DYNAMIC CONSTRAINT SATISFACTION ALGORITHM FOR RECONFIGURATION OF FEATURE MODELS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

SINA ENTEKHABI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2018

Approval of the thesis:

DYNAMIC CONSTRAINT SATISFACTION ALGORITHM FOR RECONFIGURATION OF FEATURE MODELS

submitted by **SINA ENTEKHABI** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver ____________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Halit Oğuztüzün ____________
Head of Department, **Computer Engineering**

Prof. Dr. Halit Oğuztüzün ____________
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Ferda Nur Alpaslan
Computer Engineering Department, METU ____________

Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU ____________

Assist. Prof. Dr. Ahmet Çevik
Gendarmerie and Coast Guard Academy ____________

Date: 24.01.2018

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SINA ENTEKHABI

Signature :

ABSTRACT

DYNAMIC CONSTRAINT SATISFACTION ALGORITHM FOR RECONFIGURATION OF FEATURE MODELS

Entekhabi, Sina
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Halit Oğuztüzün

January 2018, 50 pages

Dynamically reconfigurable systems are able to respond to changes in their operational environments by reconfiguring themselves automatically. Dynamic software product lines are dynamically reconfigurable systems with an explicit variability model that guides the reconfiguration. In this work, feature models are used as the variability model. Features are assumed to be mapped to system's components that realize them. A feature model corresponds to a constraint satisfaction problem (CSP), and determines the valid configurations of the system. An emerging situation in the environment can lead to some relevant changes to the current configuration: some features must be activated, and some must be deactivated. Due to constraint propagation, the status of other features might need to be changed as well. However, considering the feature state migration costs, one would like to avoid such changes to the greatest extent possible in order to mitigate the cost of the disruption to the system's operation. In this work, we devised a dynamic constraint satisfaction algorithm that efficiently

considers feature state changes to be applied to the current configuration while confronting changes in the environment so that the new configuration will be valid and the requirements of the new situation will be satisfied with the minimum cost. A set of heuristics regarding feature model relations and the overall structure of feature models are also proposed to enhance the efficiency of the algorithm.

Keywords: Dynamic Reconfiguration, Feature Model, Dynamic Algorithm, Dynamic Constraint Satisfaction Problem

ÖZ

ÖZELLİK MODELLERİNİN YENİDEN YAPILANDIRMASI İÇİN DİNAMİK KISIT SAĞLAMA ALGORİTMASI

Entekhabi, Sina
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Halit Oğuztüzün

Ocak 2018 , 50 sayfa

Dinamik olarak yeniden yapılandırılabilir sistemler kendilerini otomatik olarak yeniden yapılandırarak operasyonel ortamlarındaki değişikliklere yanıt verebilirler. Dinamik yazılım ürün hatları, yeniden yapılandırmaya kılavuzluk eden açık değişkenlik modeline sahip dinamik olarak yeniden yapılandırılabilir sistemlerdir. Bu çalışmada, değişkenlik modeli olarak özellik modelleri kullanılmaktadır. Özelliklerin, onları gerçekleştiren sistemin bileşenleri ile eşleştirildiği varsayılmaktadır. Bir özellik modeli bir kısıt sağlama problemine (KSP) karşılık gelir ve sistemin geçerli yapılandırmalarını belirler. Ortamda ortaya çıkan bir durum mevcut yapılandırmayla ilgili bazı değişikliklerin yapılmasına yol açabilir: bazı özellikler etkinleştirilmeli ve bazıları devre dışı bırakılmalıdır. Kısıt yayılımı nedeniyle, diğer özelliklerin durumunun da değiştirilmesi gerekebilir. Bununla birlikte, özellik durumu taşıma maliyetleri göz önüne alındığında, sistemin çalışmasının aksama maliyetini azaltmak için bu tür değişikliklerin mümkün mertebe

önlenmesini isteriz. Bu çalışmada, ortamdaki değişikliklere karşı koyarken özellik durum değişikliklerinin mevcut yapılandırmaya etkili bir şekilde uygulanmasını sağlayan dinamik bir kısıt sağlama algoritması geliştirdik; böylece yeniden yapılandırma geçerli olacak ve yeni durumun gereksiminleri minimum maliyetle karşılanacaktır. Ayrıca, özellik modeli ilişkileri ve özellik modellerinin genel yapısı hakkında bir dizi buluşsal yöntem de algoritmanın verimliliğini artırmak amacıyla önerilmektedir.

Anahtar Kelimeler: Dinamik Yeniden Yapılandırma, Özellik Modeli, Dinamik Algoritma, Dinamik Kısıt Sağlama Problemi

To my dear parents

Mohammad Ali Entekhabi, Nahideh Mohammadi

ACKNOWLEDGMENTS

Thanks to Allah Almighty Who gave me the opportunity to be among wise and noble people to practice being a better kind of myself.

First and foremost, I would like to submit my eternal appreciation toward my parents and my family. All stages of my life are adorned with your unconditional support and patience. You have tried far more than myself to me to step forward and get closer to the ultimate destination of humanity. Words are not capable of expressing the deep respect of mine to you. I only can settle for saying thank you. It is you the true owners of this work.

I must give my highest gratitude to my supervisor Prof. Dr. Halit Oğuzüzün for all of his guidance and support as well. It was my great honor to work with him for about two years. His wise counsels along with his kindness and gentleness helped me to deal with confronted issues much easier. Serkan Karataş was also one of the persons who helped me the most during this work. I appreciate him very much and submit my respectful gratitude to him as well. I also would like to thank Prof. Dr. Ali Doğru for all of his avuncular characteristic and nice behavior. My contact with him introduced me elegant meanings of modesty and generosity, and I feel proud of myself to have cooperation with him from the beginning of my graduate program.

This work was funded by TÜBİTAK-ARDEB-1001 program under project 215E188 and I should thank TÜBİTAK for supporting us in fulfilling our goals during our study.

There have also been a lot of people who influenced me and helped me a lot to accomplish my plans and properly finish this work, especially Alper Karamanlıoğlu who helped me to clearly translate some parts of this work in Turkish very well. It is not possible to write down the name of all of them and explain how

much they helped me and how much they are important to me, because it may take more space than the work itself. Again, I have to settle for saying thank you and wish the bests for all of them.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xii

LIST OF TABLES xiv

LIST OF FIGURES xvi

LIST OF ABBREVIATIONS xviii

CHAPTERS

1 INTRODUCTION 1

1.1 Software Product Line 1

1.1.1 Feature Modeling 1

1.1.2 Dynamic Software Product Line 2

1.2 Motivation 4

1.3 Problem Definition 8

1.4 Related Work 11

2 ALGORITHM 15

2.0.1 Finding a first solution 19
2.0.2 Finding an optimum solution 21
2.1 Heuristics 22
2.1.1 Restricting distribution of variable values . . . 23
2.1.2 Satisfaction Priority of Constraints 25
2.1.3 OR and Alternative constraints satisfaction . . 26
2.1.3.1 Value distribution toward the leaf features with fewer unsatisfied Require and Exclude constraints 26
2.2 Soundness and completeness of the algorithm 27
2.2.1 Soundness 28
2.2.2 Completeness 28
2.2.3 completeness with heuristics 29
2.3 Tracing Example 30
3 EXPERIMENTS 39
3.1 Multi-directional methods 39
3.2 Execution results 40
4 CONCLUSION 45
REFERENCES 47

LIST OF TABLES

TABLES

Table 1.1 Condition-resolution table for the Collision Avoidance System shown in Figure 1.6 7

Table 1.2 Feature state migration time cost for the feature model of Collision Avoidance System shown in Figure 1.6 8

Table 1.3 Cost of feature's state migration cost 10

Table 2.1 Definition of constraints in the constraint system of the Figure 2.1 31

Table 3.1 Multi-directional methods for the constraint X = Y ∧ Y . . . 40

Table 3.2 Dynamic feature model reconfigurations using different algorithms in a feature model including 60 features and 40 feature model relations for requests including 3 features[1] 41

Table 3.3 Dynamic feature model reconfigurations using different algorithms in a feature model including 80 features and 47 feature model relations for requests including 5 features[1] 42

Table 3.4 Dynamic feature model reconfigurations using different algorithms in a feature model including 100 features and 65 feature model relations for requests including 7 features[1] 42

Table 3.5 Dynamic feature model reconfigurations using different algorithms in a feature model including 120 features and 69 feature model relations for requests including 9 features[1] 42

Table 3.6 The average results of executing algorithms on the sample feature models 43

LIST OF FIGURES

FIGURES

Figure 1.1 The feature model diagram of a system 2

Figure 1.2 Basic feature model relations and their description 3

Figure 1.3 a DSPL configuration of the system shown in Figure 1.1 . . . 4

Figure 1.4 Model-based reconfiguration process[10]. MoRE translates contextual changes into changes in the activation/deactivation of features. 5

Figure 1.5 Basic feature model relations and their description 6

Figure 1.6 The feature model and current configuration of a collision avoidance system 7

Figure 2.1 Mapping the feature model shown in Figure 1.3 to a constraint network 30

Figure 2.2 Starting variable value distribution in the ICSP shown in Figure 2.1 by calling FindConfiguration procedure to activate C and deactivate M. 32

Figure 2.3 Satisfying C4 by calling Solve procedure for the ICSP shown in Figure 2.2, where C_u=C4, V_f = (C, E, F, G, M), V_x= (C, E, M). C4 is decided to be satisfied by Activating E and deactivating F and G. 32

Figure 2.4 Satisfying C8 by calling Solve procedure for the ICSP shown in Figure 2.3, where C_u=C8, V_f = (C, E, F, G, H, M), and V_x= (C, E, H, M). Satisfying C8 by activating H will result in an inconsistent constraint network. 33

Figure 2.5 Satisfying C8 by calling Solve procedure for the ICSP shown in Figure 2.3, where C_u=C8, V_f = (C, E, F, G, I, M), and V_x= (C, E, I, M). C8 is decided to be satisfied by activating variable I. . . . 34

Figure 2.6 Satisfying C12 by calling Solve Procedure for the ICSP shown in Figure 2.5, where C_u=C12, V_f = (C, E, F, G, I, M, N), and V_x= (C, E, I, M, N). C12 is decided to be satisfied by deactivating variable N. 34

Figure 2.7 Satisfying C10 by calling Solve procedure for the ICSP shown in Figure 2.6, where C_u=C10, V_f = (B, C, E, F, G, I, M, N), and V_x= (B, C, E, I, M, N). C10 is decided to be satisfied by deactivating variable B. 35

Figure 2.8 Satisfying C4 by calling Solve procedure for the ICSP shown in Figure 2.2 , where C_u=C4, V_f = (C, E, F, G, M), and V_x= (C, G, M). C4 is decided to be satisfied by Activating G and deactivating F and E. 36

Figure 2.9 Satisfying C4 by calling Solve procedure for the ICSP shown in Figure 2.2, where C_u=C4, V_f = (C, E, F, G, M), and V_x= (C, E, M). C4 is decided to be satisfied by Activating F and deactivating G and E. 37

Figure 2.10 The new configuration of the system shown in Figure 1.3 after responding to the request to activate C and deactivate M with the minimum reconfiguration cost 38

LIST OF ABBREVIATIONS

FM	Feature Model
CSP	Constraint Satisfaction Problem
DCSP	Dynamic Constraint Satisfaction Problem
SPL	Software Product Line
DSPL	Dynamic Software Product Line

CHAPTER 1

INTRODUCTION

1.1 Software Product Line

Having a software product, a similar product would be developed much easier by reusing the resources of the existent product. Changing the code of previous product can be challenging, but it is mostly preferable than developing a product from scratch. Reusing a software product can be facilitated if reusing is considered from the early stages of developing the family of products. Software product lines (SPLs) suggest practical approaches for developing a family of similar systems satisfying common requirements along with some different characteristics for individual systems[12].

1.1.1 Feature Modeling

The goal of SPLs is coping with variable requirements, and feature modeling is one of the approaches used for this purpose[22]. Figure 1.1 is an example of a system's feature model. In feature modeling, different products of an SPL can be visually represented in a compact diagram, by means of features and relations among them. A feature in feature modeling is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system"[22]. By selecting some of the features, to be activated, and configuring software components with respect to them, one product can be obtained. However, to have a valid product, features cannot be selected arbitrarily. Selection of features must satisfy the constraints among the features, which are represented

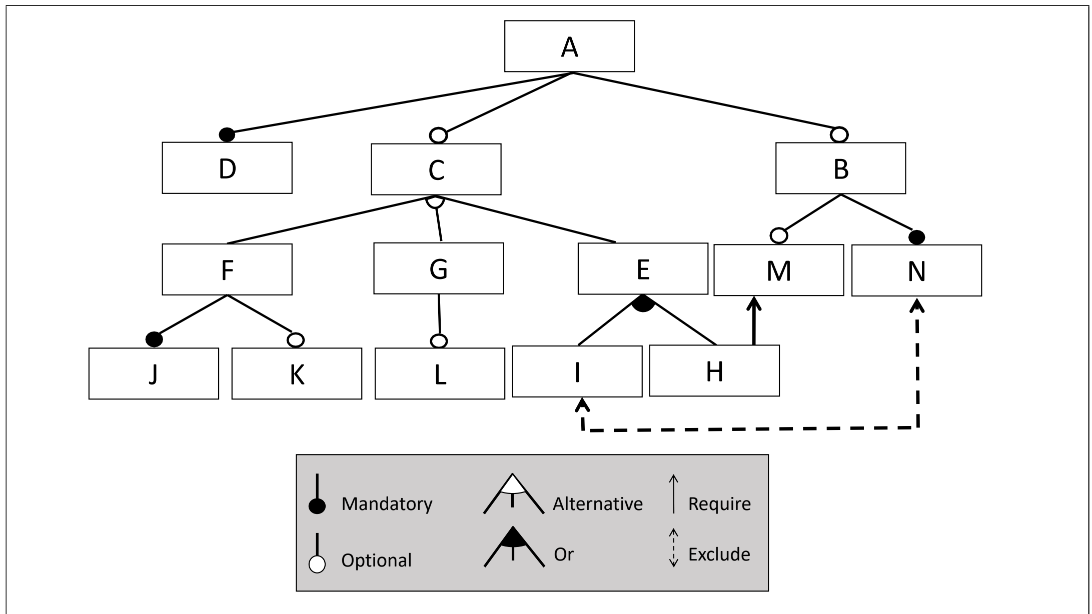

Figure 1.1: The feature model diagram of a system

by feature model relations in the diagram. Figure 1.2 includes all basic feature model relations and their description. The extended version of feature models includes attributes for the features. Using this characteristic, one product can be specialized by assigning specific values from the domain of the attributed features to those attributes.

1.1.2 Dynamic Software Product Line

In traditional SPLs, in which every product is supposed to satisfy a fixed set of requirements, selection of features and the corresponding configuration of software components occur before runtime. However, in the systems that deal with dynamic requirements, selecting features (and building the corresponding reconfigurations) must happen at run time too, which are called sometimes dynamic SPLs (DSPLs)[7] (such systems can also be referred as self-healing or autonomous systems). In traditional SPLs, every product includes a subset of features from its feature model, and as a result, every product includes only a subset of software components corresponding to the features in the feature model. On the other hand, since different features can be selected at runtime in a DSPL configuration, all of the software components related to the feature

Basic Feature Model Relation	Description
A / B	Feature B is a *Mandatory* subfeature of feature A. By selecting feature A in a single product, feature B must be selected in that product as well, and vice versa.
A / B	Feature B is an *Optional* subfeature of feature A. By selecting feature A in a product, feature B can be selected or not, but by selecting feature B in a product, feature A must be selected in that product as well.
A / B_1 … B_n	Features $\{B_1,\ldots,B_n\}$ are *Alternative* subfeatures of feature A. Either none of the features A and features $\{B_1,\ldots,B_n\}$, or feature A and one and only one member of $\{B_1,\ldots,B_n\}$ must be selected in a single product.
A / B_1 … B_n	Features $\{B_1,\ldots,B_n\}$ are Or subfeatures of feature A. Either none of the features A and features $\{B_1,\ldots,B_n\}$, or feature A and at least one member of $\{B_1,\ldots,B_n\}$ must be selected in a single product.
A → B	Features A Requires feature B. Each single product including feature A, needs feature B as well.
A ←--→ B	Features A and feature B Exclude each other. Both of feature A and feature B cannot be selected in a single product.

Figure 1.2: Basic feature model relations and their description

model must be accessible by each DSPL configuration. Figure 1.3 represents a valid DSPL configuration of the system which is mentioned in Figure 1.1.

In DSPLs, to respond to the changes in the operational environment based on the system's feature model, Cetina et al.[10] proposed Model-based Reconfiguration Process (MoRE), shown in Figure 1.4. In this process, the requirements of changes in the operational environment is translated into activation/deactivation of features. According to this process, a DSPL will respond to the changes in its operational environment by finding the corresponding features to be activated and deactivated and performing a valid reconfiguration considering all of the relations in the DSPL's feature model.

In another work, Benavides et al.[6] showed that feature models (and extended feature models) can be mapped to constraint satisfaction problems (CSPs). The definition of constraints corresponding to basic feature models are shown in Figure 1.5. Confronting a change in the operational environment, using the well known algorithmic approaches for CSPs[39], a feature set would be investigated to be activated and/or deactivated to satisfy the requirements of the context change along with the feature model relation constraints. However, trying to solve the CSP from scratch (determining values for all of the variables in the

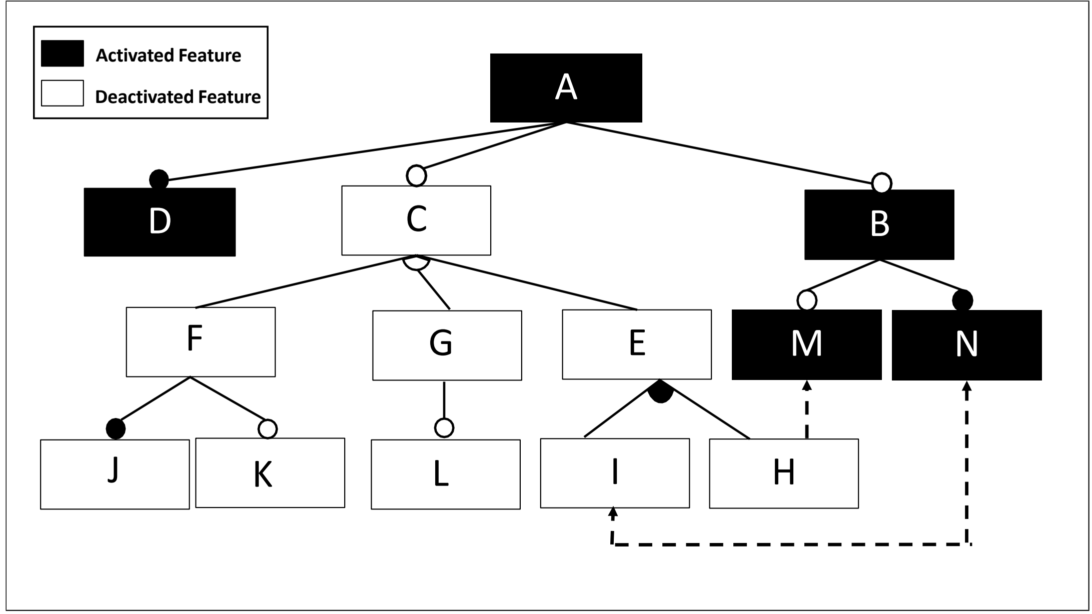

Figure 1.3: a DSPL configuration of the system shown in Figure 1.1

constraint network) by each reconfiguration request can be very costly. In such cases, the current state of the system can be used to improve the solution finding process for the new situation in an incremental way. Dynamic constraint satisfaction algorithms are proposed for this purpose[14].

1.2 Motivation

Receiving a request, a DSPL can be reconfigured in different ways with different costs. Every reconfiguration request needs changing the state or property of features. These changings affect the corresponding software components and impose specific cost on the system. In each system specific factor can be considered as cost to minimize in that system. It can be the cost that is needed to release and take the resources, and load and execute a software component. Software components can be used as services through Internet. The quality of service can be one matter to concern. Activating specific features can activate the components that control valuable hardwares and sensors. Maintaining the hardware might be of importance to some systems, and as a result hardware degradation can be considered as system cost to be concerned. Maintaining specific level of safety and security which can be changed by feature state or

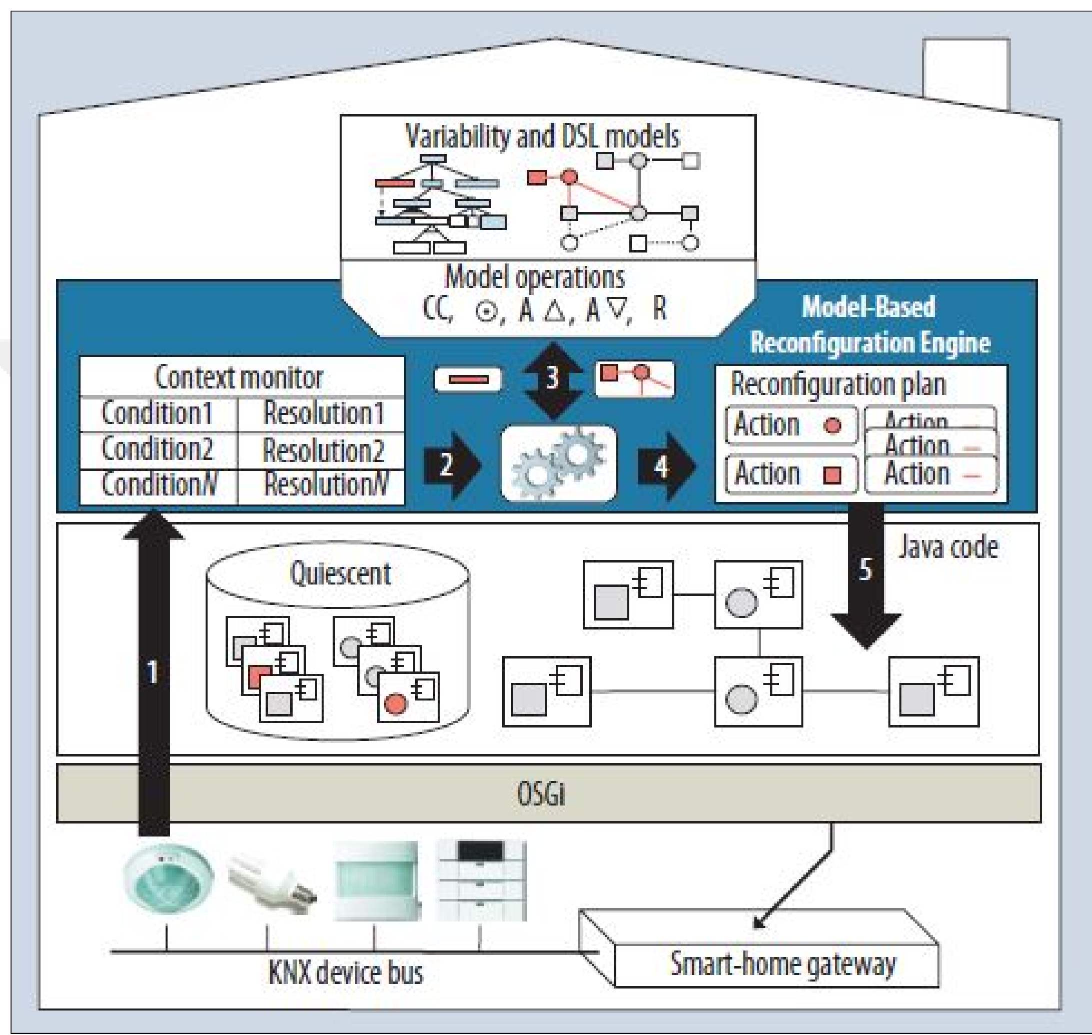

Figure 1.4: Model-based reconfiguration process[10]. MoRE translates contextual changes into changes in the activation/deactivation of features.

Relation Representation	Equivalent Constraint
A B	$A \Leftrightarrow B$
A B	$B \Rightarrow A$
A B_1 ... B_n	$A \Leftrightarrow ((B_1 \wedge B_2' \wedge \cdots \wedge B_n') \vee (B_1' \wedge B_2 \wedge \cdots \wedge B_n') \vee \cdots (B_1' \wedge B_2' \wedge \cdots \wedge B_n))$
A B_1 ... B_n	$A \Leftrightarrow (B_1 \vee B_2 \vee \cdots \vee B_n)$
A B	$A \Rightarrow B$
A B	$(A \wedge B)'$

Figure 1.5: Basic feature model relations and their description

property change can be of concern as well. Considering these factors, reconfiguration cost would be needed to be kept as low as possible in real applications of DSPLs. For example, suppose the feature model and current configuration of the collision avoidance system of a car is the one shown in Figure 1.6. The context of this system can be modeled with condition-resolution table, which proposed by MoRE process[10], such as the one shown in Table 1.1. Activation and deactivation of each feature in this system will need taking and releasing resources of the system which happens in specific time units. Table 1.3 shows the needed time for activation and deactivation of each feature in the system. Suppose the system with the configuration shown in Figure 1.6 confronts "Emergency Mode" in its context. Then, the system will need to reconfigure itself to a valid configuration as soon as possible to avoid the possible collision. According to condition-resolution table, to handle the "Emergency Mode" in the context, the "Autonomous Movement" feature in the system must be activated. However, there are different valid configurations for this system by activating this feature. Each valid configuration needs specific time to deal with context conditions, and it is preferable to choose the one that ends up in the shortest time. According to Table 1.3, the valid reconfiguration with the least cost by activating "Autonomous Movement" feature happens by activating features "Autonomous

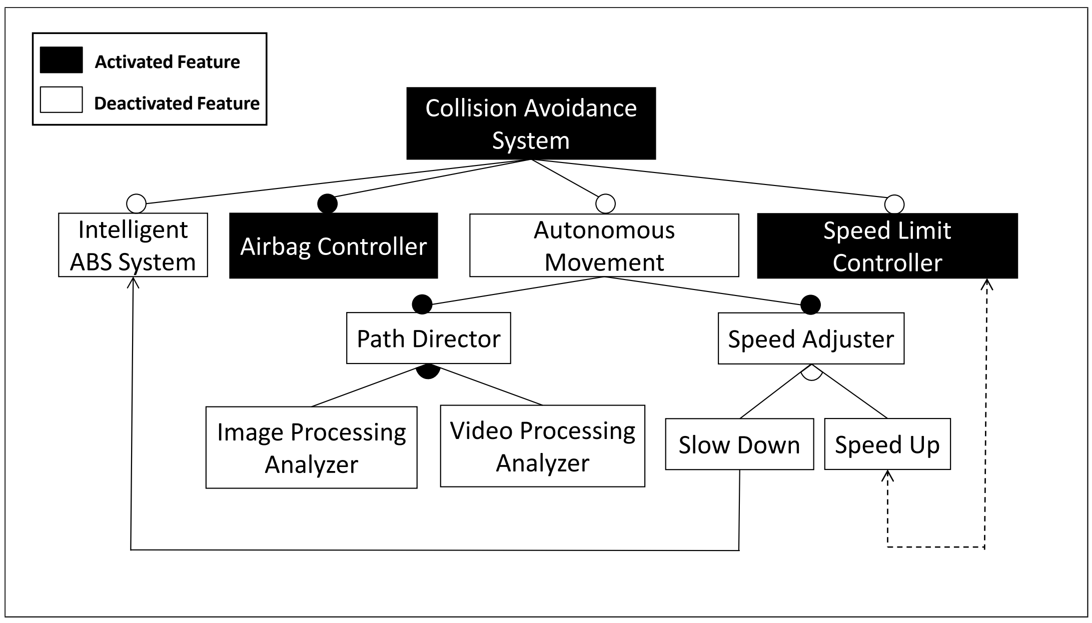

Figure 1.6: The feature model and current configuration of a collision avoidance system

Table 1.1: Condition-resolution table for the Collision Avoidance System shown in Figure 1.6

Condition	Resolution
Low Collision Possibility	Deactivate (Autonomous Movement, Intelligent ABS System)
Mediate Collision Possibility	Activate(Speed Limit Controller)
High Collision Possibility AND Bad Weather	Activate (Image Processing Analyzer, Video Processing Analyzer)
High Collision Possibility AND Technical Defects	Activate(Autonomous Movement), Deactivate(Speed up)
Emergency Mode	Activate (Autonomous Movement)

Movement", "Path Director", "Image Processing Analyzer", "Speed Adjuster", "Slow Down", and "Intelligent ABS System", and this reconfiguration demands 2 seconds cost. However, for example, an arbitrary valid reconfiguration by activating "Autonomous Movement", "Path Director", "Image Processing Analyzer", "Video Processing Analyzer", "Speed Adjuster",and "Speed Up", and deactivating "Speed Limit Controller" demands 3 seconds cost, which deals with the context condition 50% later than the time that it can do in ideal case.

Reaching the exact reconfiguration solution with a minimum cost might need great deal of time itself. Therefore, in real systems, searching for a solution can last until the time limitation allows, or the cost restriction is satisfied by the found solution up to that point. In this work, we try to approach a best reconfiguration solution of DSPLs which are modeled by feature modeling by an

Table 1.2: Feature state migration time cost for the feature model of Collision Avoidance System shown in Figure 1.6

Feature	**Feature State Migration Cost (ms)**	
	Activation Cost	**Deactivation Cost**
Collision Avoidance System	650	200
Intelligent ABS System	100	50
Airbag Controller	350	100
Autonomous Movement	700	300
Speed Limit Controller	200	100
Path Director	400	150
Image Processing Analyzer	500	100
Video Processing Analyzer	800	250
Speed Adjuster	200	150
Slow Down	100	100
Speed Up	300	100

algorithm and a set of heuristics, to do this task in an efficient way.

1.3 Problem Definition

The problem is obtaining a valid DSPL configuration that will be built using a feature model, by activating some of the features and deactivating the others. Every feature can only be in an activated or deactivated state. When there are changes in the operational environment, the system should keep or change the state of some of the features to satisfy all of the feature model constraints while switching to another valid configuration to suit the changes. However, for each request, different valid configurations can exist for performing the reconfiguration. Since each feature activation and deactivation change imposes a specific cost on the system, the reconfiguration cost will be the total cost of feature state changes that occur through the reconfiguration process. The goal is to efficiently find a valid reconfiguration that will impose the minimum reconfiguration cost while confronting the changes in the operational environment.

A constraint satisfaction problem (CSP) can be defined as a 3-tuple (V, D, C), where V is the sequence of variables, D is the sequence including the correspond-

ing domain of variables in V, and C is the set of constraints among variables in V. If by assigning values to the CSP variables from their domain, all of the constraints in a CSP become satisfied, then that CSP is called consistent, otherwise we call it inconsistent. Since every FM can be mapped to a CSP[6], we can describe our problem in the form of CSPs too. While mapping a basic FM to its corresponding CSP, every feature will be mapped to a variable where the domains of the variables are Boolean (activated feature as True or '1' and deactivated ones as False or '0'), and the feature model relations will be mapped to constraints among variables. The feature states will specify the values of the variables, and the validity of a feature model configuration can be specified by the consistency of its corresponding CSP. In the case of DSPL reconfiguration, we encounter a CSP where all of its variables are assigned with values from their domain, and all of the constraints are satisfied by the values of variables. We refer to this initialized CSP as ICSP with 4-tuples (V, D, d, C), where V, D and C are the same CSP parameters representing variables, the variable domains and constraints respectively, and d refers to the sequence of assigned values to the corresponding variables in V.

Using MoRE process [11] every change in the operational environment can be translated as a request R, which is defined by two tuples (V_R, d_R), where V_R represents a sequence of variables subsequence of V that are required to be assigned with specific values, and d_R represents the sequence of requested values corresponding to these variables. For the sequence of variables V, shown in equation 1.1, each request can be defined as shown in equation 1.2 where $m \leqslant n$, and the sequence of current values corresponding to V_R is represented with d_C as shown in equation 1.3.

$$V = (v_1, v_2, \cdots, v_n). \tag{1.1}$$

$$R = ((v_{R_1}, v_{R_2}, \cdots, v_{R_m}), (d_{R1}, d_{R2}, \cdots, d_{Rm})). \tag{1.2}$$

$$d_C = (d_{c_1}, d_{c_2}, \cdots, d_{c_m}). \tag{1.3}$$

Reconfiguration request R requires assigning the value of every v_{R_i}, which is currently d_{c_i}, to d_{Ri} for each $i \in \{1, 2, \cdots, m\}$, along with satisfying all of the constraints in C. Every request is assumed to require changing the value of at

Table 1.3: Cost of feature's state migration cost

Feature	State Migration Cost	
	Deactivation Cost	**Activation Cost**
f_1	CostD_1	CostA_1
f_2	CostD_2	CostA_2
...	...	...
f_n	CostD_n	CostA_n

least one variable. There can be no valid reconfiguration solution or multiple valid solutions for a request in a system. Every solution, if there is one, such as S for a request R and an ICSP, maintains the value of some of the variables and changes the others. Therefore, knowing the current values of all variables, each solution S can be defined with two tuples (V_S, d_S) specifying only the needed changes that must be imposed on the ICSP, as shown in equation 1.4 where $k \leqslant n$.

$$S = (V_S, d_S) = ((v_{S_1}, v_{S_2}, \cdots, v_{S_k}), (d_{S1}, d_{S2}, \cdots, d_{Sk})). \tag{1.4}$$

Here V_S represents the variables subsequence of V that must be changed, and d_S represents the sequence of new values corresponding to V_S. The idea is that by applying the changes specified by a solution, for a request and an ICSP, and maintaining the values of other ICSP variables, the request's requirements and ICSP constraints will be satisfied. Having the features's state migration cost, the cost of every solution S, represented by |S|, would be defined as the summation of all costs of feature state changes imposed by the reconfiguration solution S. Supposing the cost of changing the value of each v_{S_i} to d_{S_i} in solution S as $cost_{S_i}$, the cost of applying the solution S is shown in equation 1.5. Among different solutions, a solution is defined as optimum (or best) where there is no other solution with less cost than that solution. Thus, for a request R and an ICSP, a solution would be preferable as a best solution which have the minimum value for the formula shown in equation equation 1.5.

$$Reconfiguration\ Cost = \sum_{i=1}^{k} cost_{S_i}. \tag{1.5}$$

For extended feature models, the cost table, shown in Table 1.3, can change

to include the cost related to each property as well. In this case, the cost of selecting a property value and suspending another property can be considered separately in the total cost of the system's reconfiguration.

If we assume an equal cost for each feature state change, the cost of every solution S, would be defined as the number of variables in V_S. Thus, for a request R and an ICSP, a solution would be preferable as a best solution which have the minimum number of members in V_S.

1.4 Related Work

Sawyer et al.[33] proposed using goal-modeling techniques in DSPLs. Goal-modeling is used for modeling context and the relation between the context problems and their corresponding solutions, considering the QoS needed by each problem and provided by each solution. It is also explained in their work how a goal model can be mapped to a constraint network. Changing the context will be interpreted as changes in the corresponding CSP context variables, and solving the CSP can result in activation/deactivation of software components at the end. Sawyer et al. used VariaMos tool[26] to solve the CSP and reconfigure to a valid configuration by context changes. VariaMos is a tool for product line driven systems with a constraint based approach. It only solves a CSP and is not capable of optimizing any criterion. Using our approach, it can be extended to support dynamic reconfiguration efficiently while optimizing QoS, reconfiguration time, or other costs concerned with the system designers.

Rosenmuller et al.[29] provided a feature-base approach for self-configuration and runtime adaptation. Using a runtime adaptation framework that they provided, called FeatureAce, binding units are integrated at runtime to achieve a concrete program based on user defined feature selection. The needed configuration changes are calculated by FeatureAce using a SAT solver. As their future work, they mentioned using CSP solvers to optimize the reconfiguration proposed by FeatureAce based on different criteria. FeatureAce can use our algorithm and the specialized feature model heuristics for the purpose of optimizations and

enhancing the efficiency of feature model validation or solution finding process.

Sanchez et al.[30] proposed an approach for adaptation of systems based on specification, measurement and optimization of feature model quality attribute properties. For optimization problems, they used the algorithm of their previous work, called CSA (Configuration Selection Algorithm)[31]. For a feature model, it takes a set of features to be activated and deactivated as input and tries to find a feature model configuration which optimizes specific function based on a set of feature attributes.To find the optimal solution, it fixes the requested values for requested features and calculates some heuristic values for each feature to estimate whether its activating or deactivation would facilitate reaching an optimal solution at the end. The heuristics are based on calculating some values for each feature by relaxing some of the constraints in the feature model. In the evaluation section of that work [31], the features except the leaf features are assumed to have no effect on optimizing a specific criterion, which can be a oversimplification of the problem. The advantage of heuristics can be more realistically monitored if the problem is not oversimplified by that assumption. Moreover, this heuristic will put burden on the system if the system grows repetitively. By system evolution, the previously calculated heuristic values regarding a branch of features must be recalculated one more time. In addition, although the heuristics might be beneficial in estimating whether activation or deactivation of a feature can help to reach an optimal solution in some conditions, CSA has the same approach of constraint satisfaction optimization problem (CSOP) algorithms. In both approaches some nodes are not assigned with values, and the aim is to assign values to them while satisfying the existent constraints and optimizing a criterion from scratch. In this work, the efficiency of this approach is compared to the proposed algorithm. We enhance the efficiency of the common CSOP algorithm by using incremental approach proposed of DCSP algorithms.

In order to capture the context dependencies by SPLs, extending attributed FMs with Validity Formulas (VFs) is suggested by Mauro et al.[25]. In this approach, the constraints imposed by context are defined in the VFs of each feature, and valid configurations in each context can only include the activated features that their VF is valid at that context. They proposed HyVarRec, Hy-

brid Valid Reconfiguration Engine, for this purpose. By each change in the context, the VF of some of the activated features can be invalid, and the system will try to reconfigure to a valid configuration as a result. Finding a valid configuration happens by mapping the problem to a CSP and solving that in this work. In addition, because it is intended to find a valid configuration that is similar to the current configuration, they used CSOP algorithms to optimize the changes. If HyValRec utilizes the idea of DCSPs for doing reconfiguration and the suggested feature model heuristics, the intended valid configuration can be reached efficiently. We mention the efficiency difference of these approaches in the experiments chapter 3.

The algorithms suggested for CSOP are CSP algorithms which are used to minimize or maximize a parameter while finding a solution for the problem [39]. By applying a CSOP algorithm in our problem, for any given request, a solution with the least cost will be found, if there is any. However, because the domain of variables in our problem changes repetitively (suggested by each request), then by applying CSOP algorithm to solve the problem, the algorithm will solve the CSP from scratch by each change in the CSP. Because of the inefficiency which might be caused by CSOP algorithms, these kinds of problems can be mapped to dynamic constraint satisfaction problems (DCSPs) [14, 27, 15, 24].

There are different approaches to find solutions in DCSPs. Some approaches try to reuse the found solution of the current DCSP to reach a solution after the changes that occur in the system [5, 28, 4, 19, 32, 3, 37]. Some algorithms reuse the reasoning of reaching a DSPL solution to find the solution for the DCSP when it is changed [8, 9, 13, 35]. In some approaches, both solution-reuse and reasoning-reuse are utilized together[36, 37].

The aim of some DCSP algorithms is to find a robust solution which resists to changes [16, 34, 17, 40, 18]. Instead of finding robust solutions, some works propose finding the solutions that are easily adaptable to changes [21, 2, 20].

Verfaillie et al.[37] proposed an algorithm to approach the solution in DCSPs by reusing the previous solution. In our work, using the basic idea of this algorithm and the CSOP algorithms, we propose an algorithm to approach a solution in the

constraint network which imposes the minimum number of changes. To enhance the efficiency, feature model related heuristics are also proposed regarding this algorithm.

The reminder of this work is organized as follows. In chapter 2 we propose our algorithmic approach to reach an optimal reconfiguration of DSPLs with a feature model as the system's variability model. In our algorithm, we use the idea of imposing local changes on the previous CSP solution to find a new solution in the CSP, proposed by Verfaillie et al.[38]. We use a DCSP algorithmic approach when the variables and constraints are fixed but the domain of variables changes. In addition, we propose some heuristics to enhance the efficiency of the algorithm while applying that on a feature model. In chapter 3, we discuss the results of executing CSOP, and our algorithm with and without feature model heuristics on a sample feature model and present the results. Finally, in chapter 4 we talk about the conclusions that we made from all of the results we experiences through this work.

CHAPTER 2

ALGORITHM

Using the main ideas of the algorithm proposed by Verfaillie et al.[38] and CSOP algorithms[39], we propose an algorithm to find an optimal feature model configuration for each DSPL reconfiguration request. Our algorithm includes two sections as shown in Procedure 1 and Procedure 2. Procedure 1 starts the requested change(s) in the CSP and checks whether or not the ICSP is still consistent. If inconsistent, Procedure 2 will be called to distribute the imposed change(s) in the constraint network to find a case where the ICSP is consistent. In other words, after imposing the change(s) by Procedure 1, some of the constraints can be violated, but Procedure 2 tries to satisfy the violated constraints one by one to find the cases for which no violated constraint exists in the system at the end. If there are such consistent cases, it returns the one that imposes the minimum cost. Otherwise, it realizes that no valid reconfiguration can be performed for that request in the system. Because the problem is an NP complete problem, in worst case, the traversing steps of the algorithm would be the multiplication of the number of domain values for all of the variables in the constraint network.

FindConfiguration procedure shown in Procedure 1 takes an ICSP and a reconfiguration request as input, and if there is a way to satisfy the request along with all of the existent ICSP constraints, it will propose an optimum solution. In this work, we call a constraint and a variable *directly related* if in the definition of that constraint the possible values of that variable takes a role and has the potential to directly change that constraint's satisfactory state. For example, in the feature model relation "A excludes B", the corresponding constraint

definition would be "$\neg(A \wedge B)$". We say this constraint and the variables A and B directly related to each other.

In Procedure 1 and Procedure 2, the used notations are defined as follows.

- ICSP[d (V=X)] : the sequence of variable values corresponding the sequence of variables X in the ICSP
- ICSP[C (V=X)] : the set of unsatisfied constraints in the ICSP, where each constraint is directly related to at least one variable in the sequence of variables X
- ICSP[V (C=Z)] : the sequence of variables in the ICSP, subsequence of V, such that each variable is directly related with at least one constraint in the set of constraints Z
- A $\cup_V$ B: for two sequences A and B which are subsequences of V, A $\cup_V$ B is a subsequence of V and includes all members of A and B (For example if V=(1,2,3,4,5), A=(1,2,4) and B=(2,5), then A $\cup_V$ B is (1,2,4,5))

FindConfiguration procedure shown in Procedure 1 takes a consistent ICSP and a reconfiguration request as input, and if there is a way to satisfy the request along with all of the existent ICSP constraints, it will propose a solution with the minimum cost. This procedure first applies the request on the ICSP. Then, it determines the changed variables and the constraints directly related to them. These constraints are the only potential constraints with the possibility of being unsatisfied and must be checked and satisfied if needed. By satisfying the unsatisfied constraints in each step, some other constraints might become unsatisfied in turn and so on. The variable value propagation is intended to be continued in the ICSP with the aim of reaching a point where no constraint is unsatisfied anymore. The distributing of values in the ICSP, which is the main solution finding action, is continued by Solve procedure shown in Procedure 2.

Distribution of values is performed in the Solve procedure recursively. Each call of the Solve procedure tries to satisfy one unsatisfied constraint in the ICSP. Approaching a solution is performed by making repetitive calls of the Solve

Procedure 1 FindConfiguration

Input: Consistent ICSP (V, D, d, C), and request R (V_R, d_R)

Output: Either an optimum solution S (V_S, d_S), or $fail$

Global Shared Variables: S (V_S, d_S), $Cost_S$

```
1: procedure FindConfiguration(ICSP, R)
2:     V_S, d_S, V_changed, d_first ← empty
3:     Cost_S ← 0
4:     for i := 1 to |V_R| do
5:         if d_R[i] ≠ ICSP[d(V = V_R[i]] then
6:             add V_R[i] to the end of V_changed
7:             add ICSP[d(V = V_R[i]] to the end of d_first
8:             ICSP[d(V = V_R[i]] ← d_R[i]
9:         end if
10:    end for
11:    C_u ← ICSP[C(V = V_changed)]
12:    if C_u is empty then
13:        (V_S, d_S) ← (V_changed, ICSP[d(V = V_changed])
14:        ICSP[d(V = V_changed)] ← d_first
15:        return (V_S, d_S)
16:    end if
17:    SS ← (C_u, V_R, V_changed, total cost of V_changed)
18:    solve(ICSP, SS)
19:    ICSP[d(V = V_changed)] ← d_first
20:    if |V_S|=0 then
21:        return fail
22:    else
23:        return (V_S, d_S)
24:    end if
25: end procedure
```

procedure for each unsatisfied constraint. In each call to satisfy one constraint, specific values will be determined for that constraint's directly related variables. The next calls will be used to satisfy other unsatisfied constrains in the system, if there is any left.

Our algorithm starts by distributing the values in the system from a specific state called solving start (SS), which is defined by a 4-tuple $(C_u, V_f, V_x, Cost_x)$. Here, C_u represents the set of possibly unsatisfied constraints in the ICSP. V_f represents the set of variables whose values are already fixed through the value distribution in the constraint network. In our algorithm, a call of the Solve procedure to satisfy the unsatisfied constraints is not allowed to change the values of the variables determined by previous Solve calls. V_x represents a subsequence of V_f such that their values are changed through value distribution in the constraint network, and $Cost_x$ represents the cost of those changes in the system.

To approach an optimum solution, two separate stages are covered by our algorithms: finding a first solution and finding an optimum solution.

Procedure 2 Solve

Input: Inconsistent ICSP (V, D, d, C), and solving start state SS $(C_u, V_f, V_x, Cost_x)$

Ensure: Updating solution S(V_S, d_S) by finding a new solution for the ICSP starting from the SS, either when no solution is found before or when the new solution has less cost than the found one

Global Shared Variables: S (V_S, d_S), $Cost_S$

1: **procedure** SOLVE(ICSP, SS)
2: $(c_1, CS_{rest}) \leftarrow$ (an arbitrary member of C_u such as c_x, member(s) of C_u other than c_x)
3: **if** c_1 is satisfied **then**
4: **if** CS_{rest} is empty **then**
5: $(V_S, d_S) \leftarrow (V_x$, ICSP$[d(V = V_x])$
6: **else**
7: $SS_2 \leftarrow (CS_{rest}, V_f, V_x, Cost_x)$
8: SOLVE(ICSP, SS_2)
9: **end if**
10: **else**
11: $V_{rel} \leftarrow$ (ICSP$[V(C = c_1)]$ - V_f)

12: $d_{rel} \leftarrow$ (ICSP$[d(V = V_{rel})]$)
13: $Sat_{All} \leftarrow$ values corresponding V_{rel} satisfying c_1
14: **for each** sequence such as Sat_S from Sat_{All} **do**
15: assign $SatS$ to ICSP$[d(V = V_{rel})]$ and save the changed variables of ICSP as $V_{changed}$ and their totally imposed cost as $Cost_{new}$
16: **if** ($|V_S|$=0) OR ($|Cost_x|$+ $|Cost_{new}|$<$|Cost_S|$) **then**
17: $V_{fAll} \leftarrow V_f \cup_V V_{rel}$
18: $V_{xAll} \leftarrow V_{changed} \cup_V V_x$
19: $CS_{new} \leftarrow$ ICSP$[C(V = V_{changed})]$
20: $CS_{All} \leftarrow CS_{rest} \cup CS_{new}$
21: $Cost_{All} \leftarrow Cost_x + Cost_{new}$
22: **if** CS_{All} is empty **then**
23: $(V_S, d_S) \leftarrow (V_{xAll}$, ICSP$[d(V = V_{xAll}])$
24: $Cost_S \leftarrow Cost_{All}$
25: **else**
26: $SS_2 \leftarrow (CS_{All}, V_{fAll}, V_{xAll}, Cost_{All})$
27: SOLVE(ICSP, SS_2)
28: **end if**
29: **end if**
30: ICSP$[d(V = V_{rel})] \leftarrow d_{rel}$
31: **end for**
32: **end if**
33: **end procedure**

2.0.1 Finding a first solution

Initially, it is the FindConfiguration procedure that calls the Solve procedure, when V_S and d_S are empty, meaning no solution is found yet. This procedure takes an inconsistent ICSP and proceeds with finding a solution for that ICSP from a starting state $(C_u, V_f, V_x, Cost_x)$. We call the first starting sate in the solution finding process as the initial starting state, which is set up by the Find-Configuration procedure. To make the ICSP consistent, all of the constraints in C_u must be checked, and satisfied if needed. Solve procedure checks and satisfies

the unsatisfied constraints recursively. Assuming c_1 is an arbitrary constraint in C_u and CS_{rest} as constraints of C_u other than c_1, Solve procedure starts by satisfying C_u from c_1.

First, satisfaction of c_1 is checked (Procedure 2 line 3). If it is satisfied, then the procedure will proceed to satisfy the other constraints in CS_{rest} (Procedure 2 line 8), if there is any. If CS_{rest} is also empty, it means that all of the possibly unsatisfied constraints in the ICSP are checked and all are satisfied. Therefore, the algorithm realizes ICSP is consistent, and then, it saves the found solution. On the other hand, if c_1 is found to be unsatisfied by the Solve procedure, it will try to satisfy it. The value of the variables that are included in V_f are kept fixed and not allowed to change by distribution of values. Therefore, c_1 can be satisfied by changing only the values of its directly related variables that are not included in V_f. If by changing the values of these variables to any possible value from their domain, c_1 cannot be satisfied, it means that the ICSP cannot reach a consistent state from the given starting state. If the algorithm encounters this case while the solution S is not initialized yet, it means that no solution is found so far. On the other hand, if by assigning the values to the directly related variables of c_1 that are not fixed, called modifiable variables in this work, c_1 becomes satisfied, the algorithm continues its work by checking the other unsatisfied constraints of the ICSP in the next step, if there is any.

After satisfying c_1, if there are no other unsatisfied ICSP constraints, one solution will be reached. Otherwise, the algorithm continues and tries to satisfy all of the unsatisfied constraints of the ICSP one by one by assigning different values to the variables, and distributing new values in the constraint network.

Satisfying c_1 occurs by changing the value of some of the variables in the ICSP that are directly related to c_1. These new changes can cause some other unsatisfied constraints become satisfied as well. Similarly, it can cause some new constraints in the ICSP become unsatisfied, the constraints other than the ones included in CS_{rest} (the constraints shown in Procedure 2 line 19). Therefore, to reach a solution in such cases, these newly unsatisfied constraints must be considered to be satisfied at the next Solve calls as well (Procedure 2 line 27).

In addition, because all of the constraints are satisfied at the beginning of the solution finding process, variables with the changed values are the only points that can make some constraints of the constraint network unsatisfied. Thus, by considering all of the possibly unsatisfied constraints for each change in the ICSP and trying to satisfy them by assigning possible values, the algorithm will discover if there is a case that all of the constraints are satisfied.

In some cases, distribution of values leads to a point that all of the constraints in the ICSP are satisfied. This is the point of reaching a solution (Procedure 2 line 23). However, satisfying a constraint by assigning one of the possible values to its directly related variables and distributing its effects in the ICSP can yield a point that the ICSP cannot be satisfied anymore. In such case, the algorithm realizes that the distribution of variables has not approached to a solution. To confront such cases, the algorithm backtracks and tries to satisfy the unsatisfied constraints by assigning other values to their directly related variables, if possible. Similarly, this backtracking approach will be continued either to reach a solution (updating V_S and d_S) or to conclude that there is no solution at the end. In the case of no solution, the algorithm terminates where the solution S, (V_S, d_S), is empty.

2.0.2 Finding an optimum solution

If there is a way to satisfy the requirements of a request in the system, the Solve procedure will finally find a first solution and will update (V_S, d_S) and the cost of that solution $Cost_S$. After finding that solution, the Solve procedure will continue in order to find a solution that imposes the minimum cost (an optimum solution). Thus, after finding the first solution, it will try to satisfy the unsatisfied constraints by assigning other values to their directly related variables. In other words, for each set of possible variable values, the Solve procedure will distribute those values, aiming to reach a solution with a less cost than the previously found one. Then, for each newly found solution, the solution cost will be compared to the previously saved one. If the cost of the new solution is higher than the previous one, the new solution will be discarded,

and the Solve procedure will continue to find other solutions, if there is any. On the other hand, if the new solution has a less cost than the previous one, the new solution will be replaced with the previous one, as the best solution up to that point. The same process will happen for the next solutions, and at the end the saved solution will be the solution with the least cost.

Determining that a solution will have more cost than the previously found one can be recognized in the early steps as well. Since propagation of the variable values happens gradually, the changes occurs in the ICSP can be computed gradually as well. Therefore, the lower bound cost of the upcoming possible solution can be determined by each constraint satisfaction (Procedure 2 line 16). Thus, if in the middle of finding a new solution, the total cost exceeds the cost of the previously saved solution, it can be determined that approaching to a possible solution will lead to a cost more than the found one. In such cases, there is no need to traverse the whole branch anymore. Solve procedure can prune this branch and continue with the next cases. As a result, finding an optimum solution can be reached in a more efficient way.

2.1 Heuristics

Our algorithm is proposed to be applied to the CSP corresponding to the feature model of a DSPL and finds the reconfiguration solutions confronting the changes in operational environment. Utilizing the characteristics of the feature model constraints, the efficiency of the algorithm would be improved. Now, we discuss feature model heuristics to enhance the performance of the algorithm for feature model reconfiguration.

In this section we use some concepts for feature model diagrams similar to the ones in graph theory [41]. By eliminating the Require and Exclude relations from feature model diagram, we will have a tree structure, the feature representations as nodes and the relation representation as edges among the nodes. In this tree structure, we call the feature that has no parent feature the *root feature* and the features that have no child as *leaves*. Similarly, we define height of a feature as

the length of the longest downward path to a leaf from that feature (root with the longest height and leaves with the height zero).

2.1.1 Restricting distribution of variable values

Each basic feature model relation can be mapped to a logical expression in the constraint network[23]. However, following the exact definition of feature model constraints can lead to traverse the branches that can be unnecessary. For example, suppose a DSPL configuration includes five features A, B, C, D, and E, where A and B are activated, and C, D and E are deactivated and there is an OR relation such as 'A has OR children B, C, D, and E'. Following [23], this relation would be mapped to a constraint with the definition $(B \vee C \vee D \vee E) \Leftrightarrow A$ in the constraint network having the values (A, B, C, D, E) = (1, 1, 0, 0, 0). In this case, when a request needs B to be deactivated, the best solution will be searched by assigning B to 0 and determining acceptable values for the other variables. Using the truth table for the logical expression $(B \vee C \vee D \vee E) \Leftrightarrow A$, there are eight acceptable sets of values for the OR relation variables when B is 0, as follows:

- (A, C, D, E) = (0, 0, 0, 0)
- (A, C, D, E) = (1, 0, 0, 1)
- (A, C, D, E) = (1, 0, 1, 0)
- (A, C, D, E) = (1, 1, 0, 0)
- (A, C, D, E) = (1, 0, 1, 1)
- (A, C, D, E) = (1, 1, 0, 1)
- (A, C, D, E) = (1, 1, 1, 0)
- (A, C, D, E) = (1, 1, 1, 1)

To investigate a best solution by the algorithm, these sets of values should be assigned to the variables one by one, and their effect on the other variables in the system must be computed. Then, at the end, the set of assignments that imposes the minimum changes in the system would be a best solution. However, using our algorithm, when B is needed to be deactivated, the distribution points can be selected differently to approach a best solution. The efficient value distributions when B is intended to be 0 are the cases below:

- (A, C, D, E) = (0, 0, 0, 0)
- (A, C) = (1, 1)
- (A, D) = (1, 1)
- (A, E) = (1, 1)

Our algorithm goes through a constraint when it is unsatisfied. Here OR constraint becomes unsatisfied by deactivating one of its children. It means that only A is activated and all of its children are deactivated. In this condition, activating one or more child(s) of OR relation will satisfy the constraint. However, since the cost of changes imposed by one variable is preferable than more than one change here, activating more than one child feature can be disregarded in this step. For the cases that A is intended to remain activated, the variable value distribution can be continued by activating only one child feature. The other children should only be allowed to be activated at the same time if the distribution of values from other constraints in the constraint network requires that. By this approach in the mentioned example, covering only four branches for the OR relation features (instead of eight) can be enough to find the best solution in the constraint network.

In other words, confronting an unsatisfied OR constraint $(V_1 \vee V_2 \vee \cdots \vee V_n) \Leftrightarrow A$ in our algorithm, we can use the following approach to search the best solution efficiently.

- If *0* is needed to be assigned to A, the only branch to traverse includes assigning all of the V_i variables ($1 \leq$ i $\leq$ n) to *0*.
- If *1* is needed to be assigned to A, all of the branches including assigning *1* to only one of V_i variables ($1 \leq$ i $\leq$ n) are needed to be traversed.
- If *0* is needed to be assigned to V_i, one branch to traverse includes assigning all of the variables to *0*. The branches including assigning *1* to A and *1* to only one of the V_i variables ($V_i \neq V_j$) must be traversed as well.
- If *1* is needed to be assigned to V_i, the only branch to traverse includes assigning *1* to A.

SAT solvers use the boolean constraint networks as a set of propositions for example in CNF form, and mostly solve OR constraints by assigning value to

only one variable of OR constraint at each time, and then go ahead. However, our approach is more efficient than this technique because in our approach all of the values for all of the variables of a constraint are not going to be checked. Only the set of values for the set of variables of a constraint that makes that constraint satisfied will be checked in the network. Moreover, assigning all of the values for the variables of one constraint is more efficient than doing that one by one. The reason is that in our approach if it needs to backtrack, it would be realized earlier because of wide distribution of values at one time.

The main idea of this pruning is using the current values of directly related variables of an unsatisfied constraint as much as possible, which is only useful for OR feature model constraint. For a new kind of constraint, which is intended to be added to the feature model (e.g., complex relations), the existence of unnecessary branches to traverse by using our algorithm and following the truth table of that constraint can be considered individually as well.

2.1.2 Satisfaction Priority of Constraints

Among a set of unsatisfied feature model constraints, having higher satisfaction priority for the constraints Mandatory, Optional, Require and Exclude rather than Alternative and OR, and satisfying them earlier in order can enhance the efficiency of the algorithm. In each of the feature model constraints Mandatory, Optional, Require and Exclude, whenever the constraint is violated by changing the value of one of its directly related variables, there is only one way that makes that constraint satisfied and it is changing the value of the other variable. However, a violated Alternative or OR constraint can be satisfied in multiple ways. By limiting the domain of variables earlier, determining to avoid traversing some of the branches can be clarified earlier as well. For example, in a constraint network including two constraints 'X requires Y' and 'Z has alternative children T and Y' where all of the variables are assigned to false, while confronting a request to activate X and Z, satisfying the Require constraint first will avoid to check the branches including T=1, which will not yield a solution. However, if the alternative constraint is satisfied first, the branches including T=1 would be

traversed. Having the mentioned constraint satisfaction priority will determine the branches that will not lead to a solution earlier. Therefore, the algorithm will converge to the possible solutions more efficiently.

2.1.3 OR and Alternative constraints satisfaction

Each of the unsatisfied OR and Alternative constraints can be satisfied with different assignments to their directly related variables. By doing each assignment and distributing the effects in the constraint network, different sets of feature changes can be needed at the following steps. The question here is how to satisfy an Alternative or OR constraint to have a fewer number of changes in the following steps by distributing the values accordingly. Confronting an unsatisfied OR or Alternative constraint such as C, assuming feature A as the parent in the constraint C in the feature model and $\{V_1, V_2, \cdots, V_n\}$ are the children of A, we propose the following heuristics to satisfy C.

2.1.3.1 Value distribution toward the leaf features with fewer unsatisfied Require and Exclude constraints

When some assignments to satisfy the constraint C include changing the value of A and some include keeping the value of A, the assignments that keep the value of A would be preferable to be checked first in this heuristic. This heuristic uses the idea that the effect of changing the longer height features can lead to additional distributions in the constraint network (e.g., in a feature model of a system, deactivation of the root feature, which has the longest height, will cause *all* of the features in the system to become deactivated). In other words, this heuristic uses the fact that in feature modeling the features that have longer height are the more abstract features and changing their state can influence the configuration of the system more drastically.

While distributing the values downward, first traversing the branches that reach the leaves earlier can enhance the efficiency too. The aim of orienting value distribution direction toward the leaves is to stop value distribution at these

points. By this orientation, the value distribution will be ended in leaves if the leaves are not directly related to any Require and Exclude constraints. Require and Exclude constraints have the potential to propagate the distribution of values from each feature in the system to another one, maybe to the features with a higher height. Therefore, approaching the leaves with less unsatisfied Require and Exclude constraints, can improve the efficiency as well.

While satisfying the constraint C, to orient the value distribution toward the leaves, the average height of the child(s) of non-leaf variables $\{V_1, V_2, \cdots, V_n\}$ can be used as a criterion about how deep the values can go downward. Therefore, among the variables $\{V_1, V_2, \cdots, V_n\}$, the ones that are leaves have the highest priority to be changed, and after them the ones with lower average child(s) height has the next priorities in order.

Confronting an unsatisfied Alternative or OR constraint such as C, considering to face fewer unsatisfied OR and Alternative constraints can be useful while satisfying C as well. To do so, the assignments to the directly related variables of C that leads to fewer unsatisfied Alternative and OR constraints directly related to directly related variables of C should be checked earlier. By changing the values of directly related variables of C, some OR and/or Alternative constraints directly related to the directly related variables of C can become unsatisfied. Knowing the fact that traversing the branches of these two constraints can be more costly than the others, the efficiency of the algorithm can be enhanced by trying to avoid such cases as much as possible.

2.2 Soundness and completeness of the algorithm

In this section we talk about the soundness and completeness of the algorithm. First, we discuss why the algorithm is sound. Then, the completeness of the algorithm will be discussed.

2.2.1 Soundness

The algorithm starts its work when all of the constraints in the constraint network are satisfied. Then, it changes the values of variables that are requested in the reconfiguration request. By these changes, only the directly related constraints of the changed variables can be potentially unsatisfied. The algorithm checks the directly related variables of the changed variables, and if there are unsatisfied ones, it tries to satisfy them by assigning different values to the other directly related variables of those unsatisfied constraints. If there is a possible case that these constraint be satisfied again, the algorithm finds that.

New changes can make some other constraints in the constraint network be unsatisfied again. New values must be distributed in the network to see if other constraints will be unsatisfied again. This process continues, and at the end, if there is a set of assignments for the directly related variables of the newly unsatisfied constraints that all of them be unsatisfied, the algorithm finds that. Since before starting the algorithm all of the constraints was satisfied, and the algorithm checks and finds a set of assignments to satisfy the unsatisfied constraint by applying the request in the network, the constraint network will be consistent after applying those assignments too. In other words, if the algorithm finds a set of assignments to satisfy the request and all of the constraints in the network, the solution is correct. Furthermore, checking the cost of changes through the solution finding process, guarantee that if the algorithm finds a solution at the end, it has the minimum cost.

2.2.2 Completeness

The algorithm takes a constraint network as input where all of its constraints are satisfied by the values of variables. Then, it changes the values of the variables in the request and checks directly related constraints that got unsatisfied by those changes. It tries to satisfy them by changing the values of variables directly related to those constraints. These changes can make some other constraint unsatisfied in turn and so on. The algorithm step by step tries to satisfy the

unsatisfied constraints and if it reaches a point that there is no way to satisfy a constraint, it backtracks and tries other value assignments to the directly related variables of the unsatisfied constraints. This process continues up to the point that the algorithm reaches a solution or it realizes that the request cannot be satisfied along with having a consistent constraint network.

If, for the request and the constraint network, there are multiple solutions, the algorithm reaches the one which imposes the least cost in the system. It happens by checking the costs of the changes through approaching a possible solution in the constraint network. The solution with a less cost replaces the found solution with the higher cost. As a result, the final found solution, if there is any, would be a solution with the minimum cost.

2.2.3 completeness with heuristics

The heuristics discussed in section 2.1 are proposed to increase the chance of reaching reconfiguration solutions more efficiently. In sections 2.1.2 the priority of satisfying the unsatisfied constraints is discussed, and in section 2.1.3 a heuristic regarding the tree structure of the feature models is proposed. However, the heuristic in section 2.1.1 suggests some prunings to avoid traversing some of unnecessary branches. Having pruning in the algorithm, a question arises about the completeness of the algorithm while using that pruning. In the following, the completeness of the algorithm while using the pruning of the section 2.1.1 will be discussed over an OR constraint $(V_1 \vee V_2 \vee \cdots \vee V_n) \Leftrightarrow A$ namely constraint C.

According to the heuristic 2.1.1, if *1* is needed to be assigned to A, traversing the branches that include assigning *1* to more than one of V_i variables (1 ≤ i ≤ n) will be pruned while satisfying C. However, this pruning does not prohibit reaching a solution that includes assigning *1* to more than one V_i variables (1 ≤ i ≤ n). If a solution needs assigning *1* to more than one V_i variables, it means other constraints in the constraints network need that assignments (C can be satisfied by only one of them). Therefore, if another constraint in the constraint network needs assigning *1* to one V_i variables (1 ≤ i ≤ n), by

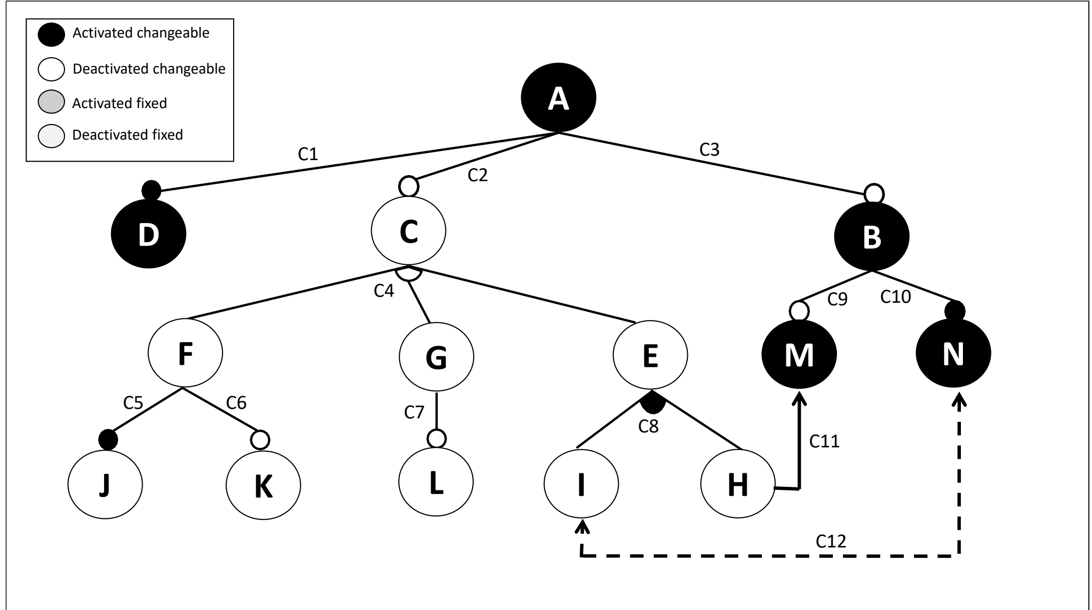

Figure 2.1: Mapping the feature model shown in Figure 1.3 to a constraint network

reaching that constraint and satisfying that, this aim will be achieved. In other words, using this pruning, the algorithm is still capable of finding the possibly optimal solutions that include assigning *1* to more than one V_i variables. With the similar reasoning, the algorithm can find the optimum solution when it is needed to change the value of one V_i variables ($1 \leq$ i $\leq$ n). Therefore, using the heuristic 2.1.1 along with the algorithm does not violate the completeness of the algorithm.

2.3 Tracing Example

To show how our algorithm works we trace the algorithm on the feature model representing the current configuration of a system shown in Figure 1.3. This feature model will be mapped to a constraint network shown in Figure 2.1, where the definitions of the constraints are mentioned in Table 2.1. For simplicity, in this example we suppose the same cost for changing each feature in the system.

Supposing that the system receives a reconfiguration request to activate feature C and deactivate feature M, the FindConfiguration procedure will be called to find the solution with the least cost by applying the request. This procedure

Table 2.1: Definition of constraints in the constraint system of the Figure 2.1

Constraint Name	**Definition**
C1	A ⇔ D
C2	C ⇒ A
C3	B ⇒ A
C4	(F'∧ G'∧ E'∧ C') ∨ (F∧ G'∧ E'∧ C) ∨ (F'∧ G∧ E'∧ C) ∨ (F'∧ G'∧ E∧ C)
C5	F ⇔ J
C6	K ⇒ F
C7	L ⇒ G
C8	E ⇔ (I ∨ H)
C9	M ⇒ B
C10	N ⇔ B
C11	H ⇒ M
C12	(I∧N)'

starts distribution of values in the ICSP (Figure 2.2).

By activating C and deactivating M, the constraint C4 would be violated, and Solve procedure must be called to continue distribution of values in order to satisfy C4. The Solve procedure will be called for the ICSP of Figure 2.2, where the value of variables C and M are fixed, the only unsatisfied constraint is C4 and the only changed variables are C and M. This call finds three ways to satisfy C4: activating one children of C (variables E, F and G) and deactivating the other ones. Supposing that, first, E is picked to be activated and F and G to be deactivated, the ICSP would be transformed to the one shown in Figure 2.3.

After activating E and deactivating F and G, the constraint C8 will become unsatisfied and Solve procedure would be needed to be called one more time. Then, Solve procedure will be called when the fixed variables are C, M, E, F and G, and the only unsatisfied constraint is C8, and the changed variables are C and E. To continue value distribution, there are two ways to satisfy C8: activating I or H. Supposing that H is picked, the ICSP will be the one shown in Figure 2.4.

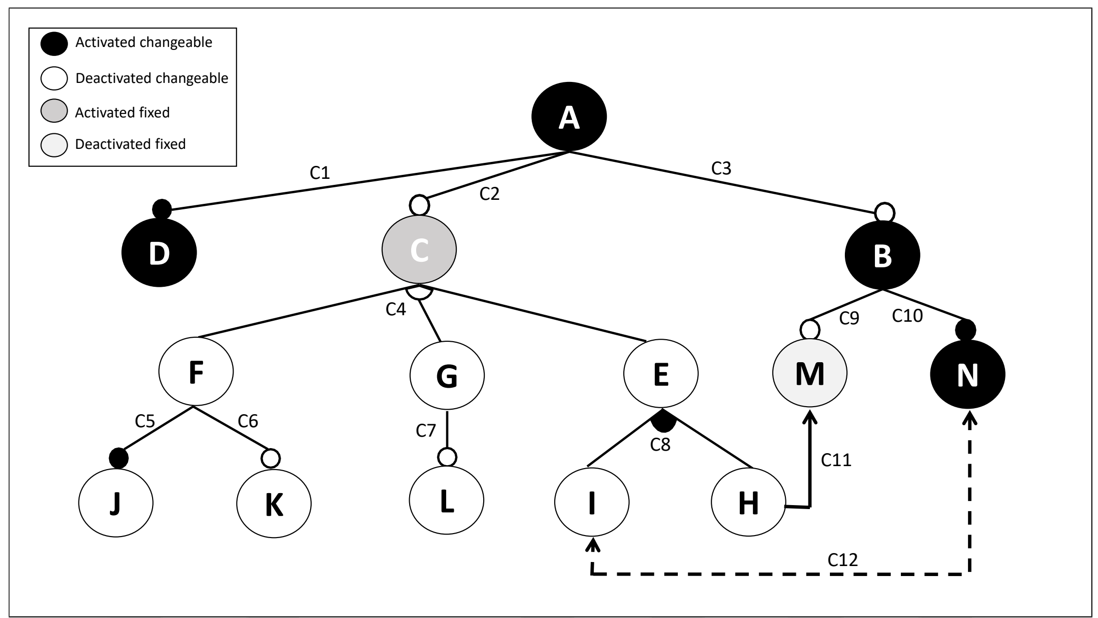

Figure 2.2: Starting variable value distribution in the ICSP shown in Figure 2.1 by calling FindConfiguration procedure to activate C and deactivate M.

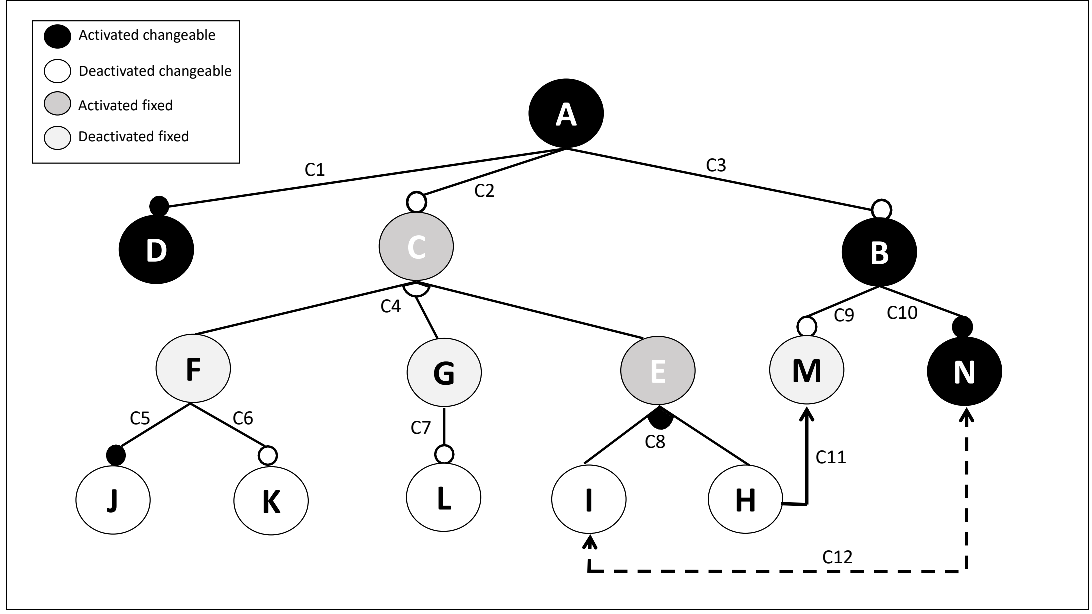

Figure 2.3: Satisfying C4 by calling Solve procedure for the ICSP shown in Figure 2.2, where C_u=C4, V_f = (C, E, F, G, M), V_x= (C, E, M). C4 is decided to be satisfied by Activating E and deactivating F and G.

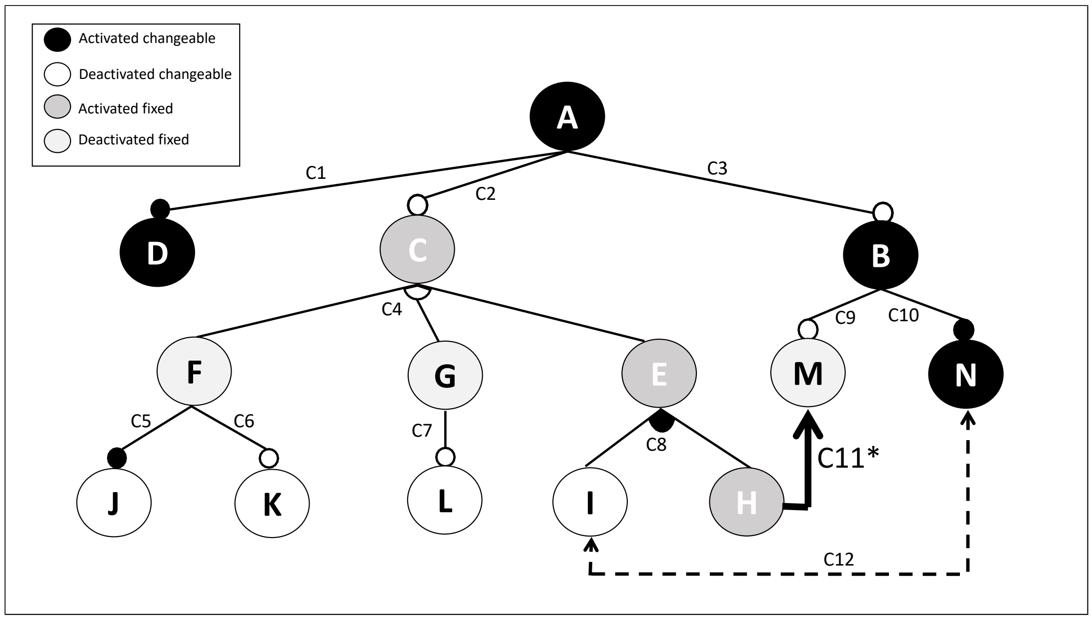

Figure 2.4: Satisfying C8 by calling Solve procedure for the ICSP shown in Figure 2.3, where C_u=C8, V_f = (C, E, F, G, H, M), and V_x= (C, E, H, M). Satisfying C8 by activating H will result in an inconsistent constraint network.

By activating H, one another constraint will become unsatisfied again, constraint C11, which another Solve procedure call is needed for that too. Then, this procedure is called when the fixed constraints are C, M, E, F, G and H, and the only unsatisfied constraint is C11, and the changed variables are C, E and H. However, in this case, C11 cannot be satisfied anymore since both of the variables directly related to C11 are fixed beforehand, and C11 are unsatisfied by those values. This is a case that distribution of values does not lead to any consistent ICSP and the algorithm has to backtrack to try another way to satisfy the unsatisfied constraints.

Through the backtracking process, C8 must be satisfied in a different way: activating variable I. By activating variable I, as in Figure 2.5, this variable will be added to the previously fixed and changed variables, and another unsatisfied constraint will be appeared in the ICSP, constraint C12, which Solve procedure must be called for that as well. Based on the parameters, C12 can only be satisfied by deactivating N, Figure 2.6.

By deactivating N, another constraint will become unsatisfied sequentially, con-

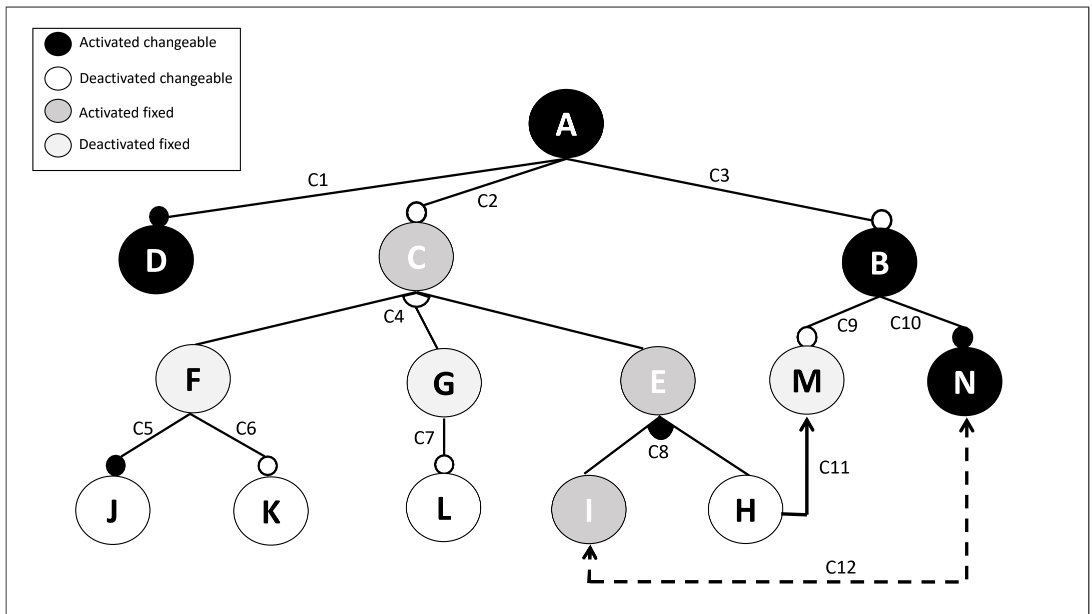

Figure 2.5: Satisfying C8 by calling Solve procedure for the ICSP shown in Figure 2.3, where C_u=C8, V_f = (C, E, F, G, I, M), and V_x= (C, E, I, M). C8 is decided to be satisfied by activating variable I.

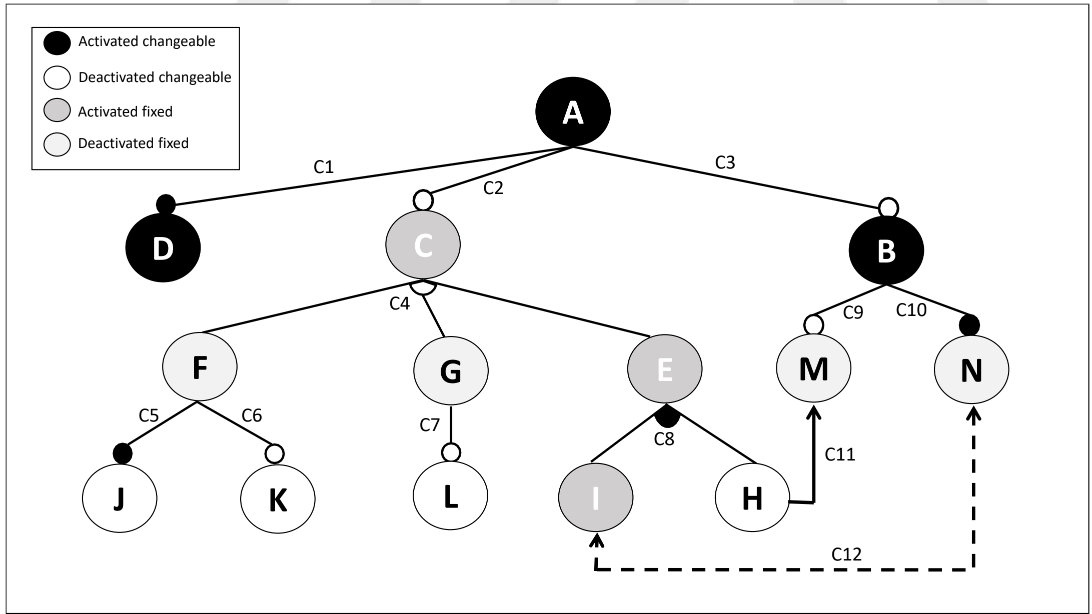

Figure 2.6: Satisfying C12 by calling Solve Procedure for the ICSP shown in Figure 2.5, where C_u=C12, V_f = (C, E, F, G, I, M, N), and V_x= (C, E, I, M, N). C12 is decided to be satisfied by deactivating variable N.

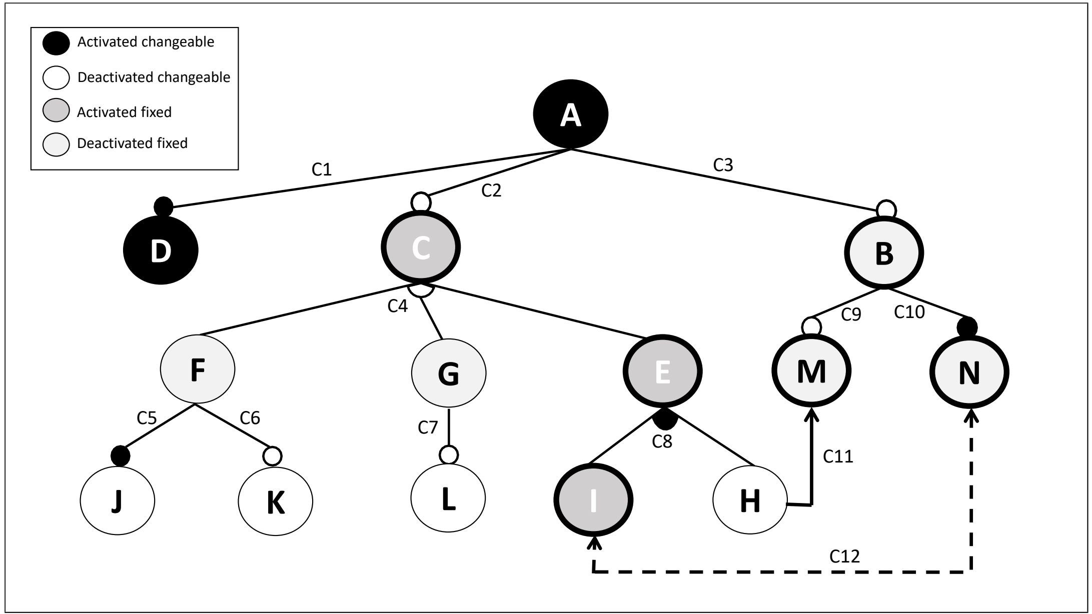

Figure 2.7: Satisfying C10 by calling Solve procedure for the ICSP shown in Figure 2.6, where C_u=C10, V_f = (B, C, E, F, G, I, M, N), and V_x= (B, C, E, I, M, N). C10 is decided to be satisfied by deactivating variable B.

straint C10. Therefore, by adding N to the previously fixed and changed variables and having C10 as the only unsatisfied constraint, the Solve procedure will be called one more time. Constraint C12 can only be satisfied in this condition by deactivating B, as in Figure 2.7. By this deactivation, the fixed variables will be C, M, E, F, G, I, N and B which among them C, E, I, M, N and B are changed, and there is no unsatisfied constraint in the system anymore. This is a case that variable value distribution leads to a consistent ICSP. The changed variables determine the solution, which is also the first solution here with 6 members, i.e. cost 6 here. This is the first solution and the best solution up to now.

After finding the first solution, the algorithm continues to search the solutions with fewer number of changes. Then, the algorithm backtracks to the first constraint that it can be satisfied in a different way, constraint C4. At this time, to satisfy the constraint C4, G will be activated and E and F will be deactivated, Figure 2.8. By this action, the fixed variables will be C, M, E, F and G, which among them C, G and M are changed, and there is not any unsatisfied constraint in the system anymore. It means this is a new solution

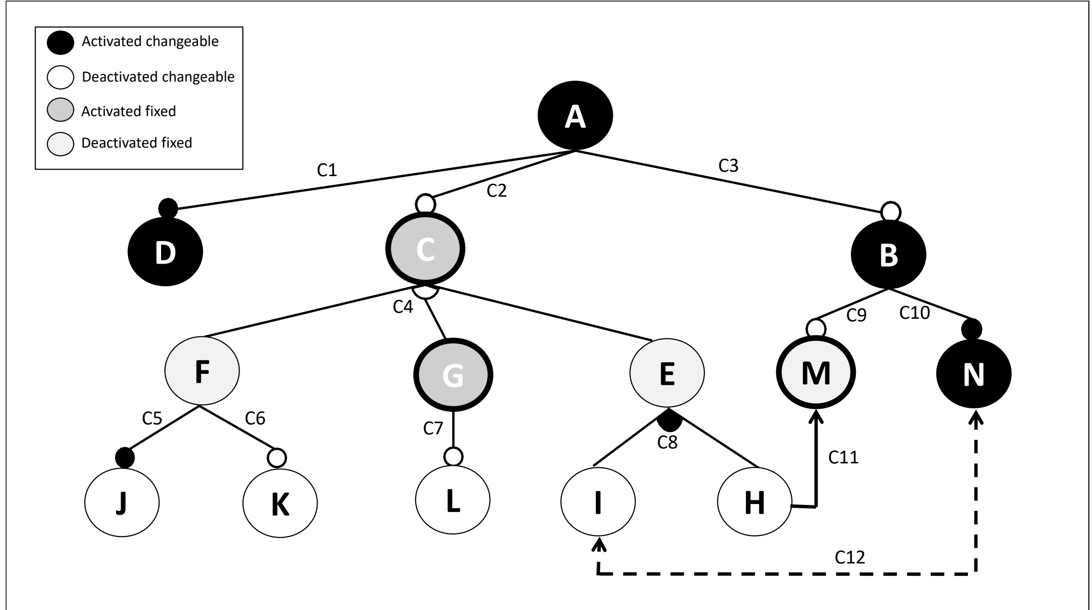

Figure 2.8: Satisfying C4 by calling Solve procedure for the ICSP shown in Figure 2.2 , where C_u=C4, V_f = (C, E, F, G, M), and V_x= (C, G, M). C4 is decided to be satisfied by Activating G and deactivating F and E.

with three changes. Because this solution has fewer number of changes, less cost, than the previously found solution, this solution will be saved as a best solution up to now.

After finding the new solution the algorithm continues to find solutions with fewer number of changes, if any. Thus, it backtracks one more time and satisfies the constraint C2 in a different way by activating F, and deactivating E and G, Figure 2.9. By this action the fixed variables will be C, M, E, F and G, which variable C, M and F among them are changed, and there is also an unsatisfied constraint in the system, constraint C5. To satisfy C5, Solve procedure must be called again, but this branch will not be traversed by the algorithm. This branch will be pruned because in this branch, up to now, three changes occurred, i.e. cost three, and the previously found solution by the algorithm also imposes cost three as well. It means that this branch will not lead to a solution better than the found one, and, as a result, it can be pruned.

At the end, because there is no other branch to backtrack, the last found solution would be returned as the best reconfiguration solution. The Figure 2.10 shows

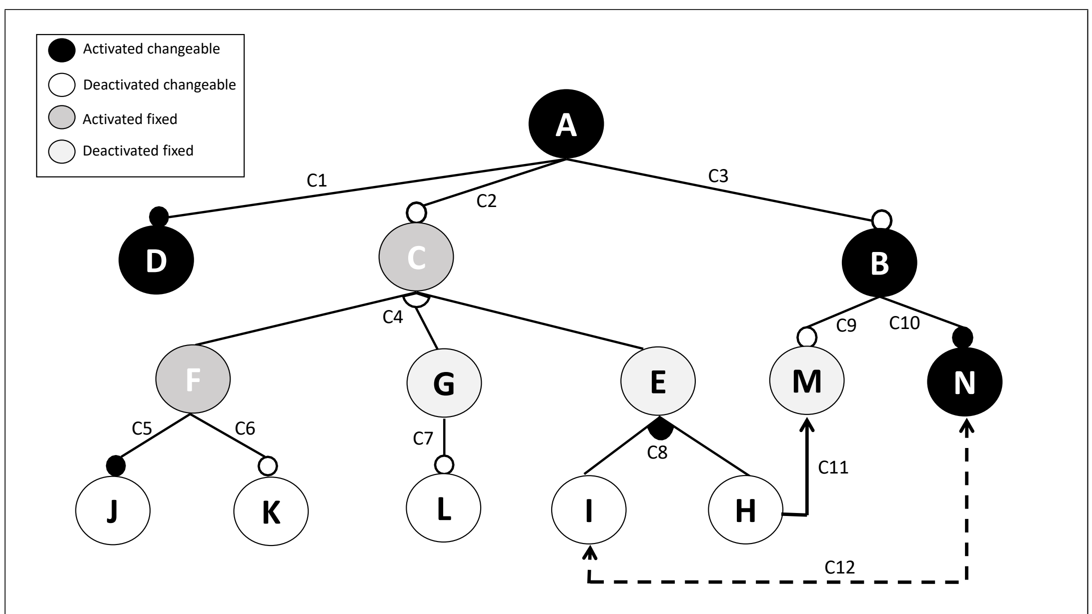

Figure 2.9: Satisfying C4 by calling Solve procedure for the ICSP shown in Figure 2.2, where C_u=C4, V_f = (C, E, F, G, M), and V_x= (C, E, M). C4 is decided to be satisfied by Activating F and deactivating G and E.

the new configuration of the system shown in Figure 1.3 after receiving the reconfiguration request to activate feature C and deactivate feature M with the minimum reconfiguration cost.

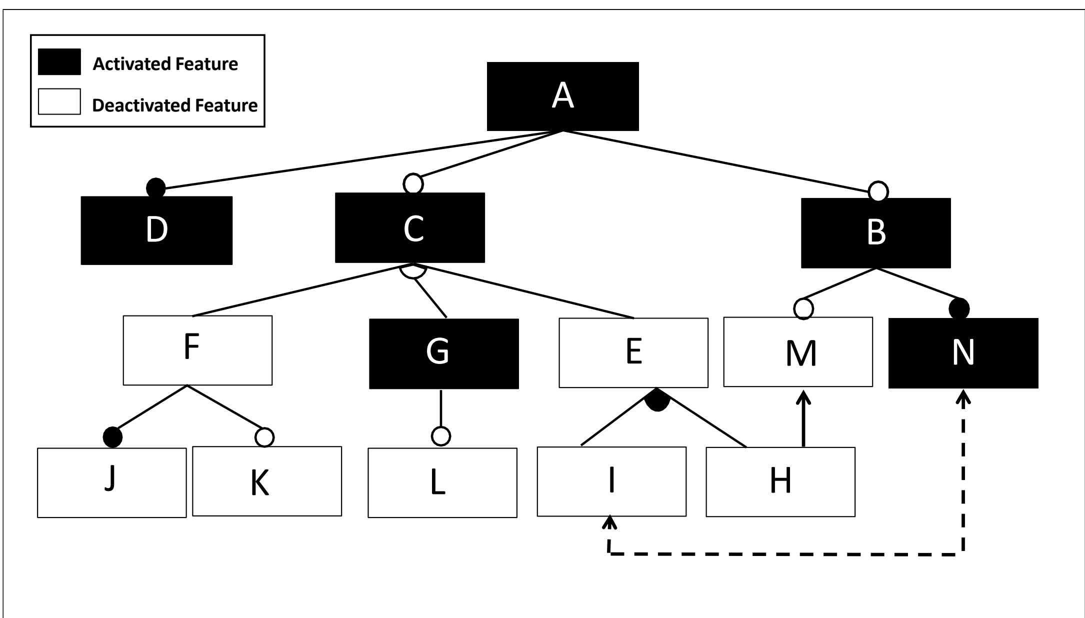

Figure 2.10: The new configuration of the system shown in Figure 1.3 after responding to the request to activate C and deactivate M with the minimum reconfiguration cost

CHAPTER 3

EXPERIMENTS

To see how the algorithm and heuristic methods can influence the performance of the algorithm, we implemented our algorithm and the heuristics to compare it with the previous algorithm, CSOP algorithm[39]. For its implementation, we used the idea of multi-directional methods which is used in other incremental solvers such as SkyBlue[32].

3.1 Multi-directional methods

The idea of multi-directional methods is defining various approaches for each constraint to satisfy the constraint in different conditions. All of the variables in the constraint network are assigned with values while using the idea of multi-directional methods, and all of the constraints are satisfied with those variable values. In this concept, every method includes a set of variables as inputs and outputs which are a subset of constraint's directly related variables. By changing the value of an input variable from a method, that method will be invoked. By a method invocation, the values corresponding to the output variables of that constraint will be determined with new values. Changing the value of an input variable can make the constraint directly related to that unsatisfied, and invoking one of those constraint's methods which the input variable of that is changed will make the constraint satisfied again. For example, the methods shown in Table 3.1 can be used for the constraint "$X = Y \wedge Y$". According to this table, for example when (X,Y,Z)=(True, True, True) and when the value of X changes to False, invoking one the methods M2, M3 or M4 will make the constraint "X

Table 3.1: Multi-directional methods for the constraint X = Y ∧ Y

Method Name	Input	Output
M1	X = True	(Y, Z) = (True, True)
M2	X = False	(Y, Z) = (False, True)
M3	X = False	(Y, Z) = (True, False)
M4	X = False	(Y, Z) = (False, False)
M5	Y = True	(X, Z) = (True, True)
M6	Y = True	(X, Z) = (False, False)
M7	Y = False	(X, Z) = (False, True)
M8	Y = False	(X, Z) = (False, False)
M9	Z = True	(X, Y) = (True, True)
M10	Z = True	(X, Y) = (False, False)
M11	Z = False	(X, Y) = (False, True)
M12	Z = False	(X, Y) = (False, False)

= Y ∧ Y" satisfied again.

3.2 Execution results

To compare our algorithm and the heuristics with the former approaches, the performance of all of the approaches should be monitored while applying them on a feature model. Because here the feature model will be mapped to it corresponding CSP, the comparison of the CSP algorithms would be more distinguishable if the size of the feature model (features and constraints among them) is big enough. On the other hand, to make the experiments more practical, it would be preferable to choose a real feature model to apply the algorithms. However, real feature models with significant number of features and relations are valuable asset of companies and private sectors, and they are reluctant to easily publish their products' feature model publicly. Therefore, since we could not find a real feature model with considerable number of features and relations, we decided to generate random feature models for that purpose[1]. To construct a pseudo-random feature model, we generated random tree structures feature

Table 3.2: Dynamic feature model reconfigurations using different algorithms in a feature model including 60 features and 40 feature model relations for requests including 3 features[1]

	Optimal Solution Cost	First Solution Cost			First Solution Time (μs)			Optimal Solution Time (ms)		
		CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic
Request 1	12	23	16	16	99.70	7.78	5.14	1721.5	123.5	12.2
Request 2	2	14	14	5	8.04	6.35	0.37	44.1	8.0	1.0
Request 3	12	16	16	13	0.51	3.47	0.65	5.6	3.9	102.8
Request 4	9	20	14	14	4.42	0.14	0.39	1342.5	2.2	1.0
Request 5	6	19	9	6	0.53	0.10	0.16	84.4	0.4	0.3
Request 6	5	8	6	6	0.54	0.17	0.37	7.5	0.2	0.4
Request 7	2	24	10	5	0.68	0.07	0.11	1555.9	0.2	0.1
Request 8	3	11	7	3	5.61	0.05	0.06	10.8	0.1	0.1
Request 9	3	13	7	3	0.41	0.07	0.07	6.2	0.1	0.1
Request 10	8	21	12	8	0.48	0.10	0.15	454.5	0.3	0.1
Average	**6.2**	**16.9**	**11.1**	**7.9**	**12.09**	**1.83**	**0.74**	**523.3**	**13.9**	**11.8**

models using pseudo-random functions. Then we generated random Require and Exclude relations among these features. Next to it, supposing an equal cost for changing the state of all of the features, we generated 10 random reconfiguration requests including different features regarding the feature models. To find the reconfiguration solutions in the systems for each request, we used the commonly used CSOP algorithm, our algorithm which we suppose it a Dynamic CSOP (DCSOP) algorithm, and our algorithm with heuristics.

In our experiment, the first request is sent to the system when the system is deactivated- i.e., all features are deactivated, and then, the results of executing the algorithms are monitored. After that, we reconfigure the system to another configuration (proposed by the algorithm with heuristics), and go for the next request and so on. The algorithms' implementation is with Java language and the supported hardware for its execution has an Intel(R) Core(TM) Duo T9550 CPU and 4.00 GB RAM. The reconfiguration results are shown in Tables 3.2 through 3.5.

We executed the algorithms on four feature models including 60, 80, 100, and 120 features. Mainly, we wanted to measure the algorithm's time to reach an optimal solution in the constraint network. This criterion shows how fast the reconfiguration can happen with the least cost in the DSPL. On the other hand, the real systems may not have time to reach the exact best solution to reconfigure. Therefore, in practice, a solution between the first and optimal solution can be used for reconfiguration according to the time restrictions that the system is

Table 3.3: Dynamic feature model reconfigurations using different algorithms in a feature model including 80 features and 47 feature model relations for requests including 5 features[1]

	Optimal Solution Cost	First Solution Cost			First Solution Time (μs)			Optimal Solution Time (ms)		
		CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic
Request 1	10	22	11	11	66.44	1.90	4.94	11028.1	73.2	7.0
Request 2	7	20	18	18	71.85	1.06	0.81	2646.7	14.8	2.0
Request 3	3	21	13	13	3965.64	0.40	1.04	5397.3	37.4	22.3
Request 4	2	21	13	2	102.08	0.09	0.13	552.8	0.6	0.3
Request 5	7	19	18	7	1441.32	0.10	4.43	3478.0	1.4	5.4
Request 6	2	21	13	2	0.30	0.06	0.08	18.6	0.1	0.1
Request 7	13	26	19	14	31.14	0.25	0.57	848.7	10.9	1.7
Request 8	14	27	20	14	133.17	0.10	10.81	174.3	0.8	11.1
Request 9	13	21	16	13	197.41	0.10	0.22	1101.3	1.7	1.1
Request 10	4	27	25	4	39.87	0.26	0.49	10998.4	1.2	0.5
Average	**7.5**	**22.5**	**16.6**	**9.8**	**604.92**	**0.43**	**2.35**	**3624.4**	**14.2**	**5.1**

Table 3.4: Dynamic feature model reconfigurations using different algorithms in a feature model including 100 features and 65 feature model relations for requests including 7 features[1]

	Optimal Solution Cost	First Solution Cost			First Solution Time (μs)			Optimal Solution Time (ms)		
		CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic
Request 1	15	23	23	23	181.11	4.77	9.93	3232.3	1097.4	111.0
Request 2	6	13	7	7	7.71	0.14	0.37	76.3	0.3	0.5
Request 3	10	23	12	12	20.66	0.17	0.55	788.0	14.9	2.4
Request 4	11	24	12	12	2495.36	0.10	0.22	2764.9	0.9	4.0
Request 5	10	34	31	10	2.15	0.46	0.15	1414.3	7.1	2.1
Request 6	9	28	26	9	6.44	0.41	0.26	102.7	2.9	0.5
Request 7	9	26	13	13	39.10	0.07	0.17	585.3	0.1	0.3
Request 8	13	29	14	14	33.13	0.09	0.18	4747.5	0.5	0.5
Request 9	5	23	21	5	50.21	0.18	0.15	136.3	0.4	0.2
Request 10	9	24	9	9	867.74	0.05	0.12	920.4	0.1	0.1
Average	**9.7**	**24.7**	**16.8**	**11.4**	**370.36**	**0.64**	**1.21**	**1476.8**	**112.4**	**12.1**

Table 3.5: Dynamic feature model reconfigurations using different algorithms in a feature model including 120 features and 69 feature model relations for requests including 9 features[1]

	Optimal Solution Cost	First Solution Cost			First Solution Time (μs)			Optimal Solution Time (ms)		
		CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic
Request 1	28	31	33	33	34.08	1.15	20.57	over 20000	18475.4	2252.9
Request 2	12	24	18	13	3.15	0.18	0.22	over 20000	1.4	0.7
Request 3	13	43	19	13	0.57	0.28	1.81	17427.9	2.1	2.6
Request 4	20	35	29	21	595.69	0.15	17.49	1593.3	22.0	20.5
Request 5	20	44	25	23	1.11	0.20	0.22	over 20000	5.4	1.5
Request 6	15	41	19	18	0.61	0.30	0.30	4499.0	5.8	5.3
Request 7	15	46	19	17	0.55	0.12	0.21	136.6	0.4	0.6
Request 8	11	37	16	11	64.39	0.10	0.10	1332.1	2.1	0.3
Request 9	12	33	18	12	0.61	0.13	0.18	3868.2	0.3	0.2
Request 10	18	44	21	18	26.82	0.11	0.19	5265.6	0.7	1.0
Average	**16.4**	**37.8**	**21.7**	**17.9**	**72.75**	**0.27**	**4.12**	**over 9000**	**1851.5**	**228.5**

Table 3.6: The average results of executing algorithms on the sample feature models

Sample Feature model	Extra Cost of First Solution on Average (%)			First Solution Time on Average (μs)			Optimal Solution Time on Average (ms)		
	CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic	CSOP	DCSP	Heuristic
60 features and 40 relations	172	79	27	12.09	1.83	6.74	523.3	13.9	11.8
80 features and 47 relations	200	121	31	604.92	0.43	2.35	3624.4	14.2	5.1
100 features and 65 relations	155	73	17	370.36	0.64	1.21	1476.8	112.4	12.1
120 features and 69 relations	130	32	7	72.75	0.27	4.12	over 9000	1851.5	228.5
Average	**164**	**76**	**20**	**265.03**	**0.79**	**3.60**	**over 3500**	**498.0**	**64.37**

confronted. Thus, we measure the first solution quality and its reaching time as an algorithm performance criterion for this purpose. To measure the quality of the first found solution by the algorithms, we consider the extra cost of the first solution regarding an optimal solution cost, using the formula in equation 3.1.

$$Extra\ cost\ of\ first\ solution = \frac{first\ solution\ cost - optimal\ solution\ cost}{optimal\ solution\ cost} \times 100 \tag{3.1}$$

Table 3.6 includes the average results of executing the CSOP algorithm and our algorithm with and without heuristics on the mentioned four sample feature models. According to this table, the commonly used CSOP algorithm, reaches an optimal reconfiguration solution in over 3.5 seconds on average. However, this time is reduced more than 85% on average by using the DCSOP algorithm. The reason is that in DCSOP algorithm, the CSP is not solved from scratch for each request. Therefore, the algorithm can saves the time to reach the first solution. In addition, using the feature model heuristics for the DCSOP algorithm reaches an optimal solution finding time to about 64 milliseconds on average, which is more than 95% reduction in time regarding CSOP algorithm. These results show how using the current state of the system and the feature model heuristics can improve the performance of finding an optimal reconfiguration solution in DSPLs.

The time for reaching a first solution and its quality is also observed in our experiments, shown in Table 3.6. CSOP algorithm, reaches the first reconfiguration solution on average in about 265 micro seconds which demands 164% more cost with respect to an optimal reconfiguration solution. However, DCSOP algorithm reduces this time by more than 99% on average, where it demands averagely 76% more cost than that of an optimal solution. It shows how the DCSP

algorithms can improve the reaching time to the first solution and the quality of the first solution. In addition, using feature model heuristics on the DCSOP algorithm results in costing only 20% more on average in the first solution than that of an optimal solution. Applying feature model heuristics can also be very beneficial in reaching a high quality first solution but with a negligible additional time over the base DCSOP algorithm (resultant of calculating heuristics).

CHAPTER 4

CONCLUSION

In this thesis we proposed using a Dynamic CSP Algorithm for finding an optimum solution for the reconfiguration of products derived from a feature model. DSPL optimization problems are of great importance in real-time DSPLs to optimize the response time, in Cloud-based DSPL applications to minimize the service utilization costs, in DSPL cyber-physical systems to minimize degradation of sensors and hardwares and so on. Our algorithm uses the basic idea of the DCSP solution-reuse algorithm proposed by Verfaillie et al. [38] along with CSOP algorithms [39] . We mentioned some characteristics of feature models when they are mapped to a CSP and proposed some heuristics related to them. Although all of the examples in this work are about basic feature models, the algorithm can be used for extended feature models and complex constraints as well. One of the main things while devising our algorithm was considering the system evolution. The algorithm can also be used when the DSPL system evolves, and the set of features and/or constraints changes.

We also use feature model heuristics for utilizing a solution reuse approach for dynamic reconfiguration. Exploring characteristics of feature models in DSPL algorithms that reuse reasoning[8, 9, 13, 35], searching solutions resistant to changes[16, 34, 17, 40, 18] or solutions easily adaptable to changes[21, 2, 20] will be our fields of interest in future work. Although our heuristics are only tailored for feature models, there are other several approaches reported in the literature for variability modeling. Finding heuristics regarding the CSPs obtained from other variability modeling approaches would be beneficial as well. For instance, goal-modeling[33] is one of the approaches used for modeling an SPL and its

context. Translations from other approaches such as goal-modeling to feature modeling in order to benefit from the results of this work would be an interesting topic as future work.

REFERENCES

[1] *FeatureModels-ReconfigurationRequests-Results*, 2018 (accessed January 7, 2018). `https://github.com/sina-entekhabi/FeatureModels--Requests--Results`.

[2] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic csps—application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[3] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[4] R. Barták, T. Muller, and H. Rudová. Minimal perturbation problem-a formal view. *Neural Network World*, 13(5):501–512, 2003.

[5] A. Bellicha. Maintenance of solution in a dynamic constraint satisfaction problem. *WIT Transactions on Information and Communication Technologies*, 2, 1970.

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[7] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *SPLC (2)*, pages 23–32, 2008.

[8] C. Bessiere. Arc-consistency in dynamic constraint satisfaction problems. In *AAAI*, volume 91, pages 221–226, 1991.

[9] C. Bessiere. Arc-consistency for non-binary dynamic csps. In *ECAI*, volume 92, pages 23–27, 1992.

[10] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10), 2009.

[11] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10), 2009.

[12] P. Clements and L. Northrop. *Software product lines.* Addison-Wesley, 2002.

[13] R. Debruyne. Arc-consistency in dynamic csps is no more prohibitive. In *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, pages 299–306. IEEE, 1996.

[14] R. Dechter and A. Dechter. *Belief maintenance in dynamic constraint networks.* University of California, Computer Science Department, 1988.

[15] B. Faltings and S. Macho-Gonzalez. Open constraint satisfaction. *CP*, 2470:356–370, 2002.

[16] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104. Springer, 1993.

[17] H. Fargier, J. Lang, and T. Schiex. Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge. In *AAAI/IAAI, Vol. 1*, pages 175–180, 1996.

[18] D. Fowler and K. Brown. Branching constraint satisfaction problems for solutions robust under likely changes. *Principles and Practice of Constraint Programming–CP 2000*, pages 500–504, 2000.

[19] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[20] E. Hebrard, B. Hnich, and T. Walsh. Super CSPs. In *Proc. of the CP-03 Workshop on "Handling Change and Uncertainty"*, 2003.

[21] P. Jégou. A logical approach to solve dynamic csps: Preliminary report. In *ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, pages 87–94, 1994.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[23] A. S. Karataş, H. Oğuztüzün, and A. Doğru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 78(12):2295–2312, 2013.

[24] E. Lamma, P. Mello, M. Milano, R. Cucchiara, M. Gavanelli, and M. Piccardi. Constraint propagation and value acquisition: why we should do it interactively. In *IJCAI*, volume 99, pages 467–473, 1999.

[25] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Context aware reconfiguration in software product lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 41–48. ACM, 2016.

[26] R. Mazo, C. Salinesi, and D. Diaz. Variamos: a tool for product line driven systems engineering with a constraint based approach. In *24th International Conference on Advanced Information Systems Engineering (CAiSE Forum'12)*, 2012.

[27] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction. In *Proceedings eighth national conference on artificial intelligence*, pages 25–32, 1990.

[28] Y. Ran, N. Roos, and J. van den Herik. Approaches to find a near-minimal change solution for dynamic csps. In *Fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimisation problems*, pages 373–387, 2002.

[29] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel. Tailoring dynamic software product lines. In *ACM SIGPLAN Notices*, volume 47, pages 3–12. ACM, 2011.

[30] L. E. Sanchez, J. A. Diaz-Pace, A. Zunino, S. Moisan, and J.-P. Rigault. An approach based on feature models and quality criteria for adapting component-based systems. *Journal of Software Engineering Research and Development*, 3(1):10, 2015.

[31] L. E. Sanchez, S. Moisan, and J.-P. Rigault. Metrics on feature models to optimize configuration adaptation at run time. In *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, pages 39–44. IEEE Press, 2013.

[32] M. Sannella. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146. ACM, 1994.

[33] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes. Using constraint programming to manage configurations in self-adaptive systems. *Computer*, 45(10):56–63, 2012.

[34] T. Schiex, H. Fargier, G. Verfaillie, et al. Valued constraint satisfaction problems: Hard and easy problems. *IJCAI (1)*, 95:631–639, 1995.

[35] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(02):187–207, 1994.

[36] P. Van Hentenryck and T. Le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9(3):257–275, 1991.

[37] G. Verfaillie and T. Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *In Proceedings of the ECAI-94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications*. Citeseer, 1994.

[38] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI*, volume 94, pages 307–312, 1994.

[39] K. Vermirovsky. Algorithms for constraint satisfaction problems. Master thesis, Masaryk University Brno Faculty Of Informatics, April 2003. 4th section.

[40] R. Wallace and E. Freuder. Stable solutions for dynamic constraint satisfaction problems. *Principles and Practice of Constraint Programming — CP98*, pages 447–461, 1998.

[41] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.