb, mGovLoop, mSafBm(CS_9); V_{CM} , V_P, V_{CT} of new active modules(CA_{26} - CA_{28})	CA_{26} - $CA_{28}; CC_{50}$ - CC_{54} ; 10 CM_i ; 1 CE_j ; 3 CP_k
(7)	mPlatfm, mSling (CS_8); all V_{CM}, V_P, V_{CT} of new active modules (CA_{20}, CA_{21})	$CS_{10}; CA_{20}, CA_{21}; CR_{10}$; $CC_{13}, CC_{47}, CC_{48}; CR_{11}$; 20 CM_i ; 12 CE_j ; 9 CP_k
(8)	mCrossHd, mCarGdshoe, mCarGdrail, mStile (CS_{10}); all V_{CM}, V_P, V_{CT} of new active modules(CA_{39} - CA_{42})	CA_{22} - CA_{25} ; CC_4, CC_{60} - CC_{64} ; 6 CM_i ; 7 CE_j ; 8 CP_k
(9)	mCwtFrame, mCwtPlate, mCwtGdshoe, mCwtGdrail, mCwtU, mCwtBuf(CS_4); V_{CM} , V_P, V_{CT} of new active modules(CA_{29} - CA_{34})	CA_{29} - CA_{34} ; CC_6, CC_{17} - CC_{34} ; 4 CM_i ; 19 CE_j ; 11 CP_k
(10)	mOverhd, mTravel, mPit (CS_2); all V_{CM}, V_P, V_{CT} of new active modules (CA_7 - CA_9)	CA_7 - $CA_9; CR_{12}$ - CR_{14} ; CC_2, CC_9 - CC_{11} ; 6 CM_i ; 17 CE_j ; 11 CP_k
(11)	mHCb, mMch, mMchBm(CS_5); all V_{CM} , V_P, V_{CT} of new active modules(CA_{36} - CA_{38})	$CS_{11}; CA_{12}; CA_{36}$ - CA_{38} ; $CC_7, CC_{36}, CC_{37}, CC_{39}$, $CC_{40}; CR_{22}$ - CR_{25} ; 25 CM_i ; 17 CE_j ; 13 CP_k
(12)	mMotor, mShv, mDefshv, mSensor(CS_{11}); all V_{CM}, V_P, V_{CT} of new active modules (CA_{39} - CA_{42})	CA_{39} - CA_{42} ; CC_{38}, CC_{55} - CC_{58} ; 11 CM_i ; 24 CE_j ; 8 CP_k
(13)	mMchBmBear (CS_{12}); all V_{CM}, V_P, V_{CT} of the new active module(CA_{43})	$CA_{43}; CC_{59}$; $CM_{74}; CP_{76}; CP_{77}$

In Activation (4), $mCab$, $mGov$, and $mSupport$ have child modules as shown in the elevator PFA (Figure 6.2, page 147). According to the principle of depth-first search, the solving process visits their child modules first. Consequently, following the

ascending order of the domain size of variables, child module variables of $mCab$, $mGov$, and $mSupport$ are activated into DCSP in Activation (5), (6), and (7) respectively. Thereafter, in Activation (8), child modules of $mSling$ (that is a child module of $mSupport$ and is activated in Activation (7)) are activated (i.e., depth-first). By now, the solving process finishes the search on the branch of module variable $mCar$, and continues to its sibling module variables. In the similar manner, all the other module variables, together with corresponding V_{CM} , V_P , and V_{CT} , are introduced into DCSP in Activation (9) to (13) respectively.

By the activation of variables and constraints, the solving process constructs a sequence of stable search phases, each of which only consists of active variables and related active constraints. In each stable phase, the algorithm FC-BJ with FF heuristic is implemented for consistent part solutions, which assign a value to each active variable without violating any active constraint. In each step, the solving process selects a variable with the smallest domain size as the current variable cv , initializes it, checks forwards the consistency of constraints on cv and its direct future variables fvs , and removes fvs ' inconsistent values. If the domain of fv becomes empty, then the solving process backjumps to fv 's conflict variables.

The original algorithm FC-BJ only checks the consistency of constraints between cv and its fvs . However, in the extended DCSP-based elevator system, variables with related constraints are introduced into the DCSP in different search phases, in each of which FC-BJ is applied; thus, constraints between past variables and new active variables are missed. So when new active variables cv are activated, constraints between cv and assigned variables need to be complementarily checked first. Whenever the domain of cv becomes empty, backjumping happens.

In both forward-checking and complementary-checking, when the domain of a variable (fv in forward-checking or cv in complementary-checking) is empty (i.e., a constraint violation is detected), backjumping happens. If there are potential fixes for the violation, backjumping is based on the fixes. Otherwise, the solving process backjumps to the variable's conflict variables.

The searching process of FC-BJ with FF on the extended DCSP-based elevator system is similar to that on the PC system proposed in Section 5.3.1. The process and examples for handling constraint violations with potential fixes have been articulated in Section 6.2. Thus, detailed solving steps are not repeated in this section. Readers can refer to Section 5.3.1 and Section 6.2 for corresponding information.

6.4 Solving DCSP-Based Elevator Configuration Using FC-BJ with the Hybrid Heuristic

In this section, the augmented FC-BJ with the hybrid heuristic that integrates an amended most-constrained heuristic with FF heuristic (proposed in Section 5.2) is performed on the extended DCSP-based elevator system.

As defined in Section 5.2, in the augmented FC-BJ algorithm with the hybrid heuristic, the activation of variables depends on the number of constraints between a variable, together with its sibling variables, and past variables that are activated in previous search phases. The variable with the largest number of such constraints is activated. Based on this principle, the activation of V^a and C^a in the augmented FC-BJ with the hybrid heuristic is summarized in Table 6.4. It is different from that in the augmented BT algorithm (Table 6.2) and in FC-BJ with FF heuristic (Table 6.3).

The first two steps of the activation process for these three algorithms are the same. In each step of the solving process of FC-BJ with hybrid heuristic (Table 6.4), after all active variables have been initialized, the number of constraints between assigned variables and an unassigned variable, together with its sibling variables, is calculated. Then the set of sibling variables that has the largest number of such constraints is activated. Specifically, when all variables in Activation (2) are assigned, the number of constraints between assigned variables and each set of child module variables of the module variables in Activation (2) is calculated. Then $mCwt$'s child module variables are activated in Activation (3), as they are involved in the largest number 27 of constraints on assigned variables activated in Activation (1) and (2). In

Table 6.4: The activation of V^a and C^a in the extended DCSP-based elevator system using FC-BJ with the hybrid heuristic.

Activation	New V^a (Triggered by constraints)	New C^a
(1)	mElevator (Initial variable); cFinalElv; V_{CT} of mElevator (CA_1)	$CS_1; CA_1$ CR_4, CR_5, CR_6
(2)	mCable, mCar, mCwt, mHoist, mSuspend (CS_1); all V_{CM}, V_P, V_{CT} of new active modules(CA_2 - CA_7)	CS_2 - $CS_6; CA_2$ - CA_6 ; CR_1, CR_2, CR_{15} - CR_{21} ; CC_1 ; 9 CE_j ; 2 CP_k
(3)	mCwtFrame, mCwtPlate, mCwtGdshoe, mCwtGdrail, mCwtU, mCwtBuf(CS_4); V_{CM} , V_P, V_{CT} of new active modules(CA_{29} - CA_{34})	CA_{29} - CA_{34} ; CC_6, CC_{17} - CC_{34} ; 3 CM_i ; 18 CE_j ; 13 CP_k
(4)	mHCb, mMch, mMchBm (CS_5); all V_{CM} , V_P, V_{CT} of new active modules(CA_{36} - CA_{38})	$CS_{11}; CA_{12}; CA_{36}$ - CA_{38} ; $CC_7, CC_{36}, CC_{37}, CC_{39}$, $CC_{40}; CR_{22}$ - CR_{25} ; 23 CM_i ; 13 CE_j ; 7 CP_k
(5)	mMotor, mShv, mDefshv, mSensor(CS_{11}); all V_{CM}, V_P, V_{CT} of new active modules (CA_{39} - CA_{42})	CA_{39} - CA_{42} ; CC_{38}, CC_{55} - CC_{58} ; 11 CM_i ; 22 CE_j ; 4 CP_k
(6)	mMchBmBear (CS_{12}); all V_{CM}, V_P , V_{CT} of the new active module(CA_{43})	$CA_{43}; CC_{59}$; $CM_{74}; CP_{76}$
(7)	mOverhd, mTravel, mPit (CS_2); all V_{CM} , V_P, V_{CT} of new active modules (CA_7 - CA_9)	CA_7 - $CA_9; CR_{12}$ - $CR_{14}; CC_2$ CC_9 - CC_{11} ; 6 CE_j ; 3 CP_k
(8)	mContCb, mCompCb (CS_6); all V_{CM}, V_P , V_{CT} of new active modules(CA_{45}, CA_{46})	$CA_{44}, CA_{45}; CC_{41}$ - CC_{44} ; 4 CM_i ; 10 CE_j ; 8 CP_k
(9)	mCarBuf, mCab, mGov, mSupport (CS_3); all V_{CM}, V_P, V_{CT} of new active modules (CA_{10} - CA_{13})	CS_7 - $CS_9; CA_3, CA_{13}$; $CC_3, CC_{15}, CC_{16}; CR_3$; 12 CE_j ; 11 CP_k
(10)	mPlatfm, mSling (CS_8); all V_{CM}, V_P, V_{CT} of new active modules (CA_{20}, CA_{21})	$CS_{10}; CA_{20}, CA_{21}; CR_{10}$; $CC_{13}, CC_{47}, CC_{48}; CR_{11}$; 12 CM_i ; 11 CE_j ; 8 CP_k
(11)	mCrossHd, mCarGdshoe, mCarGdrail, mStile(CS_{10}); all V_{CM}, V_P, V_{CT} of new active modules(CA_{39} - CA_{42})	CA_{22} - CA_{25} ; CC_4, CC_{60} - CC_{64} ; 6 CM_i ; 4 CE_j ; 6 CP_k
(12)	mDoor, mCarFrame (CS_7); mLantern, mPhone, mPosInd(CR_4 - CR_6); V_{CM}, V_P, V_{CT} of new active modules (CA_{14} - CA_{17}, CA_{19})	CA_{14} - $CA_{17}, CA_{19}; CC_{12}$, $CC_{45}, CC_{46}; CR_7$ - CR_9 ; 12 CM_i ; 4 CE_j ; 7 CP_i
(13)	mGovCb, mGovLoop, mSafBm(CS_9); V_{CM} , V_P, V_{CT} of new active modules(CA_{26} - CA_{28})	CA_{26} - $CA_{28}; CC_{50}$ - CC_{54} ; 13 CM_i ; 8 CE_j ; 9 CP_k

Activation (4), $mSuspend$'s child module variables are activated, as they are involved in the largest number 20 of constraints on assigned variables activated in Activation (1)–(3). Subsequently, in Activation (4), as variables $mMch$ and $mMchBm$ have child modules, based on the principle of depth-first search, these child modules are

under prior consideration. They are activated in Activation (5) and (6) respectively in a descending order of the number of qualified constraints. Thereafter, all unassigned variables are activated into the DCSP in Activation (7) to (13).

In each search phase listed in Table 6.4, FC-BJ with the hybrid heuristic is implemented on a stable DCSP. Its solving process is the same as that of FC-BJ with FF (Section 6.3), except the selection ordering of current variables cv . The selection is based on the following rules: First, the selection is performed in a descending order of the number of constraints that are defined between an unassigned active variable and assigned variables activated in previous search phases. That is, the unassigned variable with the largest number of such constraints is selected as cv first, and the least the latest. If there are more than one variables with the same number of such constraints, including there is no constraint, the selection is performed on FF heuristic; i.e., the variable with the smallest domain size is selected as cv , or the selection is on the lexicographic ordering if more than one variables have the same domain size. The detailed solving process of the augmented FC-BJ with the hybrid heuristic is not discussed in this section. Readers can refer to Section 6.3.

6.5 Experimental Results

In this section, experiments on the extended DCSP-based elevator system with three proposed algorithms and heuristics are implemented. To test their solving efficiency, comparisons are preformed between four pairs of algorithms: (1) between the extended DCSP-based elevator system with the augmented BT algorithm (denoted as “DCSP+BT” in experimental results) and a classical CSP-based elevator configuration system with standard backtracking algorithm (denoted as “CSP”); (2) between algorithm FC-BJ with FF heuristic (denoted as “DCSP+FC-BJ-FF”) and “CSP”; (3) between “DCSP+FC-BJ-FF” and “DCSP+BT”; and (4) between “DCSP+FC-BJ-FF” and algorithm FC-BJ with the hybrid heuristic (denoted as “DCSP+FC-BJ-hybrid”). All experiments are implemented on a personal work station with an Intel® Core™ 2 Quad Q9400 processor at 2.66 GHz with 4GB of RAM.

Figure 6.3: Experimental results for generating a feasible configuration.

Figures 6.3 to Figures 6.7 illustrate experimental results. Similar to the experimental result of the extended DCSP-based PC system (page 106), two comparison criteria are evaluated in the experiments: the number of nodes visited and CPU time in clock clicks consumed in a searching process. The abscissa “number of selected modules” refers to the number of module variables in a final solution, and varies from 39 (including only mandatory modules) to 45 (also including all optional modules). It also represents the number of module variables included in the searching process of the extended DCSP-based elevator system. Consequently, corresponding to selected module variables, the number of other variables (i.e., component, property, and cost variables) and constraints involved in the searching process also varies. However, in a classical CSP-based system, all 45 modules and all corresponding variables and constraints are involved in the searching process.

Figure 6.3 illustrates experiment results for generating the first feasible solution. It is observed that the extended DCSP-based elevator system with three proposed algorithms (i.e., “DCSP+BT”, “DCSP+FC-BJ-FF”, and “DCSP+FC-BJ-Hybrid”) outperform the CSP-based elevator system with standard BT algorithm (i.e., “CSP”), as they visit less nodes in the solving process. Whereas the better performance is not as distinct as that in the extended DCSP-based PC system (Figure 4.8, page 107, and Figure 5.3, page 138), because in the DCSP-based elevator system, there are more constraints on variables that are activated in different search phases, which may

6.4.a: Results on the no. of visited node.

6.4.b: Results on CPU time.

Figure 6.4: Experimental results for generating all configuration solutions (“DCSP+BT” vs. “CSP”).

result in more BT. Furthermore, among three proposed algorithms, “DCSP+FC-BJ-FF” exhibits better performance than “DCSP+BT”, and “DCSP+FC-BJ-Hybrid” achieves the best performance. While the performance improvement of “DCSP+FC-BJ-Hybrid” is noticeable, the improvement between “DCSP+FC-BJ-FF” and “DCSP+BT” is not obvious; in fact, it is worse (e.g., when the number of selected modules is 41). It is also because more BT result from constraints on variables activated in different phases, and counteract the improvement of solving efficiency of the augmented FC-BJ with FF heuristic. But in “DCSP+FC-BJ-hybrid”, a variable with the largest number of constraints on past variables is checked first. Thus the algorithm is able to detect violations of constraints whose variables are activated in different phases at a relatively early time, and then may greatly reduce BT occurrences.

Experiments for generating all configuration solutions are also performed, and results are compared in terms of the number of visited nodes, and CPU time respectively. Figure 6.4 illustrates experimental results between “CSP” and “DCSP+BT”. Compared with “CSP”, “DCSP+BT” significantly enhances the solving efficiency, and the improvement is gradually magnified, along with the increase of the number of active module variables, together with corresponding variables and constraints. Furthermore, comparisons among “DCSP+BT”, “DCSP+FC-BJ-FF”, and “DCSP+FC-

6.5.a: Results on the no. of visited node.

6.5.b: Results on CPU time.

Figure 6.5: Experimental results for generating all configuration solutions (“DCSP+BT” vs. “DCSP+FC-BJ-FF” vs. “DCSP+FC-BJ-Hybrid”).

Figure 6.6: Experimental results with cost optimization.

BJ-hybrid” are performed, and experimental results are illustrated in Figure 6.5. Both “DCSP+FC-BJ-FF” and “DCSP+FC-BJ-hybrid” outperform “DCSP+BT”, and “DCSP+FC-BJ-hybrid” offers the best performance. The reason is the same as that for generating the first feasible solution in the last paragraph; that is, the counteraction of complementary-checking in “DCSP+FC-BJ-FF”, and BT avoidance due to the early checking of the most constrained variables in “DCSP+FC-BJ-hybrid”.

In addition, taking cost optimization into account, such as minimum cost (CP_{77}), experiments on four algorithms are implemented, and are denoted as “CSP+BB” (referring to page 108), “DCSP+BT+Cost”, “DCSP+FF+Cost”, and “DCSP+Hybrid+Cost” respectively. Experimental results are illustrated in Figure 6.6. Similarly, three

Figure 6.7: Experimental results of DCSP+BT for the first feasible solution (with vs. without cost optimization).

proposed algorithms with cost optimization achieve better solving efficiency than “CSP+BB”, and the performance of “DCSP+Hybrid+Cost” is the best. Moreover, the solving efficiency of three proposed algorithms integrating with cost optimization is also compared with that of the algorithms without cost optimization. Figure 6.7 shows the comparison result of “DCSP+BT”. Compared with that in the extended DCSP-based PC system (Figure 4.12, page 110, and Figure 5.7, page 141), the performance improvement of “DCSP+BT+Cost” in the elevator system is relatively slight. It is because in the elevator system, potential fixes predefine backtracking steps when constraints are violated, and thus, to a great extent, result in a pre-determined solving process, and limit the diversity of configuration solutions. So the effect of cost optimization on configuration is limited. Similar results are obtained for algorithms “DCSP+FC-BJ-FF” and “DCSP+FC-BJ-hybrid”, and are not verbalized here.

For the elevator configuration task, case-based approach is not feasible, because some variables hold values with great uncertainty and diverse range, especially those related to building dimensions. For example, data of hoistway (*mHosit*) are specified for an elevator, and are changeful among different configurations. A small difference of these data may impact various variables (such as the machine beam, the platform, and the counterweight), and thus results in diverse elevator configurations that may be quite different from each other. In such scenario, it is difficult for historical cases

to provide enough support for complete configurations of new cases, unless the historical data set is unreasonable large. This exhibits the advantage of DCSP-based configuration approach; i.e., generality and flexibility. In fact, case-based approach can be integrated with the extended DCSP-based elevator system for better solving efficiency. Although case-based approach is not suitable for complete elevator configurations, it is able to be quite efficient for configuring certain part of an elevator, whose variables have no close relationship with variables that have great uncertainty and diversity (e.g., *mHosit*). For instance, in a DCSP-based elevator configuration, if car buffer is configured by case-based approach, then in the DCSP solving process, the data of car buffer is temporarily removed. Therefore the solving process has a smaller search space, and may perform less BT. However, the integration of the extended DCSP-based configuration system with case-based approach is beyond the scope of this thesis, and will not be discussed in this thesis.

6.6 Summary

In this chapter, an example of cable-operated elevator configuration task is proposed. The modeling process of the extended DCSP-based elevator system is described in detail. Subsequently, three solving algorithms proposed in the previous chapters (i.e., the augmented BT algorithm, the augmented FC-BJ with FF heuristic, and FC-BJ with the hybrid heuristic) are performed on the extended DCSP-based elevator system respectively. The solving process of these three algorithms is elaborated. Experiments are implemented on these three algorithms and a CSP-based system with standard BT algorithm, and experimental results are analyzed. Finally, the applicability of case-based approach on the elevator configuration task is discussed.

Chapter 7

Conclusions

7.1 Summary

As today's markets exhibit more but smaller niches and more uncertainty due to increasing demand heterogeneity, mass customization has been widely applied to the manufacturing industry, which aims at satisfying individual customer requirements while staying near mass production efficiency. It seeks to increase product variety and reduce production cost at the same time. Product family is an effective approach towards mass customization since it maintains high production volumes while provides a variety of products [39, 86, 170]. Due to the large amount and variety of data involved in product family design, problem representation and solution are the two key issues in product family design. Although many product family modeling methods have been proposed, most of them are not computationally efficient. Among various product family design approaches, Constraint Satisfaction Problem is a promising approach for both problem modeling and solving, due to its generality and flexibility. Moreover, costs are a significant composition and criterion in product family design. Although there have been a lot of efforts on product family design and cost estimation in both academic and industries, the integration of these two tasks has not been solved successfully.

In this thesis, an extended Dynamic Constraint Satisfaction Problem based product configuration and cost estimation system has been proposed, which is a computationally efficient approach to model and solve product family design problems. The

DCSP-based system extends classical CSP-based product configuration models such that it allows incremental extensions of a constraint network corresponding to the DCSP-based system while capturing the internal structure in product family design. In this scheme, a product family design task is represented as a DCSP by various variables with associated domains and constraints corresponding to both the semantics and structure of a product family architecture. Modules in a product family, components in product variants, and associated properties and cost drivers are represented as corresponding types of variables in DCSP. Each variable holds one of two states: active and inactive, and all the variables are initially inactive. Only active variables are instantiated and involved in final configurations. Restrictions on these variables that limit the available combinations of the values of these variables are represented as compatibility or incompatibility constraints. To capture the inherent hierarchy of data in product family design that is generally missed in existing CSP-based approaches, the extended DCSP-based system extends classical CSP-based models by structural constraints that represent relationships among modules (i.e., parent-child relationships) in product families, and extend such relationships to corresponding components in product variants. We also introduce activation constraints that entail derivation relationships between module variables and associated property and cost variables; that is, each module and corresponding instance components possess a set of properties and cost drivers. Both structural constraints and activation constraints can be utilized to extend the constraint network of the extended DCSP-based system incrementally to address the case that in many more complex real-life applications, the number of components involved in the final configuration is not known *a priori*. To tightly integrate cost estimation task with product configuration, we propose cost constraints in the form of mathematic expressions for representing cost relationships among cost drivers and other property variables. Two categories of cost constraint (i.e., cost calculation constraints and cost variation constraints) are identified, and have different implementation in a DCSP solving process. We also provide preference

constraints that have relatively lower priority in the extended DCSP-based system to incorporate customer preferences with product configuration. All variables with associated domains and constraints are predefined and stored in a configuration domain Φ of the extended DCSP-based product configuration and cost estimation system.

Traditionally, data required in product family design are stored in various product data models (PDMs) used by manufacturers. The formalism of the extended DCSP-based product configuration and cost estimation system needs to be derived from PDMs. In this thesis, three types of PDMs have been briefly introduced—PDM for product configuration, PDM for requirement, and PDM for design knowledge, to clarify elements in product family design. Mapping relationships between these elements stored in PDMs and the concepts of DCSP (i.e., variables, domains and constraints) are also analyzed in this thesis to facilitate problem representation in DCSP terms. Unfortunately, some useful information of the configuration and cost estimation problem may be lost in DCSP terms. Thus, PDMs that are semantics-rich are also used to provide background knowledge for DCSP, and provide knowledge foundation of specific DCSP solving and heuristic algorithm for particular configuration tasks. We also apply association rule mining approach on historical product configuration data for automatical generation of compatibility and incompatibility constraints in the extended DCSP-based system.

Based on the framework of extended DCSP-based product configuration with cost estimation, an augmented Backtracking algorithm has been proposed to solve it where structural constraint handler, activation constraint handler and cost variation constraint handler are embedded into a classical BT procedure for discrete CSPs. Generally speaking, the augmented BT process implemented on the extended DCSP-based model can be viewed as a sequence of stable phases, in each of which the constraint network of the extended DCSP-based system refers to a fixed classical CSP where only active constraints on active variables are propagated until a consistent state (i.e., a consistent partial solution) is reached.

Starting with an initial CSP including a set of active variables, together with associated domains, and active constraints predefined in configuration domain Φ , a solving process of the extended DCSP-based system is implemented by extending corresponding constraint network incrementally. A new phase is generated by the activation of variables with the subsequent introduction of new active constraints. Specifically, using structural constraints, module variables are activated into DCSP according to product family architecture and customer requirements; that is, based on parent-child relationships in a PFA, all common module variables must be activated and some differentiation module variables are activated to meet specific customer needs. Then an appropriate set of property and cost variables associated with activated modules are also activated into DCSP by activation constraints. Subsequently, constraints whose variables are all active are activated into DCSP.

In each stable phase, a general backtracking procedure is performed on a fixed CSP constraint network. If a consistent partial solution to all active variables is reached without new variables being activated, the partial solution is a final solution. Otherwise, the extended DCSP-based system continues with a new phase; i.e., a new extended constraint network.

In this thesis, implementations of the two categories of cost constraints in the augmented BT algorithm have also been discussed. Cost calculation constraints are performed as ordinary constraints (e.g., compatibility constraints) checking constraint consistency in the DCSP solving process. Then the results of product configuration and cost estimation can be obtained simultaneously, and results of cost calculation constraints can be used as a concurrent criterion for product configuration. Hence, cost estimation integrates with product configuration tightly. To study the impact of change of cost drivers and design parameters on the cost of a product variant, or the entire product family, or even the whole product lifecycle cost, cost variation constraint handler is embedded into the extended DCSP system.

As the augmented BT algorithm controls the number of variables activated in each search phase, it achieves good solving efficiency. However, it still suffers from the issues

of classical BT algorithm; that is, trashing and late detection of conflicts. To address these issues, a hybrid solving algorithm (i.e., Forward Checking-Backjumping algorithm with Fail-First heuristic) has been proposed on the augmented framework for solving the extended DCSP-based product configuration and cost estimation system. In each stable phase of the searching process of the extended DCSP-based system, FC algorithm checks constraint consistency between current variable and its direct future variables, such that conflicts can be detected before they really occur. When the domain of a future variable becomes empty; i.e., conflict happens, backjumping algorithm guides the DCSP searching process backwards to the most recent conflict variable of the variable, and thus, avoids unnecessary backtracking; i.e., trashing. As forward checking algorithm prunes inconsistent values of future variables with current variable, the size of their remaining domain can be computed without any additional constraint checks [8]. Then we conveniently combined Fail-First heuristic with the hybrid FC-BJ algorithm for dynamic variable reordering, which first assigns a variable with the smallest domain size as it is more likely to lead to search failure, and achieves a better performance on the extended DCSP-based system.

However, the performance of the augmented FC-BJ with FF algorithm may be reduced in the case when there are many constraints between variables activated in different activation phases, which may lead to a large amount of backjumping. To address this issue, we proposed a hybrid heuristic (i.e., a combination of an amended most-constrained heuristic with fail-first heuristic) with FC-BJ algorithm on the extended DCSP-based system. Using this hybrid heuristic, the activation of new variables to generate a new search phase and the selection of current variable to be instantiated are all based on the descending order of the number of constraints that are defined between new active variables and assigned variables, because most-constrained variables are more likely to result in backtracking in the solving process of the extended DCSP-based system. If there are more than one variables having the same number of constraints on past variables, the one having the smallest remaining domain is selected; i.e., fail-first.

7.2 Future Work

The approaches for modeling and solving product family design tasks proposed in this thesis partly solve some challenges in product family design. However, there are still several issues and/or drawbacks that remain to be addressed. Some of them are briefly summarized as below:

- The solving algorithms and heuristics proposed in this thesis are only suitable for configuration problems involving variables with discrete values and binary constraints. However, in many engineering applications, continuous variables and n-ary constraints are unavoidable [212]. One solution is to transform n-ary constraints into several binary constraints. However, a forced binarization may result in significant increase to problem size [16] and unnatural formulations causing extra difficulties in CSP solving [18], so it may be impractical. We will carefully examine how to solve CSP-based configuration with continuous variables and n-ary constraints to make the extended DCSP-based configuration and cost estimation system closer to real engineering applications.
- In the extended DCSP-based product configuration and cost estimation system proposed in this thesis, the data of product configuration, customer requirements, and product design domain knowledge have been considered. As product lifecycle management has been a trend in today's manufacturing industry, more data related to PLM, such as after-sale services, product recycling and disposal, and product lifecycle cost, should be taken into account in product configuration. Consequently, a larger amount and variety of data are involved in the extended DCSP-based system, and thus, data update becomes a tough task. Therefore, how to represent and update these data efficiently and effectively will be included in the future works.
- In this thesis, mapping relationships between elements in PDMs and the formalism of the extended DCSP-based product configuration and cost estimation

system have been discussed. However, by now, such relationships are mainly analyzed in theory, and mappings are implemented manually. How to build an automatic mapping mechanism to realize such derivation relationships will be considered.

- In this thesis, cost constraints are constructed by activity-based costing method. Although ABC is a successful cost estimation approach, it is not suitable for all cases. As PLM has become a responsibility of manufacturers, the entire product lifecycle cost such as costs occurred in product maintenance and end-of-life phases, are required to be considered in product family design and cost estimation tasks. Thus, other cost estimation algorithms or a hybrid cost estimation approach may be required to be applied to construct cost constraints.
- General speaking, in product family design, decision on the selection of a part among parts of equivalent functionality in product variants is primarily based on its purchasing price, which leads to a large number of part varieties. Additionally, the use of purchasing price as the decision criterion also contributes to the frequent change of parts whenever a cheaper alternative becomes available. The impact in costs by increased part variety and frequency of part change is ignored due to the lack of scientific and effective evaluation methods. Although cost variation constraints in the extended DCSP-based system have been introduced to analyze the cost impact of part variety and/or change on production cost, or even the entire product lifecycle costs, the work on this impact analysis is still in an initial stage and will be further examined.
- Product family design provides a wide variety of product variants within a product family to meet customer requirements. However, wide variety does not mean customization. Sometimes customers want a product that exactly matches their requirements. In this case (e.g., web-based product configuration), the configuration process generally requires user interaction [108], and is solving-time

sensitive. One solution to this case is to separate the DCSP-based system into two parts, one is static part that is constant in the solving process, another part is dynamic and needs to utilize specific algorithms to solve it in limited time [108] that will be a part of the future works.

7.3 List of Publications

[Journal papers]

Extended DCSP Approach on Product Configuration with Cost Estimation,

Wang Lin, Ng Wee Keong and Song Bin

Journal of Concurrent Engineering: Research and Applications [SAGE], volume 19, number 2, pp 123-138, 2011.

Hybrid Solving Algorithms for an Extended DCSP-Based Configuration System,

Wang Lin, Ng Wee Keong

Journal of Concurrent Engineering: Research and Applications [SAGE] volume 20, number 3, pp 223-236, 2012.

[Conference papers]

Semantic Modeling for DCSP-Based Product Configuration,

Wang Lin, Ng Wee Keong

IEEE TENCON 2009 (Theme: Emerging Technologies for Sustainable Development), Singapore, 2009.

Sustaining Product Competitiveness with CSP on Product Configuration and Life-cycle Cost,

Wang Lin, Ng Wee Keong, and Song Bin

IEEE TENCON 2009 (Theme: Emerging Technologies for Sustainable Development), Singapore, 2009.

Constraint Satisfaction Approach on Product Configuration with Cost Estimation,

Wang Lin, Ng Wee Keong, and Song Bin

The 22nd International Conference on Industry, Engineering & Other Application of Applied Intelligent Systems (IEA_AIE09), Taiwan, 2009.

A Product Family Based Life Cycle Cost Model for Part Variety and Change Analysis,

Wang Lin, Song Bin, Li Xiang, and Ng Wee Keong

International Conference on Engineering Design (ICED'07), Paris, France, 2007.

References

- [1] R. Aboulafia, *Airbus Pulls Closer to Boeing*, Aerospace America, **38**, pp. 16-18, 2000.
- [2] B. Agard and A. Kusiak, *Data-Mining-Based Methodology for the Design of Product Families*, International Journal of Production Research, **42**(15), pp. 2955-2969, 2004.
- [3] M. Agarwal and J. Cagan, *Shape Grammars and Their Languages A Methodology for Product Design and Product Representation*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC97/DTM-3867, 1997.
- [4] M. Agarwal and J. Cagan, *Shape Grammar-Base Expert System for Engineering Design*, Artificial Intelligence in Design (J.S. Gero Ed.), pp. 193-202. Worcester, MA: Kluwer, 2000.
- [5] S. Akundi, T.W. Simpson, and P.M. Reed, *Multi-Objective Design Optimization for Product Platform and Product Family Design Using Genetic Algorithms*, ASME Design Engineering Technical Conferences, DETC2005/DAC-84905, Long Beach, CA, USA, 2005.
- [6] M. Aldanondo, K. Hadj-Hamou, G. Moynard, and J. Lamothe. *Mass Customization and Configuration: Requirement Analysis and Constraint Based Modeling Propositions*, Integrated Computer-Aided Engineering, **102**, pp. 177-189, 2003.
- [7] M.C. Andrade, R.C.P. Filho, A. M. Espozel, L.O.A. Maia and R.Y. Qassim, *Activity-Based Costing for Production Learning*, Int. J. Prod. Econ., **62**(3), pp. 175-180, 1999.

- [8] F. Bacchus and A. Grove, *On The Forward Checking Algorithm*, CP'95 Proceedings of the First International Conference on Principles and Practice of Constraint Programming, **976**, pp. 292-309, France, September 19-22, 1995.
- [9] J. Bachant, RIME Preliminary Work toward a Knowledge-Acquisition Tool, Automating Knowledge Acquisition for Expert Systems (S. Marcus, ed.), Kluwer Academic Publishers, Boston, pp. 201-224, 1988.
- [10] R.A. Baldwin and M.J. Chung, *Managing Engineering Data for Complex Products*, Research in Engineering Design, **7**(4), pp. 215-231, 1995.
- [11] C.Y. Baldwin and K.B. Clark, *Design Rules: The Power of Modularity*, Cambridge, MA: MIT Press, 2000.
- [12] V.E. Barker, D.E. O'Connor, J. Bachant, and E. Soloway. *Expert Systems for Configuration at Digital: XCON and Beyond*, Communications of the ACM, **323**, pp. 298-318, 1989.
- [13] R. Barták, *Constraint Programming - What is behind?*, In J. Figwer (editor) Proceedings of the Workshop on Constraint Programming in Decision and Control, Poland, June, 1999.
- [14] R. Barták, *Constraint Propagation And Backtracking-based Search - A brief introduction to mainstream techniques of constraint satisfaction*, available from <http://kti.mff.cuni.cz/~bartak/downloads/CPschool05notes.pdf>, publisher: Cite-seer, pp. 1-42, 2005.
- [15] Y. Bei and K.J. MacCallum, *Decision Support for Configuration Management in Product Development*, The 3rd International Conference on Computer Integrated Manufacturing, **1**, pp. 278-286, Singapore: World Scientific, 1995.
- [16] C. Bessière, *Non-Binary Constraints*, Proceedings of Principles and Practice of Constraint Programming (CP-99), pp. 24-27, 1999.

- [17] C. Bessière, R. Coletta, and T. Petit, *Learning Implied Global Constraints*, Proc. of the 22th Int'l Joint Conf. Artificial Intelligence (IJCAI 07), pp. 44-49, 2007.
- [18] C. Bessière, P. Meseguer, E. Freuder, and J. Larrosa, *On Forward Checking for Non-Binary Constraint Satisfaction*, Artificial Intelligence, pp. 205-224, 1999.
- [19] J.R.J. Bielefeld and G.D. Rucklos, *Cost Scaling Factors: How accurate are they?*, Cost Engineering, **34**(10), pp. 15-20, 1992.
- [20] M. Blackenfelt, *Design of Robust Interfaces in Modular Products*, ASME Design Engineering Technical Conferences, DETC00/DAC-14486, Baltimore, MD, USA, 2000.
- [21] J. Bode, *Neural Networks for Cost Estimation: Simulations and Pilot Application*, International Journal of Production Research, **38**(6), pp. 1231-1254, 2000.
- [22] G. Boothroyd, P. Dewhurst and W. A. Knight, *Product Design for Manufacture and Assembly*, New York: Marcel Dekker, 1994.
- [23] J. Bowen and D. Bahler, *Conditional Existence of Variables in Generalized Constraint Networks*, Proceedings of the 9th National Conference on Artificial Intelligence, pp. 215-220, 1991.
- [24] B. Bras and J. Emblemavag, *Designing for the Life Cycle: Activity Based Costing and Uncertainty*, In design for X, Concurrent Imperatives, Chapman & Hall, UK, 1996.
- [25] P.G. Brabazon and B. MacCarthy, *Virtual-Build-to-Order as a Mass Customization Order Fulfilment Model*, Concurrent Engineering: Research and Application, **12**(2), pp. 155-165, 2004.
- [26] R.T. Caffrey, T.W. Simpson, R. Henderson, and E. Crawley, *The Economic Issues with Implementing Open Avionics Platforms for Spacecraft*, 20th AIAA Int.

- Communications Satellite Systems Conf. and Exhibit, Montreal, AIAA-2002-1870, 2002.
- [27] S. Cavalieri, P. Maccarrone and R. Pinto, *Parametric Vs Neural Network Models for the Estimation of Production Costs: A Case Study in the Automotive Industry*, Int. J. Prod. Econ., **91**(2), pp. 165-177, 2004.
- [28] O.L. Cetin and K. Saitou, *Decomposition-Based Assembly Synthesis for Structural Modularity*, ASME Journal of Mechanical Design, **126**(2), pp. 234-243, 2004.
- [29] B. Chandrasekaran, R.B. Stone, and D.A. Mcadams, *Developing Design Templates for Product Platform Focused Design*, Journal of Engineering Design, **15**(3), pp. 209-228, 2004.
- [30] K.L. Choy, W.B. Lee, H.C.W. Lau, D. Lu, and V. Lo, *Design of An Intelligent Supplier Relationship Management System for New Product Development*, International Journal of Computer Integrated Manufacturing, **17**:692, 2004.
- [31] A. Claesson, H. Johannesson, and S. Gedell, *Platform Product Development: Product Model- A System Structure Composed of Configurable Components*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC2001/DTM-21714, 2001.
- [32] B. Corbett and D.W. Rosen. *A Configuration Design Based Method for Platform Commonization for Product Families*. Artificial Intelligence in Engineering, Design, Analysis and Manufacturing, **18**, pp. 21-39, 2004.
- [33] D. Cutherell, *Product Architecture*. M. Rosenau, G. Griffin, G. Castellion & N. Anschuetz (Eds.), The PDMA handbook of new product development. John Wiley & sons, 1996.

- [34] Z. Dai and M.J. Scott, *Product Platform Design Through Sensitivity Analysis and Cluster Analysis*, ASME Design Engineering Technical Conferences, DETC2004/DAC-57464, Salt Lake City, UT, USA, 2004.
- [35] J.B. Dahmus and K.N. Otto, *Incorporating Lifecycle Costs into Product Architecture Decisions*, ASME 2001 Design Engineering Technical Conferences, Pittsburgh, PA, ASME, DETC2001/DAC-21110, 2001.
- [36] O.L. De Weck, E.S. Suh, and D. Chang, *Product Family and Platform Portfolio Optimization*, ASME Design Engineering Technical Conferences, DETC03/DAC-48721, Chicago, IL, USA, 2003.
- [37] P. Dewhurst and G. Boothroyd, *Design for Assembly: Automatic Assembly*, Machine Design, January 26, pp. 87-92, 1984.
- [38] P. De Lit, T.L'Eglise, and A. Delchambre, *Functional Entities: A Concept to Support Product Family and Assembly System Design*, Proc. IEEE Int. Symp. Assembly and Task Planning, Fukuoka, Japan, pp. 160-165, 2001.
- [39] X.H. Du, J.X. Jiao and M.M. Tseng, *Architecture of Product Family: Fundamental and Methodology*, Concurrent Engineering: Research and Application, **9**(4), pp. 309-325, 2001.
- [40] X.H. Du, J.X. Jiao and M.M. Tseng, *Graph Grammar Based Product family Modeling*, Concurrent Engineering: Research and Applications, **10**(2), pp. 113-128, 2002.
- [41] X.H. Du, J.X. Jiao and M.M. Tseng, *Product Family Modeling and Design Support: An Approach Based on Graph Rewriting Systems*, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, **16**(2), pp. 103-120, USA, 2002.
- [42] H.A. ElMaraghy and N. Mahmoudi, *Concurrent Design of Product Modules Structure and Global Supply Chain Configuration*, Supply Chain, Theory and Applications, I-Tech Education and Publishing, Vienna, Austria, pp. 558, 2008.

- [43] F.J. Erens and H.M.H. Hegge, *Manufacturing and Sales Co-ordination for Product Variety*, International Journal of Production Economics, **37**(1), pp. 83-99, 1994.
- [44] F.J. Erens and K. Verhulst, *Architecture of Product Family*, Computers in Industry, **33**(2-3), pp. 165-178, 1997.
- [45] A. Ericsson and G. Erixon, *Controlling Design Variants: Modular Product Platforms*, New York: ASME, 1999.
- [46] G. Erixon and B. Ostgren, *Synthesis and Evaluation Tool for Modular Designs*, International conference on engineering design, Hague, Netherlands, pp. 898-905, 1993.
- [47] W. Eversheim, J. Neuhausen and M. Seaterhenn, *Design to Cost for Production Systems*, Annual of the CIRP, **47**, pp. 357, 1998.
- [48] R. Farrell and T.W. Simpson, *Product Platform Design to Improve Commonality in Custom Products*, Journal of Intelligent Manufacturing, **14**(6), pp. 541-556, 2003.
- [49] E. Feitzinger and H.L. Lee, *Mass Customization at Hewlett - Packard: The Power of Postponement*, Harvard Business Review, **75**(1), pp. 116-121, 1997.
- [50] A. Felfernig, G. Friedrich and D. Jannach, *Conceptual Modeling for Configuration of Mass Customizable Products*, Artificial Intelligence in Engineering, **15**(2), pp. 165-176, 2001.
- [51] M. Ficko, I. Drstvensek, M. Brezocnik, J. Balic and B. Vaupotic, *Prediction of Total Manufacturing Costs for Stamping Tool on the Basis of CAD-Model of Finished Product*, J. Mater. Process. Technol., 164-165, pp. 1327-1335, 2005.

- [52] W. Finch, *Set-Based Models of Product Platform Design and Manufacturing Processes*, ASME design engineering technical conferences, DETC99/DTM-8763, Las Vegas, NV, 1999.
- [53] S.K. Fixson, *Linking Modularity and Cost: A Methodology to Assess Cost Implications of Product Architecture Differences to Support Product Design*, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2002.
- [54] S.K. Fixson, *Assessing Production Architecture Costing: Production Life Cycles, Allocation Rules, and Cost Models*, Proceedings of DETC04, Salt Lake City, September 28-October 2, 2004.
- [55] S.K. Fixson, *Product Architecture Assessment: A Tool to Link Product, Process, and Supply Chain Design Decisions*, Journal of Operations Management, **23**(3-4), pp. 345-369, 2005.
- [56] E. Freuder, *Constraint Solving Techniques*, Constraint Programming (B. Mayoh, E. Tyngu, and J. Penjaen eds.), Vol. 131 of Series F Computer and Systems Sciences, NATO ASI Series, pp. 51-74, 1992.
- [57] E. Freuder and D. Sabin, *Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction*, Proc. 14th Nat'l Conf. Artificial Intelligence, AAAI Press, pp. 191-196, 1997.
- [58] K. Fujita, *Product Variety Optimization Under Modular Architecture*, Comput. Aid. Des., **34**, pp. 953-965, 2002.
- [59] K. Fujita and K. Ishii, *Task Structuring towards Computational Approaches to Product Variety Design*, Proceeding of 1997 ASME Design Engineering Technical Conference, Sacramento, USA, 1997.

- [60] K. Fujita, H. Sakaguchi, and S. Akagi, *Product Variety Deployment and Its Optimization Under Modular Architecture and Module Commonalization*, ASME Design Engineering Technical Conferences, DETC99/DFM-8923, Las Vegas, NV, USA, 1999.
- [61] K. Fujita and H. Yoshida, *Product Variety Optimization Simultaneously Designing Module Combination and Module Attributes*, Concurrent Engineering: Research and Application, **12**(2), pp. 105-118, 2004.
- [62] D. Garwood, *Bills of Material: Structured for Excellence*, Dogwood Publishing Co., Marietta, GA, 1988.
- [63] A. Gayretli and H.S. Abdalla, *A Featured Based Prototype System for the Evaluation and Optimization of Manufacturing Processes*, Comput. Ind. Eng., **37**, pp. 481-484, 1999.
- [64] E. Gelle and B.V. Faltings, Solving Mixed and Conditional Constraint Satisfaction Problems, Constraints, **8**(2), pp. 107-141, 2003.
- [65] E. Gelle and M. Sabin, *Solving Methods for Conditional Constraint Satisfaction*, Proc. IJCAI Workshop on Configuration, affiliated with the 18th Int'l Joint Conf. Artificial Intelligence (IJCAI 03), pp. 7-12, 2003.
- [66] E. Gelle and M. Sabin, *Conditional Constraint Satisfaction and Configuration - A Win-Win Proposition*, IEEE Intelligent Systems, **22**(1), pp. 81-83, 2007.
- [67] J. Gershenson, G.J. Prasad, and Y. Zhang, *Product Modularity: Definitions and Benefits*. Journal of Engineering Design, **14**(3), pp. 295-313, 2003.
- [68] J.H. Gilmon and B.J. Pine, *The Four Faces of Mass Customization*, Harvard Business Review, **75**(1), pp. 91-101, 1997.
- [69] J.P. Gonzalez-Zugasti, K. Otto, and J. Baker, *A Method for Architecting Product Platforms*, Research in Engineering Design, **12**(2), pp. 61-72, 2000.

- [70] T. Gruber, *Towards Principles for the Design of Ontologies used for Knowledge Sharing*, International Journal of Human-Computer Studies, **44**, pp. 907-928, 1995.
- [71] A. Günter, *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995.
- [72] A. Haselböck, *Knowledge-Based Configuration and Advanced Constraint Technologies*, Ph.D Dissertation, Institut für Informations systeme, Technische Universität Wien, 1993.
- [73] W. Havens and A. Mackworth, *Representing knowledge of the visual world*, IEEE Computer, 16(10), pp. 90-96, 1993.
- [74] H.M.H. Hegge and J.C. Wortmann, *Generic Bill-of-Material: A New Product Model*, Int. J. Production Economics, **23**(1-3), pp. 117-128, 1991.
- [75] M. Heinrich and E.W. Jungst. *A Resource-Based Paradigm for The Configuring of Technical Systems from Modular Components*, Proc. Seventh IEEE Conference of Artificial Intelligence Applications, IEEE Press, Piscataway, N.J., USA, pp. 257-264, 1991.
- [76] G. Hernandez, J.K. Allen, G.W. Woodruff, T.W. Simpson, E. Bascaran, L.F. Avila, and F. Salinas, *Robust Design of Families of Products with Production Modeling and Evaluation*, ASME Journal of Mechanical Design, **123**(2), pp. 183-190, 2001.
- [77] A. Heylighen and H. Neuckermans. *A Case Base of Case-Based Design Tools for Architecture*, Computer Aided Design, **33**:1111, 2001.
- [78] T.H. Ho and C.S. Tang, *Product Variety Management: Research Advances*, Boston: Kluwer Academic Publishers, 1998.

- [79] K. Hölttä and M. Salonen, *Comparing Three Modularity Methods*, ASME Design Engineering Technical Conferences, DETC2003/DTM-48649, Chicago, IL, USA, 2003.
- [80] L. Hotz and T. Krebs, *Configuration - State of the Art and New Challenges*, Proceedings of 17. Workshop Planen, Scheduling und Konfigurieren, Entwerfen (PuK2003) - KI 2003 Workshop. pp. 145-157, 2003.
- [81] Y.L. Huang, *Machine Learning in Customer-Driven Product Configuration Based on Lifecycle Metrics*, Master Thesis, 2007.
- [82] V. Hubka and W.E. Eder, *Theory of Technical System*, Springer-Verlag, Berlin, Heidelberg, Germany, 1988.
- [83] M.S. Hundal, *Product Costing: A Comparison of Conventional and Activity-Based Costing Methods*. J. Eng. Des., **8**, pp. 91-103, 1997.
- [84] F. H'midaa, P. Martinb, F. Vernadat, *Cost Estimation in Mechanical Production: The Cost Entity Approach Applied to Integrated Product Engineering*, Int. J. Production Economics, **103**, pp. 17-35, 2006.
- [85] K. Ishii, B.H. Lee, and C.F. Eubanks, *Design for Product Retirement and Modularity Based on Technology Life-Cycle*, ASME Manufacturing Science and Engineering (E. Kannatey-Asibu Ed.), San Francisco, CA, USA, MED-Vol. 2-2/MH-Vol. 3-2, pp. 921-933, 1995.
- [86] J.X. Jiao, T.W. Simpson, and Z. Siddique, *Product Family Design and Platform-Based Product Development: A Start-of-the-Art Review*, Journal of Intelligent Manufacturing, **18**(1): 4-29, 2007.
- [87] J.X. Jiao and M.M. Tseng, *A Pragmatic Approach to Product Costing Based on Standard Time Estimation*, International Journal of Operations & Production Management, **19**(7), pp. 738-755, 1999.

-
- [88] J.X. Jiao and M.M. Tseng, *Fundamentals of Product Family Architecture*, Journal of Engineering Design, **11**(3), pp. 225-243, 2000.
- [89] J.X. Jiao and M.M. Tseng, *Customizability Analysis in Design for Mass Customization*. Computer-Aided Design, **36**(8), pp. 745-757, 2004.
- [90] J.X. Jiao, M.M. Tseng, V.G. Duffy and F. Lin, Product Family Modeling for Mass Customization. Computers and Industrial Engineering, **35**(3-4), pp. 495-498, 1998.
- [91] J.X. Jiao, M.M. Tseng, Q. Ma, and Y. Zou, *Generic bill-of-materials and operations for high-variety production management*, Concurrent Engineering: Research and Applications, **8**(4), pp. 297-322, 2000.
- [92] J.Y. Jung, *Manufacturing Cost Estimation for Machined Parts Based on Manufacturing Features*, J. Intell. Manuf., **13**(4), pp. 227-238, 2002.
- [93] A.K. Kamrani and R. Gonzalez, *Agnetic Algorithm-Based Solution Methodology for Modular Design*, Journal of Intelligent Manufacturing, **14**(6), pp. 599-616, 2003.
- [94] K. Kim and D. Chhajed, *An Experimental Investigation of Valuation Change due to Commonality in Vertical Product Line Extension*, Journal of Product Innovation Management, **18**(4), pp. 219-230, 2001.
- [95] F. Kimura, S. Kato, T. Hata, and T. Masuda, *Product Modularization for Parts Reuse in Inverse Manufacturing*, CIRP Annals, **50**(1), pp. 89-92, 2001.
- [96] R. Klein, A Logic-Based Description of Configuration: The Constructive Problem Solving Approach, Configuration-Papers from the 1996 Fall Symp., Tech. Report FS-96-03, AAAI Press, pp. 1-10, 1996.

- [97] N. Kohlhase and H. Birkhofer, *Development of Modular Structure: the Prerequisite for Successful Modular Products*, Journal of Engineering Design, **7**, pp. 279-291, 1996.
- [98] T. Kokeny, *A New Arc-Consistency Algorithm for CSPs with Hierarchical Domains*, In Workshop Notes of the ECAI'94, Workshop on Constraint Satisfaction Issues Raised by Practical Applications, 1994.
- [99] V. Kumar, *Algorithms for Constraint Satisfaction Problems: A Survey*, AI Magazine, **13**(1), pp. 32-44, 1992.
- [100] A. Kusiak, *Integrated Product and Process Design: A Modularity Perspective*, Journal of Engineering Design, **13**(1), pp. 223-231, 2002.
- [101] A. Kusiak and C.C. Huang, *Development of Modular Products*, IEEE Transactions on Components, Packaging, and Manufacturing Technology. Part A, **19**(4), pp. 523-538, 1996.
- [102] A. Layer, E.T. Brinke, F. Van Houten, H. Kals and S. Haasis, *Recent and Future Trends in Cost Estimation*, INT. J. Computer Integrated Manufacturing, **15**(6), pp. 499-510, 2002.
- [103] H.J. Lee and J.K. Lee, *An Effective Customization Procedure With Configuration Standard models*, Decision Support Systems, **41**, pp. 262-278, 2005.
- [104] J.X. Li, *A Novel Approach to Computer-Aided Configuration Design Based on Constraint Satisfaction Paradigm*, Master Thesis, Univ. Saskatchewan, Saskatoon, CA, 2005.
- [105] W.Y. Liang and C.C. Huang, *Agent-Based Collaboration Information System of Product Development*, International Journal of Information Management, **22**(3), pp. 211-224, 2002.

- [106] E.A.M. Lotfy and A.S. Mohamed, *Applying Neural Networks in Case-Based Reasoning Adaptation for Cost Assessment of Steel Buildings*, International Journal of Computers & Applications, **24**(1), pp. 28-38, 2002.
- [107] A. Mackworth, J. Mulder, and W. Havens, *Hierarchical Arc Consistency: Exploiting Structured Domains in Constraint Satisfaction Problems*, Computational Intelligence, 1, pp. 118-126, 1985.
- [108] J. N. Madsen, *Methods for Interactive Constraint Satisfaction*, Master's thesis, Departure of computer Science, University of Copenhagen, Danmark, 2003.
- [109] J. Malmqvist, *Towards Computational Design Methods for Conceptual and Parametric Design*, Ph.d Thesis, Departure of Machine and Vehicle Design, Chalmers University of Technology, Goteborg, Sweden, 1993.
- [110] M.V. Martin and K. Ishii, *Design for Variety: Development of Complexity Indices and Design Charts*, ASME design engineering technical conferences, DFM-4359, Sacramento, CA, USA, 1997.
- [111] M.V. Martin and K. Ishii, *Design for Variety: Developing Standardized and Modularized Product Platform Architectures*, Research in Engineering Design, **13**(4), pp. 213-235, 2002.
- [112] J.A. Martinez-Larrosa and Z. Siddique, *CAD Support for Product Family Design Using Parametrics, Mating relationships, and Modularity*, Advances in Concurrent Engineering (R. Jardim-Gonçalves, R. Roy, & A. Steiger-Garção Eds.), pp. 535-543. Lisse: A.A. Balkema Publishers, 2002.
- [113] C.A. Mattson and S.P. Magleby, *The Influence of Product Modularity During Concept Selection of Consumer Products*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC2001/DTM-21712, 2001.

- [114] D.A. McAdams and K.L. Wood, *A Quantitative Similarity Metric for Design-by-Analogy*, ASME Journal of Mechanical Design, **124**(2), pp. 173-182, 2002.
- [115] J. McDermott, *R1: A Rule-Based Configurer of Computer Systems*, Artificial Intelligence, **19**(1), pp. 39-88, 1982.
- [116] D.L. McGuinness and A.T. Borgida. *Explaining Subsumption in Description Logics*, Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95), Montréal, Québec, Canada, **1**, pp. 816-821, 1995.
- [117] M.E. McGrath, *Product Strategy for High-Technology Companies*. New York: Irwin Professional Publishing, 1995.
- [118] A. McKay, F.J. Erens and M.S. Broor, *Relating Product Definition and Product Variety*, Research in Engineering Design, **2**(2), pp. 63-80, 1996.
- [119] A. Messac, M.P. Martinez, and T.W. Simpson, *A Penalty Function for Product Family Design Using Physical Programming*, ASME Journal of Mechanical Design, **124**(2), pp. 164-172, 2002.
- [120] M.H. Meyer and A.P. Lehnerd, *The Power of Product Platform - Building Value and Cost Leadship*, New York: Free Press, 1997.
- [121] M.H. Meyer, P. Tertzakian, and J.M. Utterback, *Metrics for Managing Research and Development in the Context of the Product Family*, Management Science, **43**(1), pp. 88-111, 1997.
- [122] J.H. Mikkola and O. Gassmann, *Managing Modularity of Product Architectures: Towards an Integrated Theory*, IEEE Transactions on Engineering Management, **50**(2), pp. 204-218, 2003.
- [123] S. Mittal and B. Falkenhainer, *Dynamic Constraint Satisfaction Problems*, Proceedings of the 8th National Conference on Artificial Intelligence, AAAI Press, pp. 25-32, 1990.

- [124] S. Mittal and F. Frayman, *Towards a Generic Model of Configuration Tasks*, Proc. 11th Int. Joint Conf. Of Artificial Intelligence, Sanfrancisco, USA, pp. 1395-1401, 1989.
- [125] T. Mänistö, *A Conceptual Modeling Approach to Product Families and Their Evolution*, Ph.D thesis, Helsinki University of Technology, Acta Polytechnica Scandinavica, Mathematics and Computing Series, No. 106, Espoo, 2000.
- [126] W.L. Moore, J.J. Louviere and R. Verma, *Using Conjoint Analysis to Help Design Product Platforms*, Journal of Product Innovation Management, **16**(1), pp. 27-39, 1999.
- [127] M. Muffatto, *Introducing a platform strategy in product development*, International Journal of Production Economics 60-61, pp. 145-153, 1999.
- [128] M. Muffatto and M. Roveda, *Product Architecture and Platforms: A Conceptual Framework*, International Journal of Technology Management, **24**(1), pp. 1-16, 2002.
- [129] J. Nanda, H.J. Thevenot, and T.W. Simpson, *Product Family Design Knowledge Representation, Integration, and Reuse*, Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration, pp. 32-37, 2005.
- [130] K. Naughton, E. Thornton, K. Kerwin, and H. Dawley, *Can Honda Build a World Car?* Business Week, **100**(7), 1997.
- [131] R. Nayak, W. Chen, and T.W. Simpson, *A Variation-Based Method for Product Family Design*, Engineering Optimization, **34**, pp. 65-81, 2002.
- [132] P.J. Newcomb, B. Bras, and D.W. Rosen, *Implication of Modularity on Product Design for a Life Cycle*, proceeding of ASME Design Engineering Technical Conference, DET96/DTM-1516, Irvine, CA, USA, 1996.

- [133] P.J. Newcomb, B. Bras, and D.W. Rosen, *Implications of Modularity on Product Design for the Life Cycle*, ASME Journal of Mechanical Design, **120**(3), pp. 483-490, 1998.
- [134] A. Niazi, J.S. Dai, S. Balabani, and L. Seneviratne, *Product Cost Estimation: Technique Classification and Methodology Review*, Journal of Manufacturing Science and Engineering, **128**, pp. 563-575, 2006.
- [135] K. Otto and K. Hölttä, *A Multi-Criteria Framework for Screening Preliminary Product Platform Concepts*,
- [136] K. Otto, J. Gonzalez-Zugasti, and J. Dahmus, *Modular Product Architecture*, ASME Design Engineering Technical Conferences, DETC2000/DTM-4565, Baltimore, MD, USA, 2000.
- [137] K. Otto and K. Wood, *Product design: Techniques in reverse engineering and new product development*, Upper Saddle River, NJ: Prentice Hall, 2001.
- [138] M. Özbayraka, M. Akgünb, and A. K. Türker, *Activity-based cost estimation in a push/pull advanced manufacturing system*, Int. J. Production Economics 87, 49-65, 2004.
- [139] G. Pahl and W. Beitz, *Engineering Design: a Systematic Approach*, Springer, London, 1996.
- [140] E.J. Park, L.F. da Luz, and A. Suleman, *Multidisciplinary Design Optimization of an Automotive Magnetorheological Brake Design*, Computers and Structures, **86**(3-5), pp. 207-216, 2008.
- [141] J. Park and T.W. Simpson, *Development of a Production Cost Estimation Framework to Support Product Family Design*, International Journal of Production Research, **43**(4), pp. 731-772, 2005.

- [142] M.W. Pessina and J.R. Renner, *Mass Customization at Lutron Electronics - A total Company Process*. Agility & Global Competition, **2**(2), pp. 50-57, 1998.
- [143] B.J. Pine, *Mass customization: The new frontier in business competition*, Boston: Harvard Business School Press, 1993.
- [144] S.J. Phua, W.K. Ng, H.F. Liu, X. Li, and B. Song, *Customer Information System for Product and Service Management: Towards Knowledge Extraction from Textual and Mixed-Format Data*, 4th IEEE International Conference on Service Systems and Service Management (ICSSSM2007), Chengdu, China, June 9-11, 2007.
- [145] Qin Y.H. and Wei G.X. *Product Configuration Based on CBR and CSP*, Int. Conf. on Measuring Technology and Mechatronics Automation, Hunan, China, pp. 681-684, 2009.
- [146] K.C. Ranze, T. Scholz, T. Wagner, A. Günter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt, *A Structure-based Configuration Tool: Drive Solution Designer DSD*, 14th Conf. Innovative Applications of AI, 2002.
- [147] I. Razgon, *Complexity Analysis of Heuristic CSP Search Algorithms*, Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic programming, Sweden, pp. 88-99, June, 2005.
- [148] S. Rehman and M.D. Guenov, *A Methodology for Modeling Manufacturing Costs at Conceptual Design*, Comput. Ind. Eng., **35**(3-4), pp. 623-626, 1998.
- [149] D. Robertson and K. Ulrich, *Planning Product Platforms*, Sloan Management Review, **39**(4), pp. 19-31, 1998.
- [150] K. Sabbagh, *Twenty-First Century Jet: The Making and Marketing of the Boeing 777*, New York: Scribner, 1996.

- [151] D. Sabin and E.C. Freuder, *Configuration as Composite Constraint Satisfaction*, Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop (G.F. Luger Ed.), AAAI Press, pp. 153-161, 1996.
- [152] D. Sabin and E.C. Freuder, *Understanding and Improving the MAC Algorithm*, Proc. 3rd Int'l Conf. Principles and Practice of Constraint Programming, Springer-Verlag, Heidelberg, Germany, pp. 167-181, 1997.
- [153] D. Sabin and E.C. Freuder, *Modeling and Solving Configuration Tasks with ILOG Solver*, Proc. 3rd Int'l Users Meeting, ILOG Optimization Suite, ILOG, Mountain View, Calif., USA, 1997.
- [154] M. Sabin and E.C. Freuder, *Detecting and Resolving Inconsistency and Redundancy in Conditional Constraint Satisfaction Problems*, Web-published papers of the CP'98 Workshop on Constraint Problem Reformulation, Pisa, Italy, 1998.
- [155] D. Sabin and R. Weigel, *Product Configuration Frameworks - a Survey*, IEEE Intelligent Systems, pp. 42-49, 1998.
- [156] M.A. Salido, *A Non-Binary Constraint Ordering Heuristic for Constraint Satisfaction Problems*, Applied Mathematics and Computation, **1981**, pp. 280-295, 2008.
- [157] S.W. Sanderson, *Cost Models for Evaluating Virtual Design Strategies in Multicycle Product Families*, Journal of Engineering and Technology Management, **8**, pp. 339-358, 1991.
- [158] S.W. Sanderson and M. Uzumeri, *Managing Product Families*, Chicago: Irwin, 1997.
- [159] M.S. Sawhney, *Leveraged High-Variety Strategies: From Portfolio Thinking to Platform Thinking*. Journal of the Academy of Marketing Science, **26**(1), pp. 54-61, 1998.

- [160] D. Searls and L. Norton, *Logic Based Configuration with a Semantic Network*, Journal of Logic Programming, **8**(1-2), pp. 53-73, 1990.
- [161] E.M. Shehab and H.S. Abdalla, *A Design to Cost System for Innovative Product Development*, Proc. Inst. Mech. Eng., Part B, **216**(7), pp. 999-1019, 2002.
- [162] W. Shen, D.H. Norrie, and J.P.A. Barthès, *Multi-Agent Systems for Concurrent Intelligent Design and Manufacturing*, New York: Taylor & Francis, 2001.
- [163] Z. Siddique, *Estimating Reduction in Development Time for Implementing A Product Platform Approach*, ASME Design Engineering Technical Conferences, DETC2001/CIE-21238, Pittsburgh, PA, USA, 2001.
- [164] Z. Siddique and D.W. Rosen, *Product Platform Design: A Graph Grammar Approach*, ASME design engineering technical conferences, DETC99/DTM-8762, Las Vegas, NV, USA, 1999.
- [165] Z. Siddique, D.W. Rosen, and N. Wang, *On the Applicability of Product Variety Design Concepts to Automotive Platform Commonality*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC980/DTM-5661, 1998.
- [166] Z. Siddique and Z. Shao, *A Graph Grammar Based Approach for Development of an Internet-Based Product Family Information System*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC2001/CIE-21280, 2001.
- [167] Z. Siddique and Y.J. Zhou, *Automatic Generation of Product Family Member CAD Models Supported by a Platform Using A Template Approach*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC2002/CIE-34407, 2002.
- [168] G.D. Silverira, D. Borenstein, and F.S. Fogliatto, *Mass Customization: Literature Review and Research Directions*, Int. J. Production Economics, **72**, pp. 1-13, 2001.

- [169] H. Simonis, *Models for Global Constraint Applications*, Constraints, **12**, pp. 63-92, 2007.
- [170] T.W. Simpson, *Product Platform Design and Customization: Status and Promise*, AIEDAM, **18**(1), pp. 3-20, 2004.
- [171] T.W. Simpson and B. D'Souza, *Assessing Variable Levels of Platform Commonality within a Product Family Using a Multiobjective Genetic Algorithm*, Concurrent Engineering: Research and Applications, **12**(2), pp. 119-130, 2004.
- [172] T.W. Simpson, J.R.A. Maier, and F. Mistree, *A Product Platform Concept for Product Family Design*, Proceeding of 1999 ASME Design Engineering Technical Conference, DETC/DTM-8761, Las Vegas, Nevada, USA, 1999.
- [173] T.W. Simpson, J.R.A. Maier, and F. Mistree, *Product Platform Design: Method and Application*, Research in Engineering Design, **13**(1), pp. 2-22, 2001.
- [174] T.W. Simpson, Z. Siddique, and J.X. Jiao, *Product Platform and Product Family Design: Methods and Applications*, New York: Springer, 2005.
- [175] C. Sinz and A. Haag, *Configuration*, IEEE Intelligent Systems, **22**(1), pp. 78-90, 2007.
- [176] M.E. Sosa, S.D. Eppinger and C.M. Rowles, *The Misalignment of Product Architecture and Organizational Structure in Complex Product Development*, INSEAD Working Paper, 2003/68/TM, 2003.
- [177] S.N. Spitz and A.A.G. Requicha, *Hierarchical Constraint Satisfaction for High-Level Dimensional Inspection Planning*, proceeding of the 1999 IEEE, International Symposium on Assembly and Task Planning, Porto, Portugal, July, 1999.
- [178] B.M. Stein, *Functional Models in Configuration Systems*, Ph.D. dissertation, Universität at Paderborn, 1995.

- [179] R.B. Stone, K.L. Wood, and R.H. Crawford, *A Heuristic Method to Identify Modules from a Functional Description of a Product*, Design Studies, **21**(1), pp. 5-31, 2000.
- [180] M. Stumptner and A. Haselböck, *A Generative Constraint Formalism for Configuration Problems*, Third Congress of Italian Association for Artificial intelligence, pp. 302-13, Torino, Italy, 1993.
- [181] M. Stumptner, A. Haselböck, and G. Friedrich, *COCOS - A Tool for Constraint-Based, Dynamic Configuration*, Proceedings of the 10th Conference on Artificial Intelligence for Applications, San Antonio, TX, USA, pp. 373-380, 1-4 Mar., 1994.
- [182] N.P. Suh, *Axiomatic Design: Advances and Applications*, New York: Oxford University Press, 2001.
- [183] N. Sundgren, *Introducing Interface Management in Product Family Development*, Journal of Production Innovation Management, **16**(1), pp. 40-51, 1999.
- [184] T. Suzue and A. Kohdate, *Variety Reduction Program: A Production Strategy for Product Diversification*, Cambridge: Productivity Press, 1990.
- [185] P.N. Tan, M. Steinback, and V. Kumar, *Chapter 6 Association Analysis: Basic Concepts and Algorithms, Introduction of Data Mining*, Pearson Education, Inc.
- [186] J. Toussaint and K. Cheng, *Web-Based CBR (Case-Based Reasoning) As a Tool with the Application to Tooling Selection*, International Journal of Advanced Manufacturing Technology, **29**, pp. 24-34, 2006.
- [187] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London, 1993.
- [188] M.M. Tseng and J. Jiao, *Design for Mass Customization*, CIRP Annuals, **45**(1), pp. 153-156, 1996.

- [189] M.M. Tseng and J. Jiao, *Case-Based Evolutionary Design for Mass Customization*, Computers in Industrial Engineering, **33**(1-2), pp. 319-323, 1997.
- [190] E. Uhlmann and K. Schauer, *Dynamic Load and Strain Analysis for the Optimization of Micro End Mills*, CIRP Annals-Manufacturing Technology, **54**(1), pp. 75-78, 2005.
- [191] K. Ulrich, *The Role of Product Architecture in the Manufacturing Firm*, Research Policy, **24**, pp. 419-440, 1995.
- [192] K. Ulrich and S.D. Eppinger, *Product Design and Development*, New York, McGraw-Hill, 1995.
- [193] K. Ulrich and K. Tung, *Fundamental of Product Modularity*, Sharon, A. (Ed.) Issues in Mechanical Design International 1991, ASME, New York, DE-39, pp. 73-79, 1991.
- [194] E.A. van Veen, *Modeling Product Structures by Generic Bill of Material*, Elsevier Science Publishers, Amsterdam, 1992.
- [195] M.J. Van Wie, J.L. Greer, M.I. Campbell, R.B. Stone, and K.L. Wood, *Interfaces and Product Architecture*, ASME Design Engineering Technical Conferences, DETC01/DTM-21689, Pittsburgh, PA, USA, 2001.
- [196] M.J. Van Wie, R.B. Stone, H. Thevenot, and T. Simpson, *Examination of Platform and Differentiating Elements in Product Family Design*, Journal of Intelligent Manufacturing, **18**(1):77-96, 2007.
- [197] B. Wang and E. Antonsson, *Information Measure for Modularity in Engineering Design*, ASME Design Engineering Technical Conference, DETC2004/DTM-57515, 2004.

- [198] L. Wang, B. Song, X. Li, and W.K. Ng, *A Product Family Based Life Cycle Cost Model for Part Variety and Change Analysis*, Int. Conf. on Engineering Design, ICED'07, Paris, France, CD-ROM, 2007.
- [199] *Constraint Satisfaction Problems*, <http://aima.cs.berkeley.edu/newchap05.pdf>.
- [200] *Choco. A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving*, URL: <http://sourceforge.net/projects/choco>, 2005.
- [201] *Drool Rule Language*, <http://labs.jboss.com/drools>.
- [202] *Ilog. Solver 5.0 User's Manual*, URL: <http://www.ilog.fr>, 2000.
- [203] *Jess Rule Language and Rule Engine*, <http://herzberg.ca.sandia.gov>.
- [204] S.C. Wheelwright and K.B. Clark, *Leading Product Development: the Senior Manager's Guide to Creating and Shaping the Enterprise*, New York: Free Press, 1995.
- [205] D.E. Whitney, *Physical Limits to Modularity*, Working paper, ESD-WP-2003-01.03-ESD, Massachusetts Institute of Technology, 2003.
- [206] B. Wilhelm, *Platform and Modular Concepts at Volkswagen - Their Effects on the Assembly Process*. Transforming Automobile Assembly: Experience in Automation and Work Organization (K. Shimokawa, U. Jürgens, and T. Fujimoto Eds.), pp. 146-156, New York: Springer, 1997.
- [207] K. Willcox and S. Wakayama, *Simultaneous Optimization of a Multiple-Aircraft Family*, Journal of Aircraft, **40**(4), pp. 616-622, 2003.
- [208] H.C. Wortmann and F.J. Erens, *Control of Variety by Generic Product Modeling*, Proceeding of The First World Congress on Intelligent Manufacturing Processes and Systems, University of Puerto Rico, Part, **2**, pp. 1327-1342, San Juan, Puerto Rico, 1995.

- [209] J.R. Wright, D.L. McGuinness, C. Foster, and G.T. Vesonder, *Conceptual Modeling Using Knowledge Representation: Configurator Applications*, Proc. Artificial Intelligence in Distributed Information Networks Workshop, IJCAI-95, Montreal, Canada, August, 1995.
- [210] J.R. Wright, E.S. Weixelbaum, G.T. Vesonder, K.E. Brown, S.R. Palmer, J.I. Berman, and H.H. Moore, *A Knowledge-Based Configurator That Supports Sales, Engineering, and Manufacturing at AT&T Network Systems*, Proc. Innovative Applications of Artificial Intelligence Conf., AAAI Press, pp. 183-193, 1993.
- [211] W.X. Wu and M.S. Hsiao, *Mining Global Constraints for Improving Bounded Sequential Equivalence Checking*, Proceedings of the 43rd Annual Conference on Design Automation (DAC 2006), San Francisco, CA, USA, July 24-28, 2006.
- [212] H. Xie, P. Henderson, and M. Kernahan, *Modelling and Solving Engineering Product Configuration Problems by Constraint Satisfaction*. International Journal of Production Research, **43**20, pp. 4455-4469, 2005.
- [213] H. Xie and M. Kernahan, *Product Configurator Technology and Its Applications*, The 5th International Workshop on Advanced Manufacturing Technologies, London, Canada, pp. 1-6, 2005.
- [214] X. Xu, J.L.Q. Chen, and S.Q. Xie, *Framework of a Product Lifecycle Costing System*, Journal of Computing and Information Science in Engineering, **6**, pp. 69-77, 2004.
- [215] A.S. Yigit, A.G. Galip-Ulsoy, and A. Allahverdi, *Optimizing Modular Product Design for Reconfigurable Manufacturing*, Journal of Intelligent Manufacturing, **13**(4), pp. 309-316, 2002.
- [216] G.R. Yost and T.R. Rothenfluh, *Configuring elevator systems*, Int. J. Human-Computer Studies, **44**, pp. 521-568, 1996.

- [217] T.L. Yu, A.A. Yassine, and D.E. Goldberg, *Agnetic Algorithm for Developing Modular Product Architectures*, ASME Design Engineering Technical Conferences, Chicago, Illinois, USA, 2003.
- [218] E.J. Zamirowski and K.N. Otto, *Identifying Product Portfolio Architecture Modularity Using Function and Variety Heuristics*, In ASME design engineering technical conferences, DETC99/DTM-8760, Las Vegas, NV, 1999.
- [219] Y.F. Zhang, J.Y.H. Fuh and W.T. Chan, *Feature-Based Cost Estimation for Packaging Products Using Neural Networks*, Comput Ind., **32**, pp. 95-113, 1996.
- [220] Y. Zhang, J.K. Gershenson, and S. Allamneni, *Determining Relationships Between Modularity and Cost in Product Retirement*, Proc. ASME Design Engineering Technology Conf., Paper No. DETC2001/DTM-21686, 2001.