Improving Post-Deployment Configuration of Cyber-Physical Systems Using Machine Learning and Multi-Objective Search

Safdar Aqeel Safdar

Thesis submitted for the degree of Ph.D.

Department of Informatics, Faculty of Mathematics and Natural Sciences
University of Oslo, 2020

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2379

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

Abstract

Today, Cyber-Physical Systems (CPSs) are increasingly becoming an essential part of our daily lives and can be found in various domains such as energy, communication, and logistics. To accommodate different needs of users and provide customizations, CPS producers often adopt Product Line Engineering (PLE) methodologies. Consequently, CPSs are developed by integrating multiple products within/across product lines (PLs) that communicate with each other through information networks. Several PLE methodologies exist in the literature, however, their suitability for CPS PLs needs to be evaluated because of unique characteristics of CPS PLs (e.g., variabilities corresponding to multiple domains (e.g., electronics, mechanics), complex configuration processes). Hence, we need to identify key requirements of CPS PLE and evaluate existing PLE methodologies to assess their capabilities of supporting CPS PLE. Furthermore, most of the existing studies address challenges related to the pre-deployment configuration (i.e., making configuration decisions at design time) of individual products. There is a need for studies focusing on the post-deployment configuration (i.e., making configuration decisions at runtime) of interacting products.

In this thesis, first, we conducted a systematic domain analysis and proposed a conceptual framework for CPS PLs, based on which we evaluated existing PLE methodologies. Then, we focused on the post-deployment configuration of CPSs and made another two contributions: we proposed 1) an approach to capture patterns of configurations in the form of configuration rules and, and 2) another approach for recommending configurations to improve the post-deployment configuration experience from the perspective of testers and end-users.

To conduct the domain analysis, we analyzed three real-world CPS case studies. Based on the knowledge collected from the domain analysis and a thorough literature review on PLE, we proposed a conceptual framework, in which we 1) clarify the context of CPS PLE by formalizing CPSs, PLE, and configuration process; 2) present classifications of Variation Point (VP), constraint, and view types in addition to other modeling requirements to support the domain engineering of CPS PLs; and 3) formalize various types of automation that can be enabled to support the application engineering of CPS PLs. The completeness of the framework was evaluated using three real-world case studies containing 2161 VPs, 3943 constraints, and 40 views, 11 configuration tools, and an extensive literature review. Furthermore, we also evaluated four representative variability modeling techniques (VMTs): Feature Model (FM), Cardinality-Based Feature Model (CBFM), Common Variability Language (CVL), and SimPL. With the selected VMTs, we modeled a case study to assess if they can capture variabilities of CPS PLs. Results show that using SimPL, CVL, CBFM, and FM, we can capture only 81%, 75%, 50%, and 15% of the total variabilities, respectively.

To capture the configuration patterns in the form of configuration rules, we proposed the Search-Based Rule Mining (SBRM$^+$) approach. SBRM$^+$ combines multi-objective search with machine learning to mine configuration rules in an incremental and iterative way. We evaluated the performance of SBRM$^+$ using multiple real-world and open-source case studies from the communication domain and compared its performance with Random Search Based Rule Mining (*RBRM*$^+$). Results show that *SBRM*$^+$ performed significantly better than *RBRM*$^+$ in terms of fitness values, six quality indicators, and 17 Machine Learning Quality Measurements MLQMs. As compared to *RBRM*$^+$, *SBRM*$^+$ improved the quality of rules up to 28% in terms of MLQMs.

To improve the post-deployment configuration experience, we proposed the Search-Based Configuration Recommendation (SBCR) approach, which recommends faulty configurations for CPSs with interacting products under test, based on mined rules. These configurations can be used to test CPSs and create guidelines for end-users to improve the post-deployment configuration experience. We evaluated SBCR using the same case studies, for which we mined the rules using SBRM$^+$. Results show that SBCR significantly outperformed Random Search-Based Configuration Recommendation (RBCR) in terms of six quality indicators and the percentage of faulty configurations. Overall, SBCR made up to 22% more accurate recommendations than RBCR.

Acknowledgements

Postponing the idea of having my own start up and going into research to undertake Ph.D. has been a transformative experience, which allowed me to be who I am today! …And of course, this would not have been possible without persuasiveness of Muhammad Zohaib Iqbal to whom I owe a great deal of gratitude.

I am also sincerely grateful to my supervisors Tao Yue and Shaukat Ali. Without their continuous support, encouragement, and invaluable knowledge and expertise in software engineering research, this work would not have been possible. I am also thankful to Hong Lu for her constructive feedback, insights, and many discussions that we had during my Ph.D. years. I would also like to acknowledge the Norwegian Research Council for funding this PhD and providing me an opportunity to conduct the research. A special acknowledgment to Simula Research Laboratory and Simula School of Research and Innovation for providing an excellent workplace.

I would like to thank my family and friends in particular Noman Bashir for their continuous support and encouragement. I would also like to show my deepest gratitude to all the colleagues at Simula especially my office mates Helge Spieker and Man Zhang. I would also like to express my gratitude to Shuai Wang for all positive pep talks. Last but not the least, a big thanks to Dipesh Pradhan for being a great friend and a continuous source of motivation as well as for all the fun time and discussions we had together during my Ph.D.

To the people with dreams..!

Abbreviation

CPS	Cyber-Physical Systems
PLE	Product Line Engineering
PL	Product Line
VP	Variation Point
VMT	Variability Modeling Technique
FM	Feature Model
CBFM	Cardinality-Based Feature Model
CVL	Common Variability Language
CPL	Cross-Product Line
SBRM	Search-Based Rule Mining
RBRM	Random Search Based Rule Mining
RDBRM	Real Data Based Rule Mining
MLQM	Machine Learning Quality Measurement
SBCR	Search-Based Configuration Recommendation
RBCR	Random Search-Based Configuration Recommendation
VCS	Video Conferencing System
SPL	Software Product Line
MHS	Material Handling System
SPS	Subsea Production System
SBSE	Search-Based Software Engineering
GA	Genetic Algorithm
EA	Evolutionary Algorithm
NSGA	Non-dominated Sorting Genetic Algorithm
MoCell	Multi-objective Cellular Genetic Algorithm
IBEA	Indicator-based Evolutionary Algorithm
SPEA2	Improved Strength Pareto Evolutionary Algorithm
PAES	Pareto Archived Evolution Strategy
SMPSO	Speed-constrained Multi-objective Particle Swarm Optimization
RS	Random Search
RIPPER	Repeated Incremental Pruning to Produce Error Reduction
PART	Pruning Rule-Based Classification algorithm
HV	Hypervolume
IGD	Inverted Generational Distance
ED	Euclidean Distance from the Ideal Solution
GD	Generational Distance
GS	Generalized Spread
ARI	Average Relative Improvement

PFC	Percentage of Faulty Configurations

List of paper

The following papers are included in this thesis:

Paper A. Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering
Safdar Aqeel Safdar, Tao Yue, Shaukat Ali, and Hong Lu.
Published in the Proceedings of International Conference on *System Analysis and Modeling (SAM)*, 2016.

Paper B. A Framework for Automated Multi-Stage and Multi-Step Product Configuration of Cyber-Physical Systems
Safdar Aqeel Safdar, Hong Lu, Tao Yue, Shaukat Ali, and Kunming Nie.
Published in the Journal of *Software and Systems Modeling (SoSym)*, 2020.

Paper C. Mining Cross Product Line Rules with Multi-Objective Search and Machine Learning
Safdar Aqeel Safdar, Hong Lu, Tao Yue, and Shaukat Ali.
Published in the Proceedings of *The Genetic and Evolutionary Computation Conference (GECCO)*, 2017.

Paper D. Combining Multi-Objective Search with Machine Learning to Infer Cross Product Line Rules for Interacting Products
Safdar Aqeel Safdar, Tao Yue, Shaukat Ali, and Hong Lu.
Published in the Journal of *Automated Software Engineering (ASE)*, 2019.

Paper E. Recommending Faulty Configurations for Interacting Systems Under Test Using Multi-Objective Search
Safdar Aqeel Safdar, Tao Yue, and Shaukat Ali.
Submitted to the Journal of *Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

Note that all the five papers are self-contained and thus some information might be redundant across different papers. Also, different abbreviations may have been used in the papers.

My contributions

For all the five papers mentioned above, I am the main contributor for the idea, implementation, case study design, experimentation, results and analysis, and paper writing. My supervisors Tao Yue and Shaukat Ali were involved throughout different phases of the work. Hong Lu was also involved in scientific discussions for idea development, experiment designs, and paper writing for Paper A-D. Moreover, during my Ph.D., I also contributed in Paper-F, which is not included in this thesis.

Paper F. Quality Indicators in Search-based Software Engineering: An Empirical Evaluation
Shaukat Ali, Paolo Arcaini, Dipesh Pradhan, Safdar Aqeel Safdar, and Tao Yue.
Published in the Journal of *Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

Contents

PART-I: Summary

Part I
Summary

Summary

1 Introduction

Cyber-Physical Systems (CPSs) are highly connected large-scale systems that use embedded computers to monitor and control physical processes using sensors and actuators [1-5]. Communication is an integral part of these systems, where various subsystems communicate with each other through information networks (e.g., Internet). Today, such systems are increasingly becoming an essential part of our daily lives and can be found in diverse domains such as energy, communication, maritime, and logistics [6, 7]. To address different needs of users, CPSs require customizations, and thus, many CPS producers opt for Product Line Engineering (PLE) methodologies [6, 8]. Consequently, CPSs are developed by integrating multiple interacting products (i.e., subsystems of CPSs) belonging to one or more product lines (PLs). A video conferencing system (VCS) with multiple endpoints is an example of CPSs (Figure 1), where these endpoints are products belonging to one or more PLs [9]. Such systems are highly configurable, as each product has a large number of configurable parameters. For example, a VCS product developed by Cisco[1] can have more than 120 configurable parameters, offering different configuration options to users. Each product has a set of state variables defining system states and a set of operations to enable interactions among various products.

PLE has two phases: domain engineering and application engineering. Domain engineering focuses on capturing abstractions in form of commonalities and variabilities, and various types of constraints for PLs using a modeling methodology (aka variability modeling technique–VMT). Application engineering involves configuring products using a configuration tool with various types of automation to support a specific configuration process. A large number of VMTs [10-19] and configuration tools [20-26] exist in the literature, however, they are confined to traditional software product lines (SPLs) in various contexts. CPS PLs differs from traditional SPLs in many ways: 1) CPS PLs has complex variabilities, e.g., variabilities corresponding to multiple domains (e.g., electronics, software), physical properties of CPSs (e.g., length, temperature), complex interactions among different components and products, and complex topologies; 2) multiple binding times (e.g., design time, post-deployment) for captured variabilities; 3) complex constraints, e.g., dependencies across multiple domains; and 4) a complex collaborative configuration process where various domain experts from different

[1] www.cisco.com/c/en/us/products/collaboration-endpoints/index.html

department/organizations configure a part of the product during different phases of the product development lifecycle. Thus, there is a need to conduct a domain analysis for identifying key requirements of CPS PLE and evaluating existing PLE methodologies to assess their capabilities in terms of supporting CPS PLE. Moreover, most of the literature addresses challenges related to the pre-deployment configuration of individual products and lacks the studies focusing on the post-deployment configuration of interacting products.

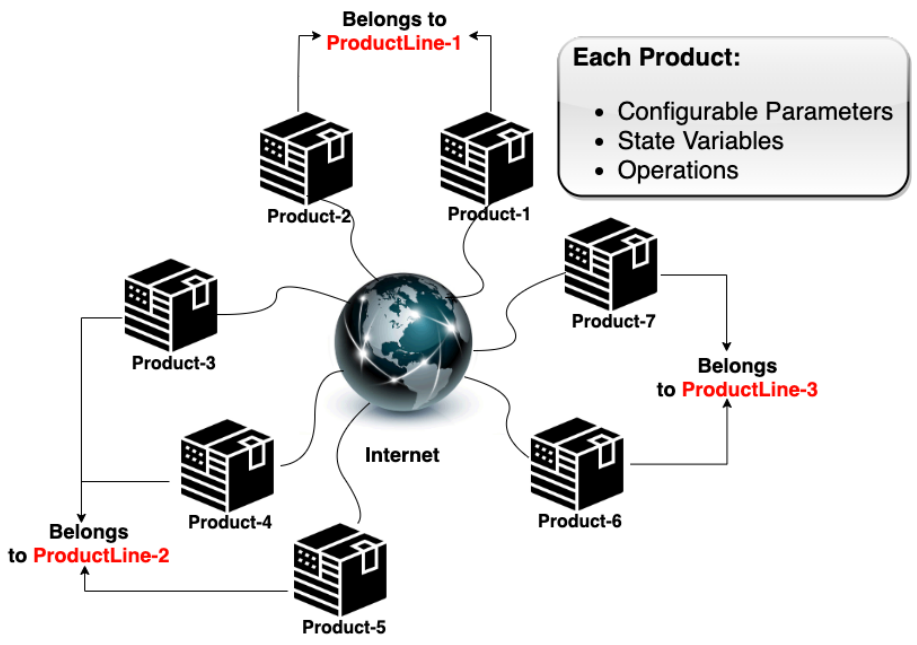


Figure 1: An example of CPSs with multiple interacting products within/across PLs

In this thesis, first, we conducted a systematic domain analysis and then proposed a conceptual framework for CPS PLs, based on which we evaluated existing PLE methodologies. After conducting a broader scope study to clarify the problem of supporting CPS PLE, we narrowed down the scope by focusing on the post-deployment configuration of interacting products constituting CPS. To this end, we proposed an approach to capture patterns of configurations in the form of configuration rules and an approach to recommend configurations for interacting products to improve the post-deployment configuration experience for testers and end-users.

To conduct domain analysis, we selected and analyzed three real-world CPS case studies: Material Handling System (MHS), Video Conferencing System (VCS), and Subsea Production System (SPS). Based on the knowledge collected from the analysis of the CPS case studies and a thorough literature review on CPS PLE, we proposed a conceptual framework to support both domain engineering and application engineering of CPS PLs (i.e., Paper-A and Paper-B in Figure E-2). The proposed framework 1) clarifies the context of CPS PLE by formalizing CPS PLE based on the PLE ISO/IEC standard for Product Line Engineering and Management [27], and multi-stage multi-step configuration process; 2) facilitates domain engineering by presenting classifications of VP types, constraint types, and view types in addition to formalizing other concepts related to modeling of CPS PLs (e.g., models, model elements, constraint types); and 3) supports application engineering by formalizing 14 possible functionalities of an automated configuration tool. We evaluated the completeness of the framework using three real-world case studies (i.e., VCS, MHS, and SPS), 11 configuration tools, and extensive literature reporting configuration automation techniques. Evaluation results show that the framework has all the necessary VP, constraint, and view types required to capture and manage variabilities and constraints of selected CPS case studies. In total, three case studies have 2161 VPs, 3943 constraints, and 40 views that can be modeled using the framework. Furthermore, 13 out of 14 functionalities in the framework are covered by at least one of the existing tools or techniques in the literature. However, none of the existing tools has all 14 functionalities. Furthermore, we selected four representative VMTs: Feature Model (FM) [28], Cardinality-Based Feature Model (CBFM) [29], Common Variability Language (CVL) [30], and the SimPL methodology [18]. With the selected VMTs, we modeled the MHS case study to assess if they fulfill requirements of CPS PLs. Evaluation results show that none of the four VMTs can capture all the CPS-specific VPs. SimPL, CVL, CBFM, and FM provide support for 81%, 75%, 50%, and 15% of the total CPS-specific VP types, respectively.

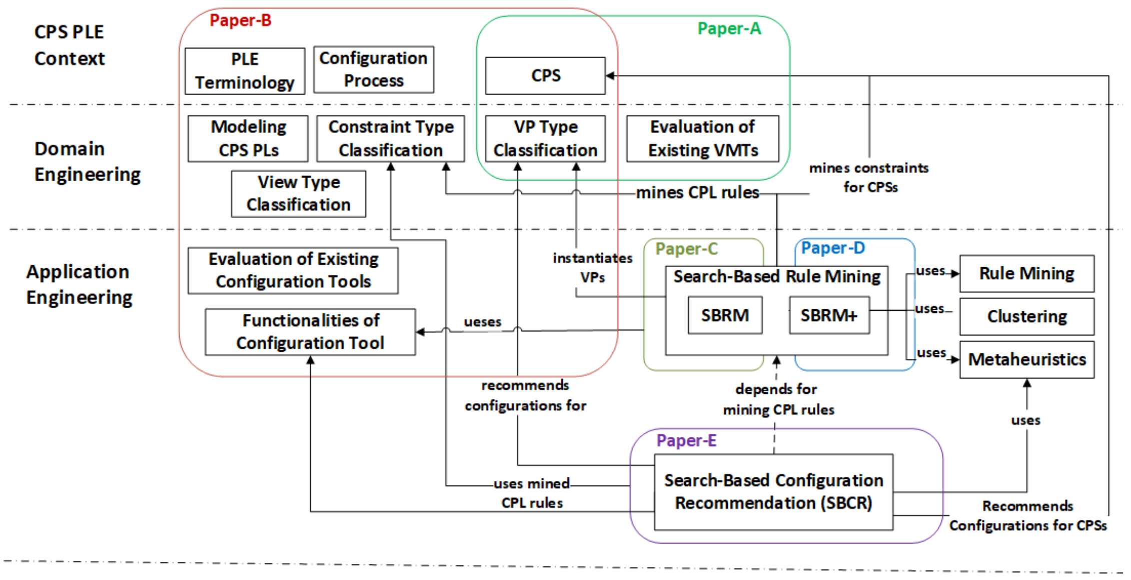


Figure 2: Overall contribution of the thesis

From the domain analysis, we noticed that CPSs have a large number of configurations [31] and their runtime behavior is dependent on the configurations of communicating products constituting CPSs as well as information networks [32, 33]. Also, there exist faulty configurations that can lead to unwanted behavior of CPSs. This requires identifying the patterns of configurations for these interacting products in the form of configuration rules, which can be used to improve post-deployment configuration in various contexts (e.g., testing). Manually specifying configuration rules based on domain knowledge is tedious and time-consuming, and heavily relies on experts' knowledge of the domain [34]. Also, certain information (e.g., network related information such as bandwidth, traffic congestion) is only known at runtime [34], which makes it impossible to specify these rules manually, merely based on the domain knowledge. This requires a sophisticated approach to automatically infer the configuration rules.

In [35], Temple et al. proposed a rule mining approach for a PL based on randomly generated and labeled (faulty or non-faulty) configurations. However, randomly generating configurations to mine rules is inefficient, as rules with all classes are not equally important (i.e., rules with faulty classes are more important than non-faulty ones). Thus, we employ search for generating configurations with three search heuristics instead of generating randomly. We proposed an approach called Search-based Rule Mining (*SBRM*), which combines multi-objective search with machine-learning techniques, to mine configuration rules (named as Cross-Product Line (CPL) rules) in an incremental and iterative way (Paper-C in Figure 2). The three search heuristics aim to generate configurations that maximally violate high confidence rules with non-faulty classes and satisfy low confidence rules with non-faulty classes and rules with faulty classes.

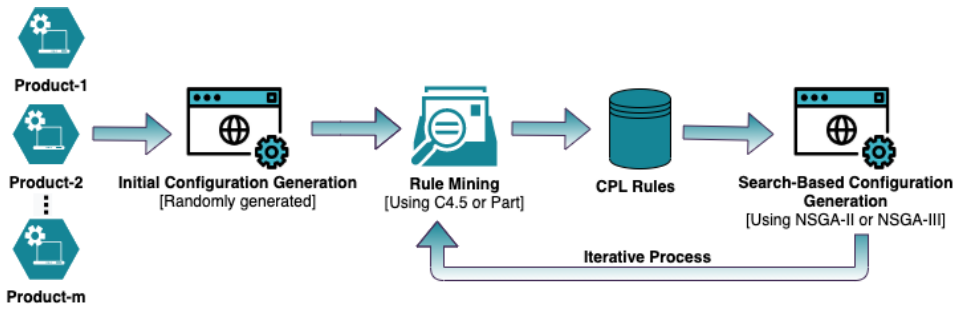


Figure 3: The overall context and scope of SBRM and SBRM⁺

SBRM has three major components (Figure 3): 1) *Initial Configuration Generation*: randomly generating an initial set of configurations for communicating products; 2) *Rule Mining*: taking the generated configurations as input along with corresponding system states and applying the machine learning algorithm to mine CPL rules; and 3) *Search-based Configuration Generation*: taking the mined CPL rules as input and generating another set of configurations using multi-objective search algorithm, which is combined with the previously generated configurations to mine a refined set of CPL rules. *SBRM* obtains CPL rules with different degrees of confidence (i.e., the probability of being correct) with an emphasis on mining rules that can reveal invalid configurations, i.e., the configurations that may lead to abnormal (i.e., unwanted) system states [36]. Instead of collecting a large amount of data required for machine learning all at once, we obtain input data incrementally over multiple iterations. During each iteration, we use rules mined from the previous iteration to guide the search for generating configurations. Newly generated configurations are combined with configurations from all the previous iterations to incrementally refine the aforementioned rules.

We evaluated *SBRM* using a real-world case study of two VCS products belonging to different PLs. Note that the systems used for experiments are real; however, the experiments were not performed in the industrial setting. The performance of *SBRM* is compared with the Random Search Based Rule Mining (*RBRM*) and Real Data Based Rule Mining (*RDBRM*) approaches, in terms of fitness values, Hypervolume (HV), and seven Machine Learning Quality Measurements (MLQMs). Results show that *SBRM* significantly outperformed *RBRM* in terms of fitness values, HV, and MLQMs. Similarly, in comparison to *RDBRM*, *SBRM* performed significantly better in terms of *Failed Precision* (18%), *Failed Recall* (72%), and *Failed F-measure* (59%).

We further refined *SBRM* (referred to as *SBRM*$^+$) (Paper IV in Figure 2), where we made the following changes: instead of using thresholds to classify the rules, we employed k-Mean clustering algorithm; integrated a search algorithm NSGA-III and a rule mining algorithm C4.5, in addition to the existing NSGA-II and PART algorithms; and conducted a thorough empirical evaluation using two case studies (Cisco and Jitsi) of relatively higher complexity. Note that for the Cisco case study, the experiments were conducted using real systems but not in the industrial setting. Evaluation results show that all the *SBRM*$^+$ approaches performed significantly better than *RBRM*$^+$ approaches in terms of fitness values, six quality indicators, and 17 MLQMs. As compared to *RBRM*$^+$ approaches, *SBRM*$^+$ approaches have improved the quality of rules based on MLQMs up to 27% for the Cisco case study and 28% for the Jitsi case study.

Since CPSs are highly configurable, testing them with all possible configurations is not possible due to limited available resources. Thus, often these systems are tested with only a few valid configurations selected randomly, based on expert's opinions, or based on some coverage criteria, such as pairwise feature coverage [37-40], which can compromise the quality of developed systems. Similarly, end-users also suffer from bad post-deployment configuration experience when proper guidelines are not available. Towards this direction, we proposed an approach called Search-Based Configuration Recommendation (*SBCR*) that makes use of previously mined CPL rules and recommends the most critical faulty configurations for CPSs. Recommended configurations can be used for testing CPSs and creating guidelines for end-users to avoid such faulty configurations and improve the post-deployment configuration experience. In *SBCR*, we defined four search heuristics based on CPL rules and combined them with six multi-objective search algorithms to find the best-suited algorithm for the configuration recommendation problem. We evaluated *SBCR* with the same two case studies for which we mined the rules in Paper-D using *SBRM*$^+$. We compared the performance of *SBCR* with Random Search-Based Configuration Recommendation (*RBCR*). Results show that *SBCR* significantly outperformed *RBCR* in terms of the six quality indicators and the percentage of faulty configurations. Overall, *SBCR* made up to 22% more accurate recommendations than *RBCR*. Among the six variants of *SBCR*, *SBCR*$_{SPEA2}$ performed the best for the faulty configuration recommendation problem.

This thesis is divided into two parts:

Part-I. Summary: This part summarizes the research work done for the entire thesis, which is organized into the following sections: In Section 2, we provide background details required to understand the thesis, followed by research methods used in Section 3. Section 4 briefly discusses the contributions of the thesis, whereas, the key results are summarized in Section 5. In Section 6, we discuss the threats to validity. Section 7 outlines future research directions, and finally, in Section 8, we conclude the thesis.

Part-II. Papers: This part presents the published or submitted research papers included in the thesis. Figure 2 gives an overview of the contribution of different papers.

2 Background

In this section, we provide the background knowledge required to understand the rest of the thesis. Section 2.1 gives a brief overview of PLE followed by an introduction to optimization problems in Section 2.2. In Section 2.3 and Section 2.4, we introduce multi-objective search and branch distance heuristic, respectively. Section 2.5 briefly discusses machine learning techniques.

2.1 Product Line Engineering (PLE)

As opposed to traditional software engineering, PLE focuses on developing a family of products (aka PL) through reuse and mass customization [27, 41, 42]. Subsequently, PLE enhances the overall quality of produced systems and the productivity of the development process while reducing the overall engineering effort and time-to-market [27, 43-45]. A product line is a set of similar products having explicitly defined common and variable features while sharing the same domain architecture. To exploit the common feature of a product line, reusable artifacts (e.g., architecture, code, test cases) are developed, which are customized and reused by various member products of the product line.

PLE has two major activities: domain engineering and application engineering. Domain engineering enables us to specify and manage reusable artifacts for a product line. To be more specific, in domain engineering, we capture abstractions as commonalities and variabilities as well as various types of constraints for the product line, using a VMT (e.g., Feature Model (FM) [28], SimPL methodology [18]). The VMT provides well-defined variation points (VPs) and constraint types to capture different types of variabilities and constraints for the product line. Moreover, the VMT also supports various views to manage and present abstractions and constraints efficiently.

Application engineering focuses on product configuration to derive the products from a product line according to the user requirements. Usually, the application engineering is supported by a configuration tool (e.g., Pure::Variants [20], Zen-Configurator [26]). The configuration tool provides different types of automated functionalities such as consistency checking [46], collaborative configuration [47], and decision inference [48].

2.2 PLE Optimization Problems

Various PLE problems such as feature selection, configuration fixing, feature model construction, and architectural improvements can be formulated as optimization problems. The purpose of formulating optimization problems is to find the best solution(s) from the set of possible solutions in terms of one or more measurements (often called objectives) to be optimized. To formulate a PLE problem as an optimization problem, we need to define: 1) a problem representation allowing symbolic manipulation, 2) a fitness function with one or more objective(s) to be optimized, and 3) manipulation operators to change the solutions (i.e., elements in the search space) [49]. The solution representation is dependent on the nature of the optimization problem. The quality of a solution is evaluated using a fitness function for guiding the search to find the optimal solution. Manipulation operators produce new solutions by either mutating the solution or exchanging parts of two solutions. An optimization problem is defined as either a minimization or maximization problem to get the minimum or maximum value of the objectives within the search space.

An optimization problem can be a single objective or multi-objective depending on the number of objectives to be optimized. Usually, multi-objective optimization problems have conflicting objectives to be optimized at the same time, thus, require analyzing tradeoffs among the objectives. Single objective optimization problems have only one optimal solution, whereas, the multi-objective optimization problems have more than one optimal solution due to tradeoffs among the objectives. Hence, for a multi-objective optimization problem, a set of solutions with equivalent quality (aka non-dominated solutions) is produced based on *Pareto dominance* and *Pareto optimality* [50-52].

Let $O = \{o_1, o_2, \dots, o_n\}$ be a set of *n* objectives and $F = \{f_1, f_2, \dots, f_n\}$ be a set of *n* objectives functions to measure the *n* objectives for a multi-objective optimization problem. In case of a minimization problem, where a lower value of an objective shows better performance, solution A *dominates* B (i.e., $A \succ B$) *iff*: $\forall_{i=1,2,\dots,n} f_i(A) \leq f_i(B) \wedge \exists_{i=1,2,\dots,n} f_i(A) < f_i(B)$. Moreover, solution A^* is *Pareto optimal* if no other solution in the feasible region Ω dominates A^*, which can be presented mathematically as: *iff* $A^* \succ C \; \forall \; C \neq A^* \in \Omega$. Note that for Pareto optimal solutions, objective values cannot be improved simultaneously, which means improving one of the objectives will worsen the other objective functions [53]. The non-dominated solutions form *Pareto-optimal* set, whereas, the corresponding objective vectors (i.e., objectives' values) make a *Pareto frontier*.

The optimization problems with large search space require specific techniques to solve them in a reasonable time. Search-Based Software Engineering (SBSE) tackles such optimization problems efficiently by applying various metaheuristics. These metaheuristics combine basic heuristics approaches in higher-level frameworks to find solutions for combinatorial problems at an acceptable computational cost [54, 55]. As mentioned earlier, the problem-specific *fitness functions* are used to guide the search to find optimal solutions from a huge search space.

Most of the PLE problems require optimizing multiple objectives at the same time, which often conflict with each other. For example, a configuration fixing problem requires dealing with multiple conflicting objectives such as minimizing the number of fixes and impact of a fix on other configurations while maximizing the configuration inference [44]. SBSE has been quite effective to solve these problems in the literature [48, 56-60].

2.3 Multi-Objective Search

Multi-objective search has been widely used in SBSE to address various optimization problems including test case prioritization, cost estimation, and configuration generation [48, 56-62]. Multi-objective search algorithms are designed to solve problems where different objectives are competing with each other and no single optimal solution exists. They aim to find a set of non-dominated solutions for trading off different objectives. Many search algorithms exist in the literature that can be applied to solve different software engineering optimization problems. In Table 1, we present a classification of the search algorithms used in this thesis.

Table 1. Classification of the selected search algorithms

Algorithm category		Algorithm
Genetic Algorithms (GAs)	Sorting based	NSGA-II
		NSGA-III
	Cellular based	MoCell
Evolutionary Algorithms (EAs)	Indicator based EA	IBEA
	Strength Pareto EA	SPEA2
	Evolution Strategies	PAES
Swarm Algorithm	Particle Swarm Theory	SMPSO

Genetic algorithms (GAs) are the most popular metaheuristic used in SBSE, which are inspired by the natural selection process and used to optimize one or more objectives. GAs start with a randomly generated population of solutions, where each individual is a potential solution for the optimization problem. The quality of each solution is assessed based on its fitness value calculated using a *fitness* function. GAs help the population evolve towards better solutions by generating new solutions using genetic operators (i.e., *selection, crossover*, and *mutation*) [63] in each generation. The *mutation operator* randomly modifies parts of individual solutions; the *crossover* *operator* recombines pairs of selected individual solutions; and the *selection operator* selects candidate solutions for the population.

Non-dominated Sorting Genetic Algorithm (NSGA-II) [64, 65] relies on the Pareto dominance theory, which yields a set of non-dominated solutions for multiple objectives [64]. NSGA-II sorts candidate solutions (i.e., the population) into various non-dominated fronts using a ranking algorithm. Afterward, the individual solutions are selected from the non-dominated fronts. In case, the number of solutions in the non-dominated front exceeds the population size, the solutions with a higher value of *crowding distance* are selected to increase the diversity of solutions. *Crowding distance* measures the distance between the individual solutions and the rest of the solutions in the population [66].

NSGA-III [67, 68] is a relatively new multi-objective algorithm that has performed better than NSGA-II in some contexts [69]. The basic working procedure of NSGA-III is quite similar to the NSGA-II but with significant changes in its selection operator. As oppose to NSGA-II, NSGA-III's selection process exploits well-spread reference points to apply the selection pressure to maintain diversity among population members.

Multi-objective Cellular Genetic Algorithm (MoCell) is based on the cellular model of GAs, which assumes that an individual only interacts with its neighbors in the population during the search process [70, 71]. MoCell stores the obtained non-dominated individual solutions in an external archive. At the end of each generation, a fixed number of randomly selected solutions are replaced by selecting the same number of solutions from the archive with a feedback procedure until the termination conditions are met. Note, this replacement only occurs when newly generated solutions are worse than the solutions in the archive.

Indicator-based Evolutionary Algorithm (IBEA) incorporates an arbitrary performance indicator (e.g., Hypervolume (HV), Epsilon) into the selection mechanism of a multi-objective evolutionary algorithm [72]. IBEA uses the quality indicators to guide the search towards optimal solutions by calculating the fitness of an individual solution as the sum of the indicator values obtained from pairwise comparisons to all other solutions. As opposed to other multi-objective search algorithms, IBEA does not use any additional diversity preservation mechanism such as fitness sharing.

Improved Strength Pareto Evolutionary Algorithm (SPEA2) calculates the fitness for each solution by adding up its raw fitness and density information [73]. The raw fitness is computed based on the number of solutions it dominates. The density information is calculated based on the distance between an individual solution and its nearest neighbors to maximize diversity. SPEA2 starts with an empty archive and fills it with the non-dominated solution from the population. In the subsequent generations, new populations are created by combining solutions from the non-dominated solutions of the original population and the archive. Moreover, if the number of combined non-dominated solutions is greater than the population size, the solution with the minimum distance to other solutions is selected by using a truncation operator.

Pareto Archived Evolution Strategy (PAES) keeps an archive of non-dominated solutions just like SPEA2. To find optimal solutions, PAES uses the dynamic mutation operator for exploring the search space [74, 75]. In the beginning, solutions are added to the archive randomly, which are then used to generate the offspring solutions. If the newly generated solutions are better than the parent solutions, then the parent solutions are replaced by newly generated solutions. Similarly, if the newly generated solutions are better than the solutions in the archive, old solutions are replaced by the new ones. However, if the newly generated solutions are worse than parent solutions, they are discarded, and new solutions are generated using parent solutions.

Speed-constrained Multi-objective Particle Swarm Optimization (SMPSO) is a metaheuristic inspired by the social foraging behavior of animals such as bird flocking [76, 77]. It selects the best solutions based on crowding distance and stores them in an archive just like SPEA2 and PAES. SMPSO uses a mutation operator to accelerate the convergence and adapts the velocity constriction mechanism to avoid the explosion of swarms [77]. As a comparison baseline, we used Random Search (RS).

2.4 Branch Distance Calculation Heuristic

In SBSE, branch distance is a commonly used heuristic that shows to what extent given data satisfy the predicate (aka condition or clause) of a specific rule/constraint [78-80]. In this thesis, we also used the branch distance heuristic for our search optimization problems, where we intend to calculate the distance between a configurable parameter and a predicate in the rule. To be more specific, we used the branch distance calculation approach presented in [81, 82]. In Table 2, we present the distance calculation formula for various operations corresponding to numerical and enumerated data.

Table 2: Branch distance functions [81] *

Predicate type	Operation	Distance function
Predicates with relational operators	a=b	0
	a!=b	a!=b → 0 **else** nor(|a−b| +1) *k
Predicate with a Boolean condition		True → 0 **else** k
Logical connective of two predicates	$Pr_1 \wedge Pr_2$	$Pr_1 + Pr_2$ (sum of branch distances for both predicates)

* *k* is a positive constant greater than zero, we used *k*=1; *nor* gives a normalized value between zero and one.

2.5 Machine Learning

Machine learning is used for classifying, clustering, and identifying/predicting patterns in data [83]. It has also been used to infer rules [35, 84]. Machine learning techniques can be categorized as supervised learning (i.e., for labeled data) and unsupervised learning (i.e., for unlabeled data). Supervised learning focuses on finding the relations between input data and its outcome. Unsupervised learning identifies hidden patterns inside input data without labeled responses. Furthermore, supervised learning makes use of class information from the training instances, as opposed to unsupervised learning that does not take the class information into account. In this thesis, we used supervised learning, as we intend to mine the rules based on product configurations (i.e., input) labeled with system states (i.e., outcome) indicating the success/failure of the communication among the products. Supervised models can be categorized as regression and classification models. A regression model maps the input data against a real-valued domain, whereas, the classification models map the input data against predefined classes [85].

Since we have pre-defined system states, we used classification models. In the classification models, there are two main methods of rule generation: 1) indirect method that converts decision trees into rules and prunes them further to get the final set of rules, which is opted by C4.5 [86]); 2) direct method that employs separate-and-conquer rule learning technique to extract rules directly from the data, which is used by Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [87].

Creating rules from decision trees using the indirect method is computationally expensive in the presence of noisy data, whereas, the direct method has hasty generalization (i.e., over-pruning) problem [88]. The Pruning Rule-Based Classification algorithm (PART) [89] avoids these shortcomings by combining the two methods of rule generation mentioned above. PART generates partial decision trees, and corresponding to each partial tree, a single rule is extracted for the branch that covers maximum nodes [88]. In this thesis, we opted for PART due to its unique characteristics in addition to C4.5, which is the most popular algorithm in the research community as well as the industry [90].

As oppose to classification, clustering is used when there is no outcome to be predicted instead input data points are to be combined into natural groups (aka clusters). The main idea behind clustering is that the data points within a cluster should be similar but different across the clusters. In this thesis, we used Lloyd's algorithm [91] for clustering rules, a commonly used *k*-means algorithm. *K*-means algorithm minimizes the average squared distance among the data points within the same cluster. In the beginning, it picks *k* data points randomly as centers of *k* clusters. It uses the Euclidean distance function [92] to compute the distances between each data point and centers of *k* clusters, and assign each data point to its nearest cluster. When all the data points are assigned to *k* clusters, the centers of *k* clusters are updated with the average of all the data points within each cluster. Once centers are updated, it recalculates the Euclidean distance for all the data points and reassigns them to *k* clusters. This process continues until the centers of *k* cluster do not change in two consecutive iterations.

3 Research Methods

In this section, we discuss the research methods used for this thesis. The research work was conducted in the context of a basic research project, Zen-Configurator, funded by the Norwegian Research Council. The employed research method includes three main steps: problem identification and formulation (Section 3.1), solution realization (Section 3.2), and solution evaluation (Section 3.3).

3.1 Problem Identification and Formulation

The thesis started with understanding the requirements for applying PLE on CPSs with interacting products with the help of real-world case studies. More specifically, we analyzed three representatives CPS case studies: Material Handling Systems (MHS), Video Conferencing Systems (VCS), and Subsea Production Systems (SPS). Based on the knowledge collected from the analysis of CPS case studies and a thorough literature review on PLE methodologies, we identified the following challenges related to CPS PLE that we addressed in this thesis.

Challenge- 1. *Variation Point and Constraint Types are not Well Defined for CPS PLs*: One of the key activities of PLE is to capture various types of variabilities of PLs using well-defined variation point types to enable reuse of assets (e.g., requirements, code). Unlike traditional software PLE, CPS PLE involves capturing variabilities for 1) multiple domains (e.g., mechanics, electronics, software), 2) physical and component properties of CPSs, 3) complex interactions among different components and subsystems, 4) topologies, and 5) software deployment on hardware. Moreover, CPS PLs also require identifying the binding time (e.g., design time) for captured variabilities. This demands identifying and defining different types of CPS specific variation point types systemically. On the other hand, we also need to capture different types of constraints, which play a crucial role in enabling various types of automation of configuration for CPS PLs. For example, enabling automated consistency checking [93] and configuration recommendation [94] for CPS PLE requires capturing consistency constraints [6] and configuration constraints [34].

Challenge- 2. *Configuration Process for CPS PLs is not Well Formulated*: The configuration process employed in CPS PLE is more complex than traditional software PLE. In CPS PLE, various components (e.g., software, hardware, network) are configured by different domain experts from different department/organizations during different phases of the product development lifecycle. This requires defining a configuration process for CPS PLs, which allows users to perform various configuration tasks sequentially or concurrently in an incremental multi-stage and multi-step manner [95]. Furthermore, the configuration process has a great impact on the implementation of various types of configuration automation. Thus, it becomes crucial to formalize the configuration process for CPS PLs to enable different types of automation of configuration,

Challenge- 3. *Lacking Formal Definitions of Various Types of Automation of Configuration for CPS PLE*: Often the effectiveness of cost-effective PLE is associated with its support for automation (e.g., consistency checking, collaborative configuration) because manual product derivation and debugging is time-consuming and error-prone [45]. Existing configuration tools support some automated functionalities for traditional software PLE, however, they are not well-suited for CPS PLE because of different configuration process employed and more complex types of variabilities for multiple domains. For example, in the case of CPS PLE, we need to propagate configuration decisions across multiple stages and steps which is not the case in traditional software PLE. This requires identifying and providing precise definitions of various automated functionalities that can be implemented in a configuration tool for CPS PLE.

Challenge- 4. *Existing PLE Methodologies are not Evaluated for CPS PLs*: A large number of PLE methodologies (i.e., both VMTs [10-19] and configuration tools [20-26]) exists in the literature to support PLE in different contexts. However, to what extent these methodologies can support the PLE of CPSs is not evaluated.

Challenge- 5. *Difficult to Predict Runtime Behaviors of CPSs for Different Configurations*: The runtime behaviors of CPSs with interacting products are determined by configurations of products and information networks. Moreover, there exist many faulty configurations that can lead to unwanted states of CPSs. Thus, we need to identify the patterns of configurations in form of configuration rules to facilitate the post-deployment configuration in various contexts (e.g., for testers or end-users).

Challenge- 6. *Difficult to Specify Configuration Rules due to High Reliance on Domain Knowledge and Certain Information Being Available only at Runtime*: Manually specifying the configuration rules based on domain knowledge is tedious and time-consuming, and heavily relies on experts' knowledge of the domain [34]. Furthermore, certain information (e.g., network related information such as bandwidth, traffic congestion, and maximum transmission unit size) is only known at runtime [34], which makes it impossible to specify these rules manually. This requires a sophisticated approach to automatically infer the configuration rules.

Challenge- 7. *A Large Number of Possible Configurations to Test:* CPSs are highly configurable systems and testing these systems with all possible configurations is not feasible due to limited time and resources [37]. This requires a sophisticated method to reduce the number of configurations to be tested.

Challenge- 8. *Difficult to Select the Most Critical Configurations due to High Reliance on Domain Expertise:* CPSs have a large number of configurations and not all configurations are equally important for ensuring the high quality of the produced systems within the time budget, which requires finding the most critical configurations for testing the CPS. To do so, often testers have to merely rely on their experience and domain knowledge [38, 96]. This motivates for an approach that can help the testers to make informed rational decisions for selecting configurations for testing CPSs.

Challenge- 9. *Unpleasant Post-Deployment Configuration Experience due to Lack of Guidance*: Usually, end-users configure the products with different configurations at the post-deployment time using a configuration tool or a user manual [97, 98]. Many configurations can lead to the unwanted behavior of the system (e.g., failed communication among the products constituting the CPS), which causes an unpleasant user experience. This requires guiding the users to avoid faulty configurations to improve the configuration experience.

3.2 Solution Realization

This step focuses on realizing the solutions to address each of the nine challenges described in Section 3.1. Table 3 gives an overview of how different challenges were addressed in this Ph.D. thesis.

Table 3. An Overview of solutions for addressing different challenges

No.	Challenge	Solution	Papers
1	Variation Point and Constraint Types are not Well Defined for CPS PLs	Proposed a conceptual framework to support PLE for CPSs.	Paper-A, Paper-B
2	Configuration Process for CPS PLs is not Well Formulated		
3	Lacking Formal Definitions of Various Types of Automation of Configuration for CPS PLE		
4	Existing PLE Methodologies are not Evaluated for CPS PLs	Evaluated existing VMTs and configuration tools	
5	Difficult to Predict Runtime Behaviors of CPSs for Different Configurations	Proposed a rule mining approach SBRM and its improved version SBRM$^+$	Paper-C, Paper-D
6	Difficult to Specify Configuration Rules due to High Reliance on Domain Knowledge and Certain Information Being Available only at Runtime		
7	A Large Number of Possible Configurations to Test	Proposed a configuration recommendation approach SBCR	Paper-E
8	Difficult to Select the Most Critical Configurations due to High Reliance on Domain Expertise		
9	Unpleasant Post-Deployment Configuration Experience due to Lack of Guidance		

As shown in Table 3, we proposed a conceptual framework to address *Challenge-1-3*. More specifically, we proposed classifications of variation point and constraint types, formalized a multi-stage multi-step configuration process, and presented a list of 14 functionalities for a configuration tool along with their formal definitions. For addressing *Challenge-4*, we evaluated existing VMTs and configuration tools in terms of their support for CPS PLE. To address *Challenge-5-6*, we devised a Search-Based Rule Mining (*SBRM*) approach and its improved version called *SBMR*+, which make use of machine learning and multi-objective search to capture the patterns of configurations in form of configuration rules. Similarly, to address *Challenge-7-9*, we developed a Search-Based Configuration Recommendation (*SBCR*) approach, which uses multi-objective search to recommend configurations based on previously mined configuration rules.

3.3 Solution Evaluation

A fundamental part of the thesis was to evaluate the proposed methodologies in terms of each research problem. To do so, we performed different case studies (e.g., real-world, and open-source) and conducted experiments in addition to extensive literature reviews. We also conducted different types of analyses (e.g., difference analysis, correlation analysis, and trend analysis [99]) and applied rigorous statistical tests such as Vargha and Mann-Whitney U test [100], Delaney statistics [101], and Spearman's test [102] to analyze the results. For instance, the completeness of the proposed conceptual framework was evaluated by performing different case studies and conducting an extensive literature review (Paper-B). Another example is the evaluation of *SBRM*+ where we conducted experiments using different case studies and conducted three types of analyses (Paper-D). The goal of such extensive evaluation is to ensure that the proposed methodologies are useful and robust, and they improve the current state of the art.

4 Research Contributions

In this section, we report our research contributions made to address different challenges presented in Section 3.1.

4.1 A Conceptual Framework for CPS PLE (Paper-A and Paper-B)

Based on the knowledge collected from the analysis of CPS case studies, a thorough literature review on PLE methodologies, and our experience of conducting research in the field of CPS PLE [103], we proposed classifications for basic and CPS-specific VP types to capture the variabilities of CPS PLs and several modeling requirements for VMTs (Paper-A).

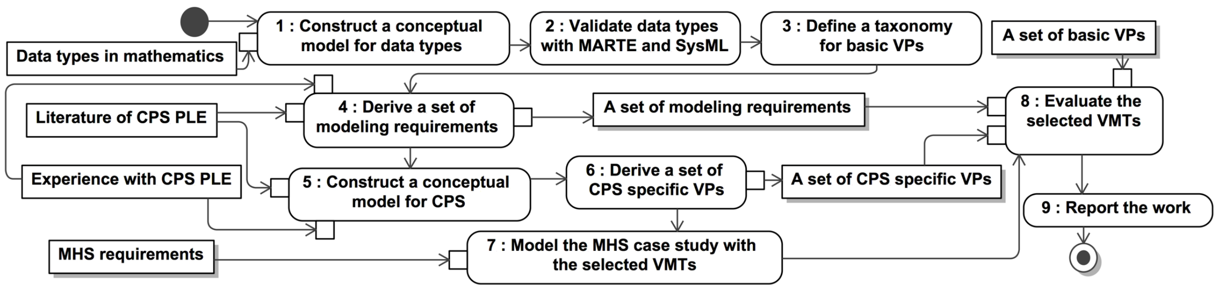

1 : Construct a conceptual model for data types
2 : Validate data types with MARTE and SysML
3 : Define a taxonomy for basic VPs
A set of basic VPs
Data types in mathematics
Literature of CPS PLE
4 : Derive a set of modeling requirements
A set of modeling requirements
8 : Evaluate the selected VMTs
Experience with CPS PLE
5 : Construct a conceptual model for CPS
6 : Derive a set of CPS specific VPs
A set of CPS specific VPs
9 : Report the work
MHS requirements
7 : Model the MHS case study with the selected VMTs

Figure 4: Deriving basic and CPS-specific VP type classification.

As shown in Figure 4, first, we constructed a conceptual model for data types in mathematics and validated the data types with MARTE [104] and SysML [105] to check their completeness. Afterward, corresponding to each basic data type, we defined a basic VP type as configuring a VP always requires assigning a value to a basic type variable. Furthermore, we systematically derived a set of CPS-specific VP types based on a conceptual model of CPS. Finally, we evaluated four representative VMTs (i.e., FM, CBFM, CVL, and SimPL) by modeling the MHS case study.

Results show that none of the selected VMTs can capture all the basic and CPS-specific VPs and meet all the modeling requirements.

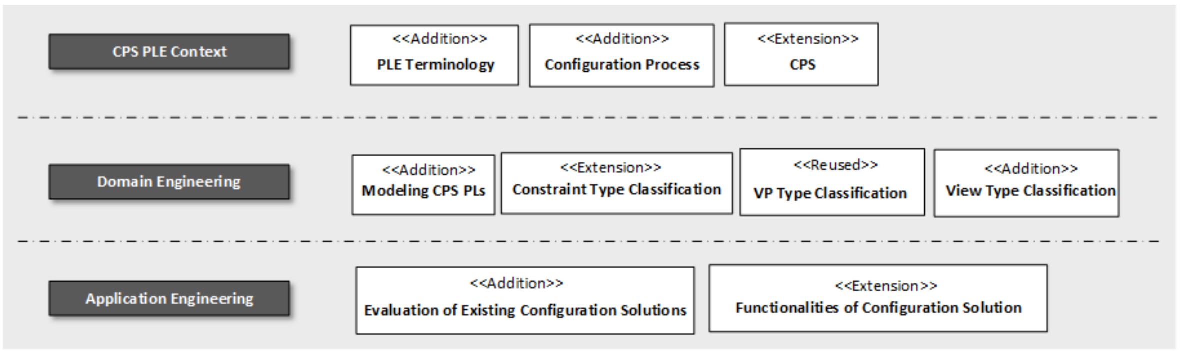

CPS PLE Context
<<Addition>> PLE Terminology
<<Addition>> Configuration Process
<<Extension>> CPS
Domain Engineering
<<Addition>> Modeling CPS PLs
<<Extension>> Constraint Type Classification
<<Reused>> VP Type Classification
<<Addition>> View Type Classification
Application Engineering
<<Addition>> Evaluation of Existing Configuration Solutions
<<Extension>> Functionalities of Configuration Solution

Figure 5: An overview of the conceptual framework for CPS PLE

Moreover, we extended our work presented in Paper-A and proposed a complete conceptual framework for supporting both domain engineering and application engineering of CPS PLs (Paper-B). As shown in Figure 5, the conceptual framework has three parts as follows:

- To clarify the context of CPS PLE, we formalized PLE based on the PLE ISO/IEC standard for Product Line Engineering and Management [27], CPSs, and multi-stage and multi-step configuration process using three conceptual models and a set of OCL constraints.
- To support the domain engineering of CPS PLs, we formalized concepts related to modeling of CPS PLs such as models, model elements. We also presented the classifications of VP, constraint, and view types. The VP types are from Paper-A, whereas, for the constraint classification, we extended our previously proposed constraint classification [6] by adding four new types. We also provided formal definitions of different types of constraints based on the set theory notation.
- To support the application engineering of CPS PLs, we present 14 possible functionalities of an automated configuration tool and provide their formal definitions. Five of the 14 functionalities were presented in [6].

We evaluated the completeness of the framework using three real-world case studies of CPS PLs (i.e., VCS, MHS, and SPS) and an extensive literature review. We evaluated the VP types, constraint types, and view types using the three case studies. We validated the functionalities and configuration process using 11 configuration tools and existing literature on the automation of configuration. Evaluation results based on the case studies suggest that the framework fulfills all the requirements of the case studies in terms of capturing and managing variabilities and constraints. The results of the literature review show that the framework has all the functionalities concerned by the literature, indicating the completeness of the framework in terms of enabling maximum automation of configuration for CPS PLs.

4.2 Search-Based Rule Mining (Paper-C, Paper-D)

CPL rules describe how configurations of communicating products within/across PLs impact their runtime interactions via information networks. Such rules are of great importance because they can be used to identify invalid configurations, where products may fail to interact and provide support for enabling automated/semi-automated configuration of future products.

Manually specifying such rules is tedious, time-consuming, and requires expert knowledge of the domain and the PLs. To address this challenge, we propose an approach called Search-based Rule Mining (*SBRM*) that combines multi-objective search with machine learning to mine rules in an incremental and iterative fashion (Paper-C). Figure 6 gives an overview of the proposed approach, where the whole process has seven steps organized into four types of activities *Generation*, *Execution*, *Mining*, and *Classification*.

As a first step, we generate a set of initial configurations randomly for configurable parameters of interacting products for which we obtain system states indicating the success or failure of interaction among the products (step 2). In step 3, we apply a rule mining algorithm (e.g., PART in *SBRM*) to mine a set of rules using generated configurations (as attributes) and their corresponding system states (as classes). In step 4, we classify the rules into three categories based on which we define three search objectives to guide the search for generating configurations for the next iteration (step 5).

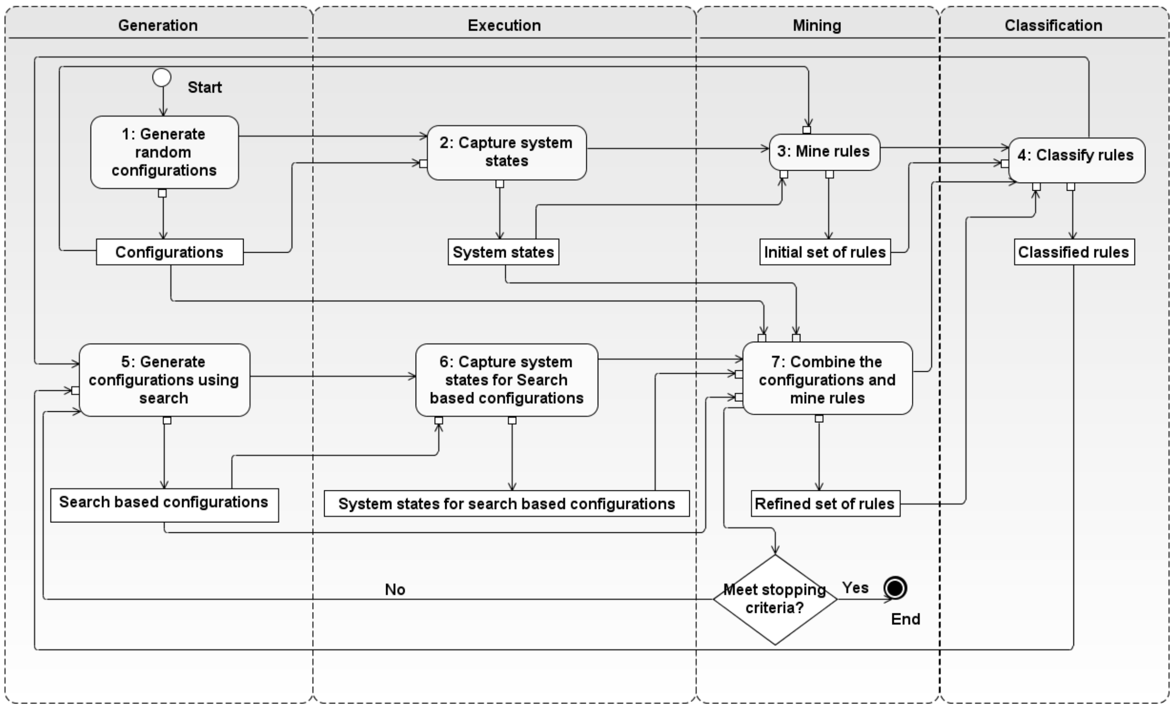

Generation
Execution
Mining
Classification
Start
1: Generate random configurations
2: Capture system states
3: Mine rules
4: Classify rules
Configurations
System states
Initial set of rules
Classified rules
5: Generate configurations using search
6: Capture system states for Search based configurations
7: Combine the configurations and mine rules
Search based configurations
System states for search based configurations
Refined set of rules
Meet stopping criteria?
No
Yes
End

Figure 6: An overview of SBRM and SBRM+

Generally, system states can be categorized as normal states and abnormal states indicating the success and failure of interaction among the products, respectively. Thus, CPL rules can also be classified as normal state rules and abnormal state rules (Category-III). Each rule has a confidence value between 0 and 1 that can be calculated as: $Cf(r_i) = \frac{SP_i - V_i}{SP_i + V_i}$, where SP_i and V_i show the support and violation of the rule. Support (violation) represents the number of data points (configurations in our context) for which the rule holds true (false). Based on confidence, support, and violation, we classify the normal state rules as high confidence (Category-I) and low confidence (Category-II) rules. Based on three categories of rules, we defined three objectives for *SBRM*: 1) avoid configuration data satisfying or close to satisfying high confidence rules with normal states), 2) generate configuration data satisfying or close to satisfying low confidence rules with normal states, and 3) generate configuration data satisfying or close to satisfying rules with abnormal states. The defined three objectives are integrated with NSGA-II for generating configurations using search and PART algorithm to mine the rules.

In step 6, we capture the system states for configurations generated with search from step 5. In step 7, we combine all the configurations generated from steps 1 and 5 along with their corresponding system states from steps 2 and 6 to mine the refined set of rules. The refined rule set is used in the next iteration to generate new configurations, which are added into the dataset from the previous iteration to mine a new set of rules. We repeat the process (step 4 to step 7) until we meet the stopping criteria, e.g., a fixed number of iterations and/or when the rules mined from two consecutive iterations are similar. To evaluate the *SBRM*, we performed a real case study of two VCS products with 17 configurable parameters, belonging to two different PLs. Results show that *SBRM* performed significantly better than Random Search-Based Rule Mining (*RBRM*) in terms of fitness values, HV, and machine learning quality measurements (MLQMs). When comparing with rules mined with real data, *SBRM* performed significantly better in terms of Precision (18%), Recall (72%), and F-measure (59%) corresponding.

In *SBRM*, we classify the rules as high confidence and low confidence rules based on a threshold for confidence and a threshold for the sum of support and violation. In *SBRM*$^+$, we improved this classification by applying k-means clustering algorithm instead of using thresholds, which is more robust (Paper-D). We also integrated one more search algorithm (i.e., NSGA-III) and a rule mining algorithm (i.e., C4.5). We also conducted a thorough empirical study to evaluate the performance of *SBRM+* using a real-world case study (i.e., Cisco) with three products belonging to three different PLs and an open-source case study (i.e., Jitsi) with three products belonging to the same PL. We have 27 and 39 configurable parameters for the Cisco and Jitsi case studies, respectively. Note that for Cisco case study, the experiments were conducted using real systems but not in the industrial setting. With the two case studies, we conducted three types of analyses difference analysis, correlation analysis, and trend analysis. Results show that *SBRM*$^+$ significantly outperformed *RBRM*$^+$ in terms of fitness values, six quality indicators, and 17 MLQMs. As compared to *RBRM*$^+$, *SBRM*$^+$ has improved the quality of rules based on MLQMs up to 27% for the Cisco case study and 28% for the Jitsi case study.

4.3 Search-Based Configuration Recommendation (Paper-E)

Testing CPSs consisting of interacting products is particularly challenging due to a large number of possible configurations and limited available resources. This requires testing these systems with specific configurations, where the products will most likely fail to communicate with each other. To cater this, we proposed a Search-Based Configuration Recommendation (*SBCR*) approach to recommend faulty configurations for SUT based on CPL rules. In *SBCR*, we defined four search objectives based on CPL rules (Figure 7): 1) maximizing the violation of normal state rules, 2) maximizing the conformance of abnormal state rules, 3) maximizing the confidence of recommended configuration by maximizing the confidence of violated (satisfied) normal (abnormal) state rules, and 4) maximizing the dissimilarity between configurations being recommended to already recommended configurations. The defined objectives are combined with six commonly used search algorithms (Figure 7).

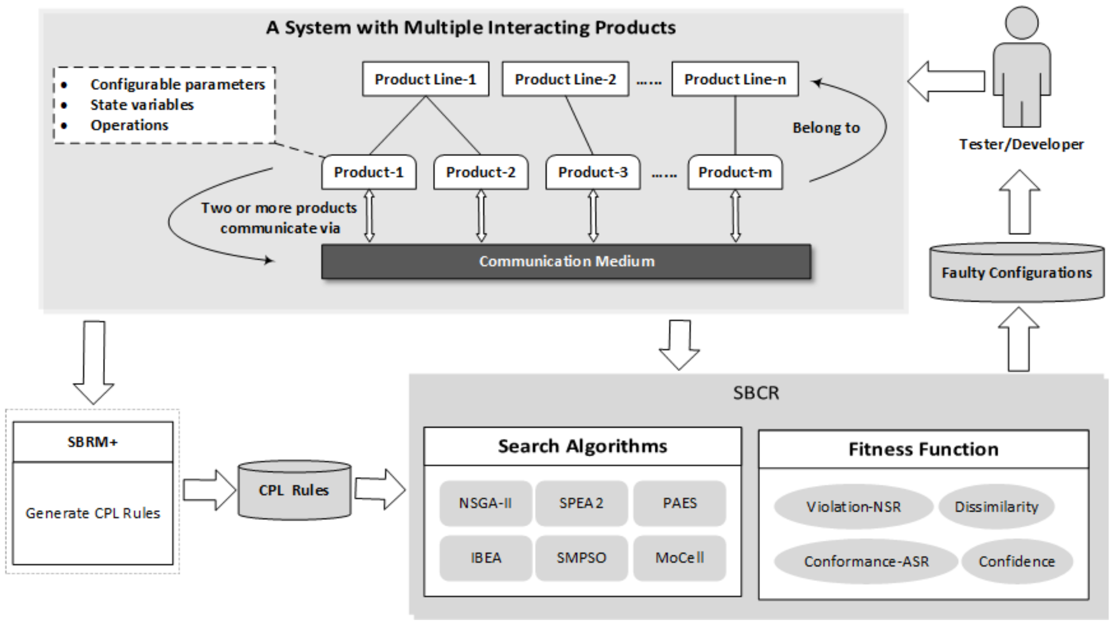


Figure 7: The overall context and scope of SBCR

To evaluate the six variants of *SBCR* (i.e., *SBCR*$_{NSGA\text{-}II}$, *SBCR*$_{IBEA}$, *SBCR*$_{MoCell}$, *SBCR*$_{SPEA2}$, *SBCR*$_{PAES}$, and *SBCR*$_{SMPSO}$), we performed two case studies (Cisco and Jitsi) and conducted difference analysis. Since we need CPL rules for applying *SBCR*, we used the same case studies as we did for *SBRM*$^+$ in Paper-D. Results show that *SBCR* performed significantly better than Random Search-Based Configuration Recommendation in terms of six quality indicators and the percentage of faulty configurations for both case studies. Among the six variants of *SBCR*, *SBCR*$_{SPEA2}$ performed the best for recommending faulty configurations for SUT.

5 Summary of Results

In this section, we present a summary and key results of each paper submitted as part of this thesis.

5.1 Paper-A: Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering

Authors: Safdar Aqeel Safdar, Tao Yue, Shaukat Ali, and Hong Lu.
Venue: Published in the Proceeding of International Conference on System Analysis and Modeling (SAM)
Publisher: Springer
Year: 2016.

In this paper, we aim to facilitate domain engineering of CPS PLs. More specifically, we analyzed the key requirements of CPS PLs in terms of capturing variabilities and constraints. In this context, we proposed a set of basic and CPS-specific variation point (VP) types and modeling requirements for CPS-specific VMTs. Furthermore, based on the proposed VP types (basic and CPS-specific) and modeling requirements, we evaluated four existing VMTs FM, CBFM, CVL, and SimPL using a real-world case study (i.e., MHS) from the logistics domain.

The following research questions are addressed in this paper.

RQ1. To what extent can each selected VMT capture the basic VPs?

The results of RQ1 show that SimPL and CVL can capture all the basic VP types, however, FM and CBFM provide only partial support. FM and CBFM support 3/8 and 7/8 basic VP types, respectively.

RQ2. To what extent can each selected VMT capture the CPS-specific VPs?

The results of RQ2 show that none of the selected VMTs can capture all the CPS-specific VP types. SimPL and CVL support 81% and 75% of the total CPS-specific VP types, respectively. Similarly, FM and CBFM support only 15% and 50% of the total CPS-specific VP types.

RQ3. To what extent does a selected VMT comply with the modeling requirements?

The results of RQ3 show that SimPL fulfills all modeling requirements except one (i.e., binding times for a variation point). FM and CBFM only satisfy one modeling requirement, whereas, CVL fully or partially meets four out of nine modeling requirements.

5.2 Paper-B: A Framework for Automated Multi-Stage and Multi-Step Product Configuration of Cyber-Physical Systems

Authors: Safdar Aqeel Safdar, Hong Lu, Tao Yue, Shaukat Ali, and Kunming Nie.
Venue: Published in the Journal of Software and Systems Modeling (SoSym)
Publisher: Springer
Year: 2020.

This paper is a journal extension of Paper-A that addresses the problem of supporting multi-stage and multi-step automated configuration of CPS PLs. In this paper, we proposed a conceptual framework based on the results of our previous works [6, 106], our experience of working with CPS PLs [103], and a thorough literature review. The framework has three parts as follows:

- *ContextFormalization:* We formalized PLE based on the PLE ISO/IEC standard for Product Line Engineering and Management [27], CPSs, and multi-stage and multi-step configuration process using UML based conceptual models and OCL constraints.
- *DomainEngineering:* To support domain engineering of CPS PLs, we 1) presented the classifications of VP types (i.e., borrow from Paper-A [106]) and constraint types (i.e., extended from [6]); 2) formalized concepts related to modeling of CPS PLs (e.g., models, model-elements, and views) and constraint types using UML models and OCL constraints; and 3) provided formal definitions of constraint types.
- *ApplicationEngineering:* To support application engineering of CPS PLs, we presented 14 possible functionalities of an automated configuration tool and provided their formal definitions.

The framework is evaluated by performing three real-world case studies of video conferencing systems (VCS), material handling systems (MHS), and subsea production systems (SPS) and a thorough literature review. With the case studies, we evaluated the VP types, constraint types, and views, whereas, the functionalities and configuration process are validated using existing configuration tools techniques in the literature.

The following research questions are addressed in this paper.

RQ1. To what extent the framework can capture the variabilities of CPS PLs based on the selected case studies?

The results of RQ1 suggest that the selected three case studies MHS, VCS, and SPS have 476, 1507, and 178 VPs respectively and all of these VPs can be captured using the CPS-specific VP types provided by the framework. Overall, the three case studies have 2161 VPs in total. The MHS case study requires all the CPS-specific VP types to capture its VPs, whereas, the other two case studies (i.e., SPS and VCS) require only 12 out of 16 CPS-specific VP types.

RQ2. To what extent the framework can capture the constraints for CPS PLs based on the selected case studies?

The results of RQ2 show that case studies MHS, VCS, and SPS have 763, 2897, and 283 constraints respectively and all of them can be captured with the constraint types provided by the framework. Overall, three case studies have 3943 constraints that can be captured using 6 out of 7 constraint types provided by the framework.

RQ3. To what extent the framework is complete for providing different views for CPS PLs based on the selected case studies?

The results of RQ3 show that the MHS case study requires 14 views, whereas, VCS and SPS both need 13 views. For modeling all the views, MHS, VCS, and SPS require 82%, 76%, and 76% of view types respectively. Overall, three case studies require 40 views and all of them are supported by the view types provided by the framework.

RQ4. To what extent the framework is complete for providing support for automation of configuration based on existing literature?

The results of RQ4 show that all the functionalities except *RedundancyDetection* are supported by one or more configuration tools (i.e., 92%), which shows that the identified functionalities are quite consistent with the literature and existing configuration tools. We also observed that none of the existing tools supports all the functionalities. Furthermore, some functionalities such as *ConsistencyChecking* and *DecisionInference* are widely considered important, and thus, they have been mostly implemented. However, the least reported functionalities such as *ConflictDetection* and *RedundancyDetection* are also vital to ensure the correctness of product configuration.

5.3 Paper-C: Mining Cross Product Line Rules with Multi-Objective Search and Machine Learning

Authors: Safdar Aqeel Safdar, Hong Lu, Tao Yue, and Shaukat Ali.

Venue: Published in the Proceeding of the Genetic and Evolutionary Computation Conference (GECCO)

Publisher: ACM

Year: 2017.

This paper focuses on mining the configuration rules (named as CPL rules) for products within/across PLs communicating with each other via information networks. To do so, we proposed a Search-based Rule Mining (*SBRM*) approach that combines multi-objective search with machine-learning techniques for mining CPL rules in an incremental and iterative manner. *SBRM* obtains CPL rules with different degrees of confidence while emphasizing on mining rules that can disclose invalid configurations. In *SBRM*, we defined three search objectives to guide the search and incorporated the most commonly used NSGA-II for generating configurations and PART algorithm to mine the rules.

We evaluated the *SBRM* using a real-world case study of two VCS products belonging to different PLs, communicating (i.e., call) with each other. The performance of *SBRM* is compared with *RBRM* in terms of fitness values, HV, and seven Machine Learning Quality Measurements (MLQMs). Moreover, we also compared the rules generated using *SBRM* with the rules mined based on real data (named as *RDBRM*) extracted from test case execution logs.

The following research questions are addressed in this paper.

RQ1. Is NSGA-II effective to solve the configuration generation problem as compared to RS?
The results of RQ1 suggest that NSGA-II significantly performed better than RS in terms of fitness values of three objectives as well as HV. This suggests that NSGA-II is more effective than RS for solving the configuration generation problem.

RQ2. Does *SBRM* produce better quality rules than *RBRM* in terms of machine learning measurements?

The results of RQ2 show that in the first iteration, *SBRM* performed better than *RBRM* in terms of MLQMs, but not significantly. However, as you move from the first iteration to the third iteration, *SBRM* significantly outperformed *RBRM*. We observed an increasing trend of improvement in terms of MLQMs against the number of iterations. Overall, *SBRM* also significantly outperformed *RBRM* in terms of all the MLQMs.

RQ3. Does *SBRM* produce better quality rules than *RDBRM* in terms of machine learning measurements?

The results of RQ3 show that *SBRM* performed significantly better than *RDBRM* in five out of the seven MLQMs, whereas, *RDBRM* outperformed *SBRM* in terms of only one MLQM (i.e., *Connected Recall*). Thus, we can conclude that *SBRM* produces better quality rules than *RDBRM*. In comparison to *RDBRM*, *SBRM* achieved 18%, 72%, and 59% higher scores for *Failed Precision*, *Failed Recall*, and *Failed F-measure*, respectively.

5.4 Paper-D: Using multi-objective search and machine learning to infer rules constraining product configurations

Authors: Safdar Aqeel Safdar, Tao Yue, Shaukat Ali, and Hong Lu.
Venue: Published in the Journal of *Automated Software Engineering (ASE)*
Publisher: Springer
Year: 2019.

This paper is a journal extension of Paper-C with several additional contributions as follows:

- A significantly improved version of *SBRM* (named as *SBRM*$^+$) is proposed.
    - *K*-means clustering algorithm is used in *SBRM*$^+$ unlike using thresholds in *SBRM* to classify rules as high and low confidence rules, which are used for defining search objectives.
    - NSGA-II and NSGA-III are incorporated into *SBRM*$^+$, whereas, in *SBRM*, we used only NSGA-II.

    - PART and C4.5 are incorporated into *SBRM*$^+$ (referred to as *SBRM*$^+_{NSGA\text{-}II}$*-C45*, *SBRM*$^+_{NSGA\text{-}III}$*-C45*, *SBRM*$^+_{NSGA\text{-}II}$*-PART*, and *SBRM*$^+_{NSGA\text{-}III}$*-PART*), whereas, in *SBRM*, we used only PART.
- The *SBRM*$^+$ approaches are evaluated using a real-world case study of three communicating VCS products belonging to three different PLs (Cisco) with 27 configurable parameters and a real-world open-source case study of three products of Audio/Video Internet Phone and Instant Messenger, belonging to the same PL (Jitsi) with 39 configurable parameters. The *SBRM* was evaluated using a case study of two communicating products with 17 configurable parameters.
- Three types of analyses *difference analysis*, *correlation analysis*, and *trend analysis* are conducted for both case studies.
    - *Difference analysis:* The performance of NSGA-II and NSGA-III integrated with PART and C4.5 is compared with RS integrated with PART and C4.5 in terms of fitness values, six quality indicators (i.e., HV, Inverted Generational Distance (IGD), Epsilon, Euclidean Distance from the Ideal Solution (ED), Generational Distance (GD), and Generalized Spread (GS)), and 17 MLQMs. Additionally, the performance of four *SBRM*$^+$ approaches is also compared to find the best performing approach. In Paper-C, we compared the performance of NSGA-II combined with PART with RS combined with PART using fitness values, HV, and seven MLQMs only.
    - *Correlation analysis:* We studied the correlation of the MLQMs with average fitness values and quality indicators, which was not done in Paper-C.
    - *Trend analysis:* The trend in the quality of rules based on MLQMs across different iterations of *SBRM*$^+$ is studied, which was not done in Paper-C.

The following research questions are tackled in this paper.

RQ1. Are NSGA-II and NSGA-III effective to generate configurations for mining rules as compared to RS?

The results of RQ1 show that *SBRM*$^+$ significantly outperformed *RBRM*$^+$ in terms of fitness values for both case studies. Similarly, *SBRM*$^+$ also performed significantly better than *RBRM*$^+$ in terms of all quality indicators except *GS* in 221/240 comparisons for both case studies, whereas, in terms of *GS*, *RBRM*$^+$ significantly outperformed *SBRM*$^+$ in 32/48 comparisons. Thus, based on the results of RQ1, we can conclude that NSGA-II and NSGA-III are more effective than RS.

RQ2. Does *SBRM*$^+$ produce better quality rules in terms of MLQMs than *RBRM*$^+$?

The results of RQ2 show that for the Cisco case study, *SBRM*$^+_{\text{NSGA-II}}$*-C45* (*SBRM*$^+_{\text{NSGA-III}}$*-C45)* significantly outperformed *RBRM*$^+$*-C45* in 87% (60%) of the total comparisons. Similarly, *SBRM*$^+_{\text{NSGA-II}}$*-PART* (*SBRM*$^+_{\text{NSGA-III}}$*-PART*) significantly outperformed *RBRM*$^+$*-PART* in 75% (61%) of the total comparisons. For the Jitsi case study, *SBRM*$^+_{\text{NSGA-II}}$*-C45* (*SBRM*$^+_{\text{NSGA-III}}$*-C45*) significantly outperformed *RBRM*$^+$*-C45* in 84% (19%) of the total comparisons. Likewise, *SBRM*$^+_{\text{NSGA-II}}$*-PART* (*SBRM*$^+_{\text{NSGA-III}}$*-PART*) significantly outperformed *RBRM*$^+$*-PART* in 86% (47%) of the total comparisons. Overall, *SBRM*$^+$ approaches significantly outperformed the *RBRM*$^+$ approach in terms of the majority of MLQMs for both case studies except *SBRM*$^+_{\text{NSGA-III}}$*-C45* for the Jitsi case study. While comparing *SBRM*$^+_{\text{NSGA-III}}$*-C45* with *RBRM*$^+$*-C45* for the Jitsi case study, neither one of the two approaches dominated the other. Thus, it can be concluded that given the same context *SBRM*$^+$ produces higher quality rules than *RBRM*$^+$. In the worst case, *SBRM*$^+$ produces rules with the same quality as for *RBRM*$^+$.

RQ3. To what extent the quality of rules improved using *SBRM*$^+$ in comparison to *RBRM*$^+$ (after the final iteration)?

The results of RQ3 show that for both case studies, *SBRM*$^+$ has significantly improved the quality of rules in terms of MLQMs as compared to *RBRM*$^+$. For the Cisco case study, we observed that *SBRM*$^+$ has positive improvements for 85% of the MLQMs with up to 27% of an average relative improvement (ARI) score. Similarly, for the Jitsi case study, *SBRM*$^+$ has positive improvements for 90% of the MLQMs with an ARI of up to 28%. Note that *SBRM*$^+$ has negative ARIs scores for MQLs only when *SBRM*$^+$ did not produce rules related to a specific system state due to fewer configurations with the same system state.

RQ4. Which one of NSGA-II and NSGA-III is more effective to generate configurations for mining rules?

The results of RQ4 show that for both case studies, *SBRM*$^+_{\textit{NSGA-III}}$*-C45* (*SBRM*$^+_{\textit{NSGA-III}}$*-PART*) significantly outperformed *SBRM*$^+_{\textit{NSGA-II}}$*-C45* (*SBRM*$^+_{\textit{NSGA-II}}$*-PART*) in terms of fitness values. Similarly, in terms of quality indicators, *SBRM*$^+_{\textit{NSGA-III}}$*-C45* (*SBRM*$^+_{\textit{NSGA-III}}$*-PART*) significantly outperformed *SBRM*$^+_{\textit{NSGA-II}}$*-C45* (*SBRM*$^+_{\textit{NSGA-II}}$*-PART*) for the Cisco case study. For the Jitsi case study, *SBRM*$^+_{\textit{NSGA-II}}$*-C45* (*SBRM*$^+_{\textit{NSGA-II}}$*-PART*) significantly outperformed *SBRM*$^+_{\textit{NSGA-III}}$*-C45* (*SBRM*$^+_{\textit{NSGA-III}}$*-PART*) in terms of the quality indicators. To summarize, in most of the cases NSGA-III significantly outperformed NSGA-II in terms of fitness values and quality indicators, however, in some cases (e.g., for *GS*) we observed otherwise.

RQ5. Which one of PART and C4.5, when combined with NSGA-II and NSGA-III, produces better quality rules?

Results of RQ5 show that for both case studies, *SBRM*$^+_{\text{NSGA-II}}$*-C45* and *SBRM*$^+_{\text{NSGA-II}}$*-PART* significantly outperformed *SBRM*$^+_{\text{NSGA-III}}$*-C45* and *SBRM*$^+_{\text{NSGA-III}}$*-PART*, respectively. In the comparison of *SBRM*$^+_{\text{NSGA-II}}$*-C45* and *SBRM*$^+_{\text{NSGA-II}}$*-PART*, *SBRM*$^+_{\text{NSGA-II}}$*-C45* significantly outperformed *SBRM*$^+_{\text{NSGA-II}}$*-PART* for Cisco, whereas, for the Jitsi case study, *SBRM*$^+_{\text{NSGA-II}}$*-PART* significantly performed better than *SBRM*$^+_{\text{NSGA-II}}$*-C45*. Thus, it can be concluded that given the default parameter settings for machine learning and search algorithms, *SBRM*$^+_{\textit{NSGA-II}}$*-C45* and *SBRM*$^+_{\text{NSGA-II}}$*-PART* produce better quality rules for the Cisco and Jitsi case studies, respectively.

RQ6. How is the quality of rules correlated with average fitness values and quality indicators?

Through correlation analysis, we intend to test our hypothesis that the quality of rules based on MLQMs is positively correlated with average fitness values and quality indicators. The results of RQ6 show that for the Cisco case study, 23%, 59%, 49%, and 36% of the total correlations are significant for *SBRM*$^+_{\textit{NSGA-II}}$*-C45*, *SBRM*$^+_{\textit{NSGA-III}}$*-C45*, *SBRM*$^+_{\textit{NSGA-II}}$*-PART*, and *SBRM*$^+_{\textit{NSGA-III}}$*-PART* respectively, where 72%, 37%, 36%, and 78% of significant correlations satisfy our hypothesis. Likewise, for the Jitsi case study, 60%, 45%, 53%, and 30% of the total correlations are significant for *SBRM*$^+_{\textit{NSGA-II}}$*-C45*, *SBRM*$^+_{\textit{NSGA-III}}$*-C45*, *SBRM*$^+_{\textit{NSGA-II}}$*-PART*, and *SBRM*$^+_{\textit{NSGA-III}}$*-PART* respectively, where 89%, 79%, 82%, and 57% of significant correlations satisfy our hypothesis.

RQ7. What is the trend of the quality of rules produced by *SBRM*$^+$ across the iterations?

From the results of RQ7 for both case studies, we noticed an increasing trend of quality of rules in terms of 81% of the MLQMs for all *SBRM*$^+$ approaches across the iterations. In only 3%

of MLQMs, we noticed a slightly decreasing trend for *SBRM*$^+$ approaches. Thus, we can conclude that the quality of rules produced using *SBRM*$^+$ improves across the iterations**.**

RQ8. Is it feasible to apply *SBRM*$^+$ in practice in terms of time required for employing search to generate configurations?

Results of RQ8 shows that approaches with NSGA-III took significantly more than others, as NSGA-III is significantly slower than NSGA-II and RS. Furthermore, the approaches with *C4.5* also took more time than approaches with the PART algorithm because C4.5 produced lengthier rules than PART. Hence, approaches producing lengthier rules have a higher cost of calculating fitness values and consequently higher execution time. The best performing approach *SBRM*$^+_{NSGA\text{-}II}$*-C45* (*SBRM*$^+_{NSGA\text{-}II}$*-PART*) took 108 (52) minutes to mine CPL rules for the Cisco (Jitsi) case study, which is acceptable as it is a one-time cost.

5.5 Paper-E: Recommending Faulty Configurations for Interacting Systems Under Test Using Multi-Objective Search

Authors: Safdar Aqeel Safdar, Tao Yue, and Shaukat Ali.

Venue: Submitted to the Journal of Transactions on Software Engineering and Methodology (TOSEM)

Year: 2020.

In this paper, we focus on testing CPSs constituting of interacting products with a large number of possible configurations. To be more specific, we proposed an approach called Search-Based Configuration Recommendation (*SBCR*) to recommend faulty configurations for SUT based on CPL rules. In *SBCR*, we defined four search objectives based on CPL rules and combined them with six commonly used search algorithms.

We evaluated the six versions of *SBCR* (i.e., *SBCR*$_{NSGA\text{-}II}$, *SBCR*$_{IBEA}$, *SBCR*$_{MoCell}$, *SBCR*$_{SPEA2}$, *SBCR*$_{PAES}$, and *SBCR*$_{SMPSO}$) using two case studies (i.e., Cisco and Jitsi used in Paper-D) and conducted difference analysis. The performance of *SBCR* is compared with Random Search-Based Configuration Recommendation (*RBCR*) in terms of six quality indicators (i.e., HV, IGD, Epsilon, ED, GD, and GS) and the percentage of faulty configurations (PFC). Moreover, we also compared the performance of six variants of *SBCR* to find the best performing approach.

The following research questions are addressed in this paper.

RQ1. Is *SBCR* effective to solve the configuration recommendation problem as compared to *RBCR*?

Results of RQ1 show that *SBCR* significantly outperformed *RBCR* in terms of all the indicators except GS for both case studies. In terms of GS, *RBCR* performed significantly better than *SBCR* in 5/6 comparisons for both case studies. In the sixth comparison, *SBCR*$_{SPEA2}$ significantly outperformed *RBCR* in terms of GS for both case studies. Overall, *SBCR* significantly outperformed *RBCR* in 31/36 comparisons for each of the two case studies. Hence, it can be concluded that *SBCR* is more effective than *RBCR* for configuration recommendation problem.

RQ2. Do *SBCR* approaches recommend better quality configurations than *RBCR*?

Results of RQ2 indicate that *SBCR* significantly outperformed *RBCR* in terms of PFC for both case studies. Thus, *SBCR* recommends better quality configurations as compared to *RBCR*.

RQ3. Which one of the six *SBCR* approaches performs the best for the configuration recommendation problem?

Results of RQ3 show that *SBCR$_{SPEA2}$* is the best performing approach, as it significantly outperformed others in 87% and 83% of total comparisons for the Cisco and Jitsi case studies, respectively.

RQ4. Which one of the six SBCR approaches recommends better quality configurations?

Results of RQ4 show that *SBCR$_{SPEA2}$* significantly outperformed others in terms of PFC in all the comparisons for both case studies. This suggests *SBCR$_{SPEA2}$* recommends better quality configurations than others.

RQ5. Is it feasible to apply *SBCR* in practice in terms of the time required for recommending configurations?

Results of RQ5 show that the average time required to recommend configurations by different variants of *SBCR* is quite comparable. All the variants of *SBCR* except *SBCR$_{SMPSO}$* took approximately 3 to 6 minutes, whereas, *SBCR$_{SMPSO}$* took 22.3 minutes. Thus, the proposed approach is feasible in terms of execution cost.

6 Threats to Validity

In this section, we discuss the threats to validity for the entire thesis. In Section 6.1, we discuss threats to the internal validity followed by threats to the construct validity in Section 6.2. We discuss threats to the conclusion validity and external validity in Section 6.3 and Section 6.4, respectively.

6.1 Internal Validity

Threats to *internal validity* consider the internal factors (e.g., parameter settings) that may influence the results [107, 108]. The first threat to *internal validity* is the selection of approaches for solving our rule mining and configuration generation/recommendation problems. To address this, we have combined different techniques from SBSE and machine learning (i.e., multi-objective search and rule mining algorithms), which have been applied in the literature to solve various software engineering problems [36, 56, 58, 59, 61, 88, 109]. The second threat to *internal validity* is the implementation of the algorithms. To address this, we implemented all the selected algorithms using the jMetal framework [110, 111] and Weka [112]. The third threat to *internal validity* is the selection of parameter settings for the selected search algorithms and rule mining algorithms. To mitigate this threat, we used default parameter settings for both search algorithms and rule mining algorithms in *SBRM* and *SBRM$^+$*, which have exhibited promising results [90, 113, 114]. In *SBCR*, we tuned mutation and crossover rates using the iRace optimization package [115-119] and used default settings for other parameters (e.g., archive size) for all the selected search algorithms. The fourth threat to *internal validity* is the selection of the *Confidence* measure for calculating fitness values, as there exist other measures (e.g., *Lift*). We acknowledge that this is a threat to *internal validity* and dedicated experiments are needed for further investigation.

6.2 Construct Validity

Threats to the *construct validity* exist when the comparison metrics are not comparable for all the treatments, or the measurement metrics do not sufficiently cover the concepts they are supposed to measure [54, 58, 107, 120]. To mitigate this threat, we compared different approaches using the same comprehensive set of measures such as fitness values, quality indicators, and MLQMs, which are commonly used in the literature [90, 109, 121]. Another threat to *construct validity* is the use of termination criteria for the search to find the optimal solutions. We used the same stopping criterion (i.e., the number of fitness evaluations) for all the selected search algorithms.

6.3 Conclusion Validity

Threats to the *conclusion validity* concern with the factors influencing the conclusion drawn from the experiment's results [122, 123]. The first threat to *conclusion validity* is the evaluation of configuration tools, which was performed by reading the literature instead of using them. Thus, it is possible that certain features are available in the tool but not reported in the literature. The second threat to *conclusion validity* is due to the random variation inherited in search algorithms. To minimize this threat, we repeated the experiment multiple times (e.g., 30) to reduce the effect caused by randomness, as recommended by existing literature on SBSE [111, 124-126]. Moreover, we also applied the Mann-Whitney test to determine the statistical significance of the results and the Vargha and Delaney $\hat{A}_{12}$ statistics as the effect size measure, which are advocated for randomized algorithms [111, 124, 126].

6.4 External Validity

The *external validity* concerns the factors affecting the generalization of the experiment results to other contexts [107, 108]. The first threat to *external validity* is the selection of case studies for the evaluation of the proposed conceptual framework, rule mining approaches (*SBRM* and *SBRM*$^+$), and configuration recommendation approach (*SBCR*). To address this, 1) for evaluating the framework, we selected three large-scale real-world case studies from three different domains, as representatives of CPS PLs; and 2) for evaluating *SBRM*, *SBRM*$^+$, *SBCR* approaches, we used one industrial case study and one open-source case study of different complexity. The second threat to *external validity* is the selection of VMTs and configuration tools. To address this, we selected four representative VMTs and 11 configuration tools, as evaluating all possible VMTs and configuration tools is infeasible. The third threat to *external validity* is the completeness of the framework. To address this, we evaluated the framework using multiple real-world case studies, modeling standards (i.e., SysML and MARTE), and an extensive literature review. Despite a thorough evaluation, the completeness of the framework cannot be fully ensured as there might be some new requirements (e.g., new variation point or constraint types) in the future. The fourth threat to *external validity* is the selection of multi-objective search and rule mining algorithms. To mitigate this threat, we selected several state-of-the-art algorithms (e.g., NSGA-II [64, 65], IBEA [72], SPEA2 [73], C4.5 [86], PART [89]), which have been widely used in the literature and industry [36, 64, 65, 88, 90, 127]. Note, such threats to *external validity* are quite common in empirical studies [128, 129].

7 Future Directions

In this section, we discuss the possible future research directions based on the work presented in this thesis. Future research work can target four research streams as follows:

Modeling CPS PLs: As shown by the results of Paper-A, there does not exist a VMT in the literature that can cater all the requirements of modeling CPS PLs. Thus, we need to extend an existing VMT or propose a new one to support domain engineering of CPS PLs by following the guidelines provided in Paper-B.

Configuring CPS products: The results of Paper-B show that existing tools do not provide all necessary functionalities for automating the configuration in CPS PLE. Also, these tools are built on top of existing VMTs, which do not cater all the requirements of CPS PLs. Therefore, we need to build a configuration tool based on a VMT specific to CPS PLs that support all the required functionalities, as mentioned in Paper-B.

Automatic post-deployment configurations for end-users: In Paper-E, we proposed an approach for recommending faulty configurations for SUT based on CPL rules. Similarly, we need an approach to recommend non-faulty configurations for interacting products such that end-users can correct the configurations automatically when the communication fails due to invalid configurations.

Improving empirical evaluations: The empirical evaluations of the work presented in the thesis can be improved in various aspects: 1) In *SBRM*$^+$ (Paper-D), we used only NSGA-II and NGSA-III combined with C4.5 and PART with their default parameter settings to mine rules for two case studies. This can be improved by integrating more algorithms (both search and machine learning algorithms) with their best parameter settings to mine the rules for more complex case studies. 2) For both *SBRM*$^+$ (Paper-D) and *SBCR* (Paper-E), we used only one interestingness measure (i.e., the confidence of CPL rules) to define search objectives. This can be improved by conducting an extensive empirical study to assess the impact of different interestingness measures (e.g., Lift). 3) The applicability of *SBCR* (Paper-E) needs to be evaluated using more complex case studies.

8 Conclusion

This thesis proposed a set of methods to address various challenges related to Product Line Engineering (PLE) of Cyber-Physical Systems (CPSs) with the focus on the post-deployment configuration. More specifically, we made three main contributions: 1) we conducted a systematic domain analysis and proposed a conceptual framework for CPS product lines (PLs) in addition to evaluating existing PLE methodologies; 2) we proposed an approach to capture the patterns of configurations in form of configuration rules for CPSs consisting multiple interacting products; and 3) we proposed an approach to recommend configurations for CPSs based on mined rules, to improve the post-deployment configuration experience for testers and end-users.

To conduct the domain analysis, we analyzed three CPS case studies. Based on the knowledge collected from the domain analysis and an extensive literature review on PLE, we proposed a conceptual framework, which 1) formalizes CPS, PLE, and configuration process to clarify the context of CPS PLE; 2) presents classifications of variation point, constraint, and view types in addition to different modeling concepts to support domain engineering of CPS PLs; and 3) formalizes 14 types of automation to tackle application engineering of CPS PLs. We evaluated the completeness of the framework using three real-world CPS case studies containing 2161 VPs, 3943 constraints, and 40 views, 11 configuration tools, and an extensive literature review. Results showed that the framework fulfills the requirements of CPS case studies and caters various aspects concerned by the literature. Moreover, we also evaluated four representative variability modeling techniques (VMTs) by modeling a CPS case study to assess if they can capture the variabilities of CPS PLs. Results show that none of the four VMTs fulfills the requirements of CPS PLs.

To capture the configuration patterns in form of configuration rules, we proposed Search-Based Rule Mining (*SBRM*$^+$) approach, which combines multi-objective search with machine learning to mine the configuration rules in an incremental and iterative way. We evaluated the performance of *SBRM*$^+$ using industrial and open-source case studies and compared its performance with Random Search Based Rule Mining (*RBRM*$^+$). Results show that *SBRM*$^+$ improved the quality of rules based on machine learning quality measurements up to 28%, in comparison to *RBRM*$^+$.

To improve the post-deployment configuration experience, we proposed a Search-Based Configuration Recommendation (*SBCR*) approach. *SBCR* recommends faulty configurations for CPSs with interacting products under test based on mined configuration rules. The recommended configurations can be used to test CPS and create guidelines for end-users to improve the post-deployment configuration experience of testers and end-users. We evaluated the *SBCR* using the same case studies for which we mined the rules using *SBRM*$^+$. We compared the performance of *SBCR* with Random Search-Based Configuration Recommendation (*RBCR*). Results showed that *SBCR* performed significantly better than *RBCR*, as it made up to 22% more accurate recommendations than *RBCR*.

9 References for Summary

1. Derler, P., E.A. Lee, and A.S. Vincentelli, *Modeling Cyber–Physical Systems.* Proceedings of the IEEE Special issue on CPS, 2012. **100**(1): p. 13-28.
2. *Cyber-Physical Systems (CPSs)*. Available from: http://cyberphysicalsystems.org/.
3. Rawat, D.B., J.J. Rodrigues, and I. Stojmenovic, *Cyber-Physical Systems: From Theory to Practice*. 2015: CRC Press.
4. Ma, T., S. Ali, and T. Yue, *Modeling foundations for executable model-based testing of self-healing cyber-physical systems.* Software & Systems Modeling, 2019. **18**(5): p. 2843-2873.
5. Zhang, M., et al., *Uncertainty-Wise Cyber-Physical System test modeling.* Software & Systems Modeling, 2017: p. 1-40.
6. Nie, K., et al. *Constraints: the core of supporting automated product configuration of cyber-physical systems*. in *Proceeding of International Conference on Model-Driven Engineering Languages and Systems (MODELS)*. 2013. Springer.
7. Iglesias, A., et al. *Product line engineering of monitoring functionality in industrial cyber-physical systems: A domain analysis*. in *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*. 2017. ACM.
8. Arrieta, A., et al., *Search-Based test case prioritization for simulation-Based testing of cyber-Physical system product lines.* Journal of Systems and Software, 2019. **149**: p. 1-34.
9. Wang, S., et al., *Automatic selection of test execution plans from a video conferencing system product line*, in *VARiability for You Workshop: Variability Modeling Made Useful for Everyone*. 2012, ACM: Innsbruck, Austria. p. 32-37.
10. Chen, L., M. Ali Babar, and N. Ali, *Variability management in software product lines: A systematic review*, in *13th International Software Product Line Conference*. 2009. p. 81-90.
11. Arrieta, A., G. Sagardui, and L. Etxeberria, *A comparative on variability modelling and management approach in simulink for embedded systems.* V Jornadas de Computación Empotrada, ser. JCE, 2014.
12. Djebbi, O. and C. Salinesi. *Criteria for comparing requirements variability modeling notations for product lines*. in *4th International Workshop on Comparative Evaluation in Requirements Engineering*. 2006. IEEE.
13. Sinnema, M. and S. Deelstra, *Classifying variability modeling techniques.* Information and Software Technology, 2007. **49**(7): p. 717-739.
14. Eichelberger, H. and K. Schmid. *A systematic analysis of textual variability modeling languages*. in *Proceedings of the 17th International Software Product Line Conference*. 2013. ACM.

15. Dhungana, D., P. Grünbacher, and R. Rabiser, *Domain-specific adaptations of product line variability modeling*, in *Situational Method Engineering: Fundamentals and Experiences*. 2007, Springer. p. 238-251.
16. Clauß, M. and I. Jena. *Modeling variability with UML*. in *GCSE 2001 Young Researchers Workshop*. 2001. Citeseer.
17. Ziadi, T., L. Hélouët, and J.-M. Jézéquel, *Towards a UML profile for software product lines*, in *Software Product-Family Engineering*. 2004, Springer. p. 129-139.
18. Behjati, R., et al., *SimPL: a product-line modeling methodology for families of integrated control systems.* Information and Software Technology (IST), 2013.
19. Haugen, Ø. and O. Øgård, *BVR–Better Variability Results*, in *System Analysis and Modeling: Models and Reusability*. 2014, Springer. p. 1-15.
20. Pure-Systems. *Pure::Variants available at: http://www.pure-systems.com/*. 2017].
21. Rabiser, R., et al. *DOPLER, Decision Oriented Product Line Engineering for effective Reuse*. Available from: http://ase.jku.at/dopler/.
22. Sinnema, M., et al., *Covamof: A framework for modeling variability in software product families*, in *Software product lines*, R.L. Nord, Editor. 2004, Springer Heidelberg. p. 197-213.
23. Mendonca, M., M. Branco, and D. Cowan. *SPLOT: software product lines online tools*. in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009. ACM.
24. Nöhrer, A. and A. Egyed, *C2O configurator: a tool for guided decision-making.* Automated Software Engineering, 2013. **20**(2): p. 265-296.
25. Thüm, T., et al., *FeatureIDE: An extensible framework for feature-oriented software development.* Science of Computer Programming, 2014. **79**: p. 70-85.
26. Hong, L., Y. Tao, and A. Shaukat. *Zen-Configurator: Interactive and Optimal Configuration of Cyber Physical System Product Lines*. [cited 2017; Available from: https://www.simula.no/research/projects/zen-configurator-interactive-and-optimal-configuration-cyber-physical-system.
27. ISO, *Software and systems engineering -- Reference model for product line engineering and management*. 2013, ISO.
28. Berger, T., et al. *A survey of variability modeling in industrial practice*. in *Proceedings of 7th International Workshop on Variability Modelling of Software intensive Systems*. 2013. ACM.
29. Czarnecki, K., S. Helsen, and U. Eisenecker, *Staged configuration using feature models*, in *Software Product Lines*. 2004, Springer. p. 266-283.
30. Haugen, Ø., A. Wąsowski, and K. Czarnecki. *CVL: common variability language*. in *Proceedings of the 16th International Software Product Line Conference-Volume 2*. 2012.
31. Arrieta, A., G. Sagardui, and L. Etxeberria. *A model-based testing methodology for the systematic validation of highly configurable cyber-physical systems*. in *The Sixth International Conference on Advances in System Testing and Validation Lifecycle*. 2014. IARIA XPS Press.
32. Safdar, S.A., et al., *Using multi-objective search and machine learning to infer rules constraining product configurations.* Automated Software Engineering (ASE), 2019. **26**(4): p. 1-62.
33. Safdar, S.A., et al., *Mining Cross Product Line Rules with Multi-Objective Search and Machine Learning* in *The Genetic and Evolutionary Computation Conference (GECCO)*. 2017, ACM: Berlin, Germany. p. 1319-1326.
34. Nadi, S., et al., *Where do configuration constraints stem from? an extraction approach and an empirical study.* IEEE Transactions on Software Engineering (TSE), 2015. **41**(8): p. 820-841.
35. Temple, P., et al. *Using Machine Learning to Infer Constraints for Product Lines*. in *Proceeding of International Systems and Software Product Line Conference (SPLC)*. 2016. ACM.
36. Frank, E. and I.H. Witten. *Generating accurate rule sets without global optimization*. in *Proceeding of International Conference on Machine Learning (ICML)*. 1998. University of Waikato, Department of Computer Science.
37. Hervieu, A., et al., *Practical minimization of pairwise-covering test configurations using constraint programming.* Information and Software Technology, 2016. **71**: p. 129-146.
38. Marijan, D., et al. *Practical pairwise testing for software product lines*. in *Proceedings of the 17th international software product line conference*. 2013.
39. Cohen, M.B., M.B. Dwyer, and J. Shi, *Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach.* IEEE Transactions on Software Engineering, 2008. **34**(5): p. 633-650.

40. Perrouin, G., et al. *Automated and scalable t-wise test case generation strategies for software product lines*. in *2010 Third international conference on software testing, verification and validation*. 2010. IEEE.
41. Pohl, K., G. Böckle, and F.J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. 2005: Springer Science & Business Media.
42. ISO13628-6, *Petroleum and natural gas industries-Design and operation of subsea production system-Part 6:Subsea production control systems*. 2006.
43. Lu, H., et al., *Model-based Incremental Conformance Checking to Enable Interactive Product Configuration.* Information and Software Technology (IST), 2015. **72**: p. 68-89.
44. Lu, H., et al., *Nonconformity Resolving Recommendations for Product Line Configuration*, in *International Conference on Software Testing*. 2016, IEEE: Chicago, USA. p. 57-68.
45. Yue, T., S. Ali, and B. Selic, *Cyber-Physical System Product Line Engineering: Comprehensive Domain Analysis and Experience Report*, in *International Systems and Software Product Line Conference (SPLC)*. 2015, ACM. p. 338-347.
46. Mazo, R., et al. *Using constraint programming to verify DOPLER variability models*. in *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. 2011. ACM.
47. Mendonça, M., T.T. Bartolomei, and D. Cowan. *Decision-making coordination in collaborative product configuration*. in *Proceedings of the 2008 ACM symposium on Applied computing*. 2008. ACM.
48. Yue, T., et al., *Search-based decision ordering to facilitate product line engineering of cyber-physical system*, in *4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2016, IEEE. p. 691-703.
49. Harman, M. and B.F. Jones, *Search-based software engineering.* Information and software Technology, 2001. **43**(14): p. 833-839.
50. Zitzler, E., K. Deb, and L. Thiele, *Comparison of multiobjective evolutionary algorithms: Empirical results.* Evolutionary Computation, 2000. **8**(2): p. 173-195.
51. Sayyad, A.S. and H. Ammar. *Pareto-optimal search-based software engineering (POSBSE): A literature survey*. in *Proceedings of the 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. 2013. IEEE.
52. Srinvas, N. and K. Deb, *Multi-objective function optimization using non-dominated sorting genetic algorithms.* Evolutionary Computation, 1994. **2**(3): p. 221-248.
53. Coello, C.A.C., D.A. Van Veldhuizen, and G.B. Lamont, *Evolutionary algorithms for solving multi-objective problems*. Vol. 242. 2002: Springer.
54. Li, Z., M. Harman, and R.M. Hierons, *Search algorithms for regression test case prioritization.* IEEE Transactions on Software Engineering, 2007. **33**(4): p. 225-237.
55. Reeves, C.R., *Modern heuristic techniques for combinatorial problems*. 1993: John Wiley & Sons, Inc.
56. Lopez-Herrejon, R.E., L. Linsbauer, and A. Egyed, *A systematic mapping study of search-based software engineering for software product lines.* Information and Software Technology (IST), 2015. **61**: p. 33-51.
57. Harman, M., et al., *Search based software engineering for software product line engineering: a survey and directions for future work*, in *International Systems and Software Product Line Conference (SPLC)*. 2014, ACM. p. 5-18.
58. Wang, S., S. Ali, and A. Gotlieb, *Cost-effective test suite minimization in product lines using search techniques.* Journal of Systems and Software (JSS), 2014. **103**: p. 370-391.
59. Sayyad, A.S., T. Menzies, and H. Ammar, *On the Value of User Preferences in Search-Based Software Engineering: A Case Study in Software Product Lines*, in *International Conference on Software Engineering (ICSE)*. 2013, IEEE. p. 492-501.
60. Yu, H., et al. *Combining constraint solving with different MOEAs for configuring large software product lines: a case study*. in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. 2018. IEEE.
61. Sayyad, A.S., et al. *Scalable product line configuration: A straw to break the camel's back.* in *Proceeding of International Conference on Automated Software Engineering (ASE)*. 2013. IEEE.
62. Guo, J., et al., *SMTIBEA: A hybrid multi-objective optimization algorithm for configuring large constrained software product lines.* Software & Systems Modeling (SoSyM), 2017. **16**(4): p. 1-20.
63. Brownlee, J., *Clever algorithms: nature-inspired programming recipes*. 2011: Lulu.
64. Deb, K., et al., *A fast and elitist multiobjective genetic algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation, 2002. **6**(2): p. 182-197.
65. Sarro, F., A. Petrozziello, and M. Harman, *Multi-objective software effort estimation*, in *International Conference on Software Engineering (ICSE)*. 2016, ACM. p. 619-630.

66. Konak, A., D.W. Coit, and A.E. Smith, *Multi-objective optimization using genetic algorithms: A tutorial.* Reliability Engineering & System Safety (RESS), 2006. **91**(9): p. 992-1007.
67. Deb, K. and H. Jain, *An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints.* IEEE Trans. Evolutionary Computation, 2014. **18**(4): p. 577-601.
68. Jain, H. and K. Deb, *An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach.* IEEE Trans. Evolutionary Computation, 2014. **18**(4): p. 602-622.
69. Mkaouer, M.W., et al. *High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III.* in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 2014. ACM.
70. Nebro, A., et al., *Design Issues in a Multiobjective Cellular Genetic Algorithm*. Evolutionary Multi-Criterion Optimization, ed. S. Obayashi, et al. Vol. 4403. 2007: Springer Berlin Heidelberg. 126-140.
71. Nebro, A.J., et al., *Mocell: A cellular genetic algorithm for multiobjective optimization.* International Journal of Intelligent Systems, 2009. **24**(7): p. 726-746.
72. Zitzler, E. and S. Künzli, *Indicator-based selection in multiobjective search*, in *International Conference on Parallel Problem Solving from Nature*. 2004, Springer. p. 832-842.
73. Zitzler, E., M. Laumanns, and L. Thiele, *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*, in *the EUROGEN Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*. 2001. p. 95-100.
74. Knowles, J. and D. Corne. *The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation*. in *Proceedings of the Congress on Evolutionary Computation*. 1999. IEEE.
75. Knowles, J.D. and D.W. Corne, *Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy.* IEEE Transactions on Evolutionary Computation, 2000. **8**(2): p. 149-172.
76. Brownlee, J., *Clever Algorithms: Nature-Inspired Programming Recipes*. 2012: lulu.com; 1ST edition.
77. Nebro, A.J., et al., *SMPSO: A new PSO-based metaheuristic for multi-objective optimization*, in *IEEE symposium on Computational intelligence in multi-criteria decision-making (MCDM)* 2009. p. 66-73.
78. Arcuri, A. and G. Fraser, *On Parameter Tuning in Search Based Software Engineering*, in *International Symposium on Search Based Software Engineering (SSBSE)*. 2011, Springer's Lecture Notes in Computer Science (LNCS)
79. Arcuri, A., *It really does matter how you normalize the branch distance in search‐based software testing.* Software Testing, Verification and Reliability, 2013. **23**(2): p. 119-147.
80. Arcuri, A., M.Z. Iqbal, and L. Briand, *Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing*, in *IFIP International Conference on Testing Software and Systems (ICTSS)*. 2010. p. 95-110.
81. McMinn, P., *Search-based software test data generation: a survey.* Software Testing Verification and Reliability (STVR), 2004. **14**(2): p. 105-156.
82. Ali, S., et al., *Generating test data from OCL constraints with search techniques.* IEEE Transactions on Software Engineering (TSE), 2013. **39**(10): p. 1376-1402.
83. Han, J., M. Kamber, and J. Pei, *Data mining: concepts and techniques*. 3rd ed. 2012: Elsevier. 703.
84. Davril, J.-M., et al. *Feature model extraction from large collections of informal product descriptions*. in *Proceeding of Joint Meeting on Foundations of Software Engineering (FSE)*. 2013. ACM.
85. Maimon, O. and L. Rokach, *Introduction to knowledge discovery and data mining*, in *Data Mining and Knowledge Discovery Handbook*. 2009, Springer. p. 1-15.
86. Quinlan, J.R., *C4.5: Programming for machine learning*. 1st ed. 1993, London, UK: Morgan Kauffmann. 302.
87. Cohen, W.W. *Fast effective rule induction*. in *Proceeding of International Conference on Machine Learning (ICML)*. 1995. Morgan Kaufmann.
88. Holmes, G., M. Hall, and E. Prank. *Generating rule sets from model trees*. in *Proceeding of Australasian Joint Conference on Artificial Intelligence (AI)*. 1999. Springer.
89. Frank, E. and I.H. Witten, *Generating accurate rule sets without global optimization.* 1998.
90. Witten, I.H. and E. Frank, *Data Mining: Practical machine learning tools and techniques*. 2nd ed. 2005, San Francisco,USA: Diane Cerra. 525.
91. Lloyd, S., *Least squares quantization in PCM.* IEEE Transactions on Information Theory, 1982. **28**(2): p. 129-137.
92. *Euclidean distance*. 2002 2017; Available from: https://wikipedia.org/wiki/Euclidean_distance.
93. Vierhauser, M., et al., *Flexible and scalable consistency checking on product line variability models*, in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, ACM: Antwerp, Belgium. p. 63-72.

94. Henard, C., et al., *Combining multi-objective search and constraint solving for configuring large software product lines*, in *37th International Conference on Software Engineering-Volume 1*. 2015, IEEE Press. p. 517-528.
95. Czarnecki, K., S. Helsen, and U. Eisenecker, *Staged configuration through specialization and multilevel configuration of feature models.* Software Process: Improvement and Practice, 2005. **10**(2): p. 143-169.
96. Mukelabai, M., et al., *Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems*, in *33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018. p. 155-166.
97. Safdar, S.A., et al., *A Framework for Automated Multi-Stage and Multi-Step Product Configuration of Cyber-Physical Systems.* Software and Systems Modeling (SoSym), 2020. **19**(3).
98. *End-user guide manuals. https://www.cisco.com/c/en/us/support/collaboration-endpoints/telepresence-mx-series/products-user-guide-list.html*. 2020, Cisco Systems.
99. Wu, J., et al., *Assessing the quality of industrial avionics software: an extensive empirical evaluation.* Empirical Software Engineering (EMSE), 2016. **22**(4): p. 1-50.
100. Mann, H.B. and D.R. Whitney, *On a test of whether one of two random variables is stochastically larger than the other.* The Annals of Mathematical Statistics, 1947: p. 50-60.
101. Vargha, A. and H.D. Delaney, *A critique and improvement of the CL common language effect size statistics of McGraw and Wong.* Journal of Educational and Behavioral Statistics, 2000. **25**(2): p. 101-132.
102. Sheskin, D.J., *Handbook of Parametric and Nonparametric Statistical Procedures*. 3rd ed. 2007, London,UK: Chapman and Hall, CRC Press. 1776.
103. Yue, T., S. Ali, and B. Selic. *Cyber-physical system product line engineering: comprehensive domain analysis and experience report*. in *Proceedings of the 19th International Conference on Software Product Line*. 2015. ACM.
104. *The UML MARTE profile, http://www.omgmarte.org/*.
105. OMG, *Systems Modeling Language (SysML) v1.4, http://sysml.org/*. 2015.
106. Safdar, S.A., et al. *Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering*. in *Proceeding of International Conference on System Analysis and Modeling (SAM)*. 2016. Springer.
107. Runeson, P., et al., *Case study research in software engineering: Guidelines and examples*. 1st ed. 2012, New Jersey, USA: John Wiley & Sons. 237.
108. Runeson, P., et al., *Case study research in software engineering: Guidelines and examples*. 2012: John Wiley & Sons.
109. Pradhan, D., et al., *Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study*, in *The Genetic and Evolutionary Computation Conference (GECCO)*. 2016, ACM: Denver, Colorado, USA. p. 1085-1092.
110. Durillo, J.J. and A.J. Nebro, *jMetal: A Java framework for multi-objective optimization.* Advances in Engineering Software, 2011. **42**(10): p. 760-771.
111. Arcuri, A. and L. Briand. *A practical guide for using statistical tests to assess randomized algorithms in software engineering*. in *Proceeding of International Conference on Software Engineering (ICSE)*. 2011. IEEE.
112. Witten, I.H., E. Frank, and M.A. Hall, *Data Mining: Practical machine learning tools and techniques*. Third ed. 2011: Morgan Kaufmann.
113. Arcuri, A. and G. Fraser. *On parameter tuning in search based software engineering*. in *Proceeding of International Symposium On Search Based Software Engineering (SSBSE)*. 2011. Springer.
114. Witten, I.H., E. Frank, and M.A. Hall, *Data Mining: Practical machine learning tools and techniques*. 4th ed. 2016, Switzerland: Morgan Kaufmann. 734.
115. López-Ibánez, M., et al., *The irace package, iterated race for automatic algorithm configuration*. 2011, Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles.
116. Liao, T., M.A.M. de Oca, and T. Stützle, *Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set.* Soft Computing, 2013. **17**(6): p. 1031-1046.
117. Caceres, L.P., M. López-Ibáñez, and T. Stützle, *Ant colony optimization on a limited budget of evaluations.* Swarm Intelligence, 2015. **9**(2-3): p. 103-124.
118. Ren, Z., et al., *Feature based problem hardness understanding for requirements engineering.* Science China Information Sciences, 2017. **60**(3): p. 032105.

119. Bezerra, L.C., M. López-Ibáñez, and T. Stützle, *Automatic configuration of multi-objective optimizers and multi-objective configuration*. High-Performance Simulation-Based Optimization. 2020, Cham: Springer. 69-92.
120. Kitchenham, B., L. Pickard, and S.L. Pfleeger, *Case studies for method and tool evaluation.* IEEE Software, 1995. **12**(4): p. 52.
121. Sokolova, M. and G. Lapalme, *A systematic analysis of performance measures for classification tasks.* Information Processing & Management (IPM), 2009. **45**(4): p. 427-437.
122. Wohlin, C., et al., *Experimentation in software engineering: an introduction*. 2000: Kluwer Academic Publishers. 204.
123. Wohlin, C., et al., *Experimentation in software engineering: an introduction. 2000*. 2000, Kluwer Academic Publishers.
124. Arcuri, A. and L. Briand, *A practical guide for using statistical tests to assess randomized algorithms in software engineering*, in *33rd International Conference on Software Engineering*. 2011, IEEE. p. 1-10.
125. Wang, S., et al., *A Practical Guide to Select Quality Indicators for Assessing Pareto-based Search Algorithms in Search-Based Software Engineering*, in *International Conference on Software Engineering (ICSE)*. 2016.
126. Wang, S., et al., *A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering*, in *International Conference on Software Engineering (ICSE)*. 2016, ACM. p. 631-642.
127. Wu, X., et al., *Top 10 algorithms in data mining.* Knowledge and Information Systems (KAIS), 2008. **14**(1): p. 1-37.
128. Wang, S., et al. *UPMOA: An improved search algorithm to support user-preference multi-objective optimization*. in *Proceedings of the 26th International Symposium on Software Reliability Engineering*. 2015. IEEE.
129. Sarro, F., A. Petrozziello, and M. Harman. *Multi-objective software effort estimation*. in *Proceedings of the 38th International Conference on Software Engineering*. 2016. ACM.

Part II
Papers

Paper A

Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering

Safdar Aqeel Safdar, Tao Yue, Shaukat Ali, Hong Lu

Published in the Proceedings of 9th International Conference on System Analysis and Modeling (SAM), 2016.

Abstract

Modern society is increasingly dependent on Cyber-Physical Systems (CPSs) in diverse domains such as aerospace, energy and healthcare. Employing Product Line Engineering (PLE) in CPSs is cost-effective in terms of reducing production cost, and achieving high productivity of a CPS development process as well as higher quality of produced CPSs. To apply CPS PLE in practice, one needs to first select an appropriate variability modeling technique (VMT), with which variabilities of a CPS Product Line (PL) can be specified. In this paper, we proposed a set of basic and CPS-specific variation point (VP) types and modeling requirements for proposing CPS-specific VMTs. Based on the proposed set of VP types (basic and CPS-specific) and modeling requirements, we evaluated four VMTs: Feature Modeling, Cardinality Based Feature Modeling, Common Variability Language, and SimPL (a variability modeling technique dedicated to CPS PLE), with a real-world case study. Evaluation results show that none of the selected VMTs can capture all the basic and CPS-specific VP and meet all the modeling requirements. Therefore, there is a need to extend existing techniques or propose new ones to satisfy all the requirements.

Keywords: Product Line Engineering, Variability Modeling, and Cyber-Physical Systems

1 Introduction

Cyber-Physical Systems (CPSs) integrate computation and physical processes and their embedded computers and networks monitor and control physical processes by often relying on closed feedback loops [1, 2]. Nowadays, CPSs can be found in many different domains such as energy, maritime and healthcare. Many CPS producers employ the Product Line Engineering (PLE) practice, aiming to improve the overall quality of produced CPSs and the productivity of their CPS development processes [3].

In [4], a systematic domain analysis of the CPS PLE industrial practice is presented, which focuses on capturing static variabilities and facilitating product configuration at the pre-deployment phase. The systematic domain analysis identifies the following key characteristics of CPS PLE: (1) CPSs are heterogeneous and hierarchical systems; (2) the hardware topology can vary from one product to another; (3) the generic software code base might be instantiated differently for each product, mainly based on the hardware topology configuration; and (4) there are many dependencies among configurable parameters, especially across the software code base and the hardware topology. Various challenges in CPS PLE were also reported in [4] such as lacking of automation and guidance and expensive debugging of configuration data. In general, cost-effectively supporting CPS PLE, especially enabling automation of product configuration, is an industrial challenge.

Cost-effectiveness of PLE is characterized by its support for abstraction and automation. Generally speaking, abstraction is a key mean that enables reuse. Concise and expressive abstractions for CPS PLE are required to specify reusable artifacts at a suitable level of abstraction as commonalities and variabilities. Such abstractions are quite critical and provide the foundation for automation. To capture variabilities at a high level of abstraction, a number of variability modeling techniques (VMTs) are available in the literature, including Feature Modeling (FM) [5], Cardinality Based Feature Modeling (CBFM) [6], a UML-based variability modeling methodology named SimPL [7], and Common Variability Language (CVL) [8]. These VMTs were proposed for a particular context/domain/purpose. For example, SimPL was designed for the architecture level variability modeling. It is however no evidence showing which VMT suits CPS PLE the best.

In this paper, we propose a set of basic variation point (VP) types, CPS-specific VP types, and modeling requirements of CPS PLE. To define basic VP types, we constructed a conceptual model for basic data types in mathematics. Corresponding to each basic data type, we defined one basic VP type (Section 4.1). We also constructed a conceptual model for CPS based on the knowledge gathered from literature about CPSs and our experience of working with industry [4]. The second and third authors of the paper have experience of working with industrial CPS case studies and have derived the conceptual model. From the CPS conceptual model, we systematically derived a set of CPS-specific VP types (Section 4.2). We also derived a set of modeling requirements based on the literature and our experience in working with industry [4] (Section 5). Based on the proposed basic and CPS-specific VP types and the modeling requirements, we evaluated FM [5], CBFM [6] , CVL [8], and SimPL [7]. FM was selected as it is the most widely used VMT in industry [9] and CBFM is an extension of FM. CVL is a language for modeling variability using any domain specific language based on Meta Object Facility (MOF), which was submitted to Object Management Group for standardization but did not go through due to Intellectual Property Rights issues. SimPL is a specific VMT dedicated for CPS PLE and has been applied to address industrial challenges. To evaluate the VMTs, we modeled a case study (Material Handling System-MHS) with all the VMTs and evaluated them using the proposed eight basic and 16 CPS-specific VP types, and nine modeling requirements.

Results of the evaluation show that 1) only SimPL and CVL can capture all the basic VP types, whereas FM and CBFM provide partial support. None of the four VMTs can capture all the CPS-specific VP types; 2) SimPL and CVL provide support for 81% and 75% of the total CPS-specific VP types respectively, whereas CBFM supports 50% and FM supports only 15% of the total CPS-specific VP types; 3) SimPL satisfies all but one of the modeling requirements, FM and CBFM only covers one modeling requirement, and CVL fully or partially fulfills four requirements out of nine requirements. Based on above results, we can conclude that it is required to either extend an existing technique or propose a new one to facilitate the variability modeling in the context of CPS PLE. The proposed VP types and modeling requirements can be also used as evaluation criteria for selecting existing VMTs or defining new ones for a particular application when necessary.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 presents the context of the work. Section 4 presents the proposed VP types. Section 5 presents the modeling requirements. In Section 0, we report evaluation results. Threats to validity are given in Section 7. Section 8 concludes the paper.

2 Related Work

This section discusses the existing literature that compares or classifies VMTs, systematic literature reviews (SLRs) and surveys of VMTs.

Galster et al. [10] conducted a SLR of 196 papers published during 2000-2011, on variability management in different phases of software systems. Results show that most of the papers focus on design time variabilities and a small portion of the papers focus on runtime variabilities. In

[11], Chen et al. conducted a SLR of 33 VMTs in software product lines and highlighted the challenges involved in variability modeling such as evolution of variability, and configuration. Arrieta et al. [12] conducted a SLR of variability management techniques, but limited their scope to techniques for Simulink published after 2008. Berger et al. [9] conducted a survey on industry practices of variability modeling using a questionnaire, aiming to discover characteristics of industrial variability models, VMTs, tools and processes. Another industrial survey of feature-based requirement VMTs was conducted to find out the most appropriate technique for a company [13]. They evaluated existing techniques based on requirements collected from the company's engineers, including readability, simplicity and expressive, types of variability and standardization.

Eichelberger and Schmid [14] classified and compared 10 textual VMTs in terms of scalability. They compared the selected techniques in five different aspects: configurable elements, constraints support, configuration support, scalability, and additional language characteristics. Similarly, Sinnema and Deelstra [15] classified six VMTs and compared them based on key characteristics of VMTs such as constraints, tool support, and configuration guidance. Czarnecki et al. [16] reported an experience report, in which they compared two types of VMTs: decision modeling and feature modeling. They compared them in 10 aspects: application, hierarchy, unit of variability, data types, constraints, modularity, orthogonality, mapping to artifacts, tool support, and binding time and mode. A comparative study [17] was reported to compare two VMTs, i.e., Kconfig and CDL, in the context of operating systems, in terms of constructs, semantics, and tool support.

All the above studies classify and evaluate various types of VMTs either in general or for a particular domain other than CPSs. We however, in this paper, propose a set of basic and CPS-specific VP types as well as a list of modeling requirements for evaluating VMTs in the context of CPS PLE, based on which we evaluated four representative VMTs with a non-trivial case study.

3 Context

Section 3.1 and 3.2 introduce the case study and the four VMTs. In Section 3.3, we present the study procedure.

3.1 Case Study

The case study is a product line of Handling Systems, which consist of various types of sub-systems such as Automatic Storage Retrieval System (ASRS), Automatic Guided Vehicle (AGV), Automatic Identification and Data Collection (AIDC) and Warehouse Management System. We selected three of these systems: AGV, AIDC, and ASRS for the evaluation of the selected VMTs. AGV is a fully automatic transport system that uses unmanned vehicles to transport all types of loads without human intervention. It is typically used within warehouse, production and logistics for safe movement of goods. AIDC is used to identify, verify, record, and track the products. Typically, these systems are used in supply chain, order picking, order fulfillment, and determination of weight, volume, and storage. ASRS is an automated system for inventory management, which is used to place and retrieve the loads from pre-defined locations in the warehouse. The descriptive statistics of the MHS case study's class diagram are given in Table A-

1. We modeled the case study (MHS) using the four selected VMTs (i.e., FM, CBFM, SimPL, and CVL). The case study models corresponding to selected VMTs are available at [18].

Table A-1. Descriptive statistics of the MHS

Element	Count
Class	132
Generalization	56
Composition	62
Association	69
Simple attribute	113
Enumerated attribute	82
Enumeration	23
Enumeration Literal	73

3.2 Variability Modeling Techniques

Feature Modeling (FM) is widely applied in practice [9]. A feature model is organized hierarchically as a tree. The root node of the tree represents the system, whereas the descendent nodes are functionalities of the system (features). A feature can be mandatory, optional or alternative. A feature can either be a compound feature that has one or more descendent features or a leaf feature with no descendent features. Figure A-1 shows an excerpt of the FM model for AGV modeled using Pure::Variants [19]. As shown in Figure A-1, *AGVHardware, Sensor,* and *Connectivity* are mandatory features. The *Connectivity* feature has three alternative features, i.e., *Bluetooth*, *Wifi*, and *NFC*. The *Sensor* feature has two optional features: *MultiRayLEDScanner and LaserScanner.*

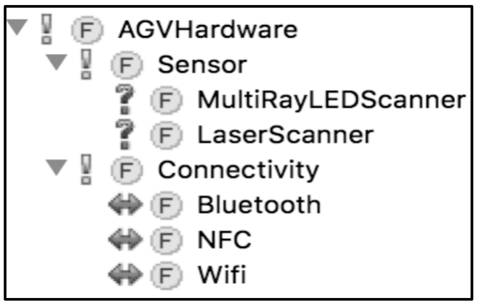
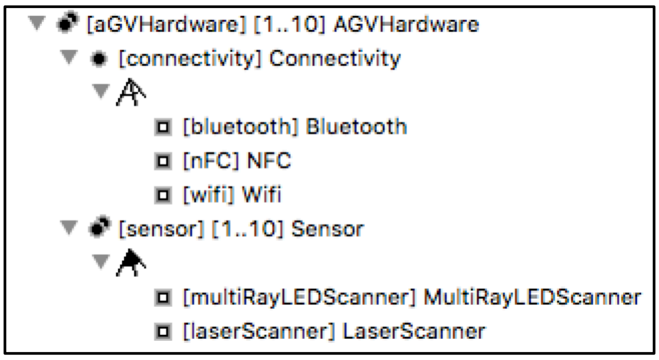


Figure A-1. An excerpt of FM for AGV

Cardinality Based Feature Modeling (CBFM) is an extension to FM, which introduces new concepts such as Feature Cardinalities, Groups and Groups Cardinalities, Attributes, and References. For Feature Cardinalities, features can be annotated with cardinalities such as <1..*> whereas alternative features and optional features are special cases with cardinality <1..1> and <0..1> respectively. A feature group can be or-group with cardinality <1..k> or alternative-group with cardinality <1..1>. For an alternative-group, one can select only one feature, whereas for or-group, one can select 1 to *k* number of features where *k* is the maximum number of features in the group. A feature can have one attribute of either String or Integer type. To achieve better modularization, a special leaf node (i.e., Reference) was introduced to refer to another feature model. This can be used to divide a large feature model into smaller ones to support modularization. As shown in Figure A-2 *AGVHardware, Sensor*, and *Connectivity* are mandatory features. *AGVHardware* and *Sensor* have feature cardinality <1..10>. *Connectivity* has an alternative-group that consists of three features: *Bluetooth*, *Wifi*, and *NFC*. The *Sensor* feature has an or-group consisting of two features with group cardinality <0..2>.



```
▼ [aGVHardware] [1..10] AGVHardware
  ▼ [connectivity] Connectivity
    ▼
        [bluetooth] Bluetooth
        [nFC] NFC
        [wifi] Wifi
  ▼ [sensor] [1..10] Sensor
    ▼
        [multiRayLEDScanner] MultiRayLEDScanner
        [laserScanner] LaserScanner
```

Figure A-2. An excerpt of CBFM for AGV

Common Variability Modeling (CVL) is a generic variability modeling language and is composed of three interrelated models: base model, variability model, and resolution model. The base model can be defined in UML or any MOF based Domain Specific Language (DSL). Corresponding to the base model a variability model is defined. The variability model has a tree structure to specify variabilities. The resolution model specifies configurations of variabilities corresponding to a particular product. To support CVL, an Eclipse-based plugin CT-CVL is available [20]. In Figure A-3, rounded rectangles (e.g., *AGVHardware, SensorType, Connectivity)* represent *Choice* elements and a rectangle (e.g., *Sensor*) represents a *VClassifier* element whereas an ellipse represents a variable. Multiplicity inside the *VClassifier Sensor* (0..10) indicates that the number of instances of sensors can be between zero to 10 where for each instance one needs to configure sensor type and model. *Connectivity* and *SensorType* are *ChoiceVP* with group cardinality (1..1), which means only one option can be selected from given alternative options.

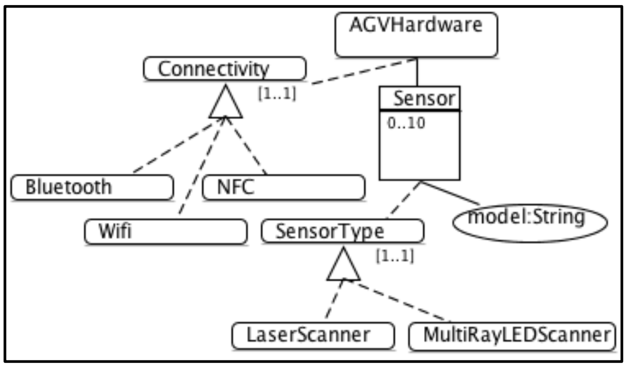


Figure A-3. An excerpt of CVL for AGV

SimPL is a UML based VMT, which provides notations and guidelines for modeling variabilities and commonalities of CPS product lines at the architecture and design level. To support SimPL, several modeling tools [21] (RSA, MagicDraw, and Papyrus) are available. It captures four types of VPs: Attribute-VP, Type-VP, Topology-VP, and Cardinality-VP. A SimPL product line model can be specified with a subset of UML structural elements and stereotypes defined in the SimPL profile. Constraints are specified in the Object Constraint Language (OCL). SimPL has two major views: SystemDesignView and VariabilityView. SystemDesignView is composed of HardwareView, SoftwareView, and AllocationView to represent hardware components, software components and their relationship. VariabilityView is for capturing and structuring variabilities using UML packages and template parameters. Stereotype «ConfigurationUnit» is applied on UML packages to group relevant variabilities. Variabilities are defined as template parameters of a package template and can trace back to hardware or software elements in the SystemDesignView. Figure A-4 presents an excerpt of the *HardwareView* of MHS, in which *AGV* is a hardware component composed of zero to many *Sensors*. *Sensor* can be of two types: *LaserScanner and MultiRayLEDScanner*. *AGV* has one Attribute-VP (*connectivity*) and one Cardinality-VP (*sensors*) denoting the number of instances of *Sensor*. For *Sensor*, two variabilities are specified: model (Attribute-VP) and type of sensor (Type-VP). *AGVConfigurationUnit* and *SensorsConfigurationUnit* are the template packages that are used to organize the variabilities corresponding to hardware component *AGV* and hardware *Sensor* respectively.

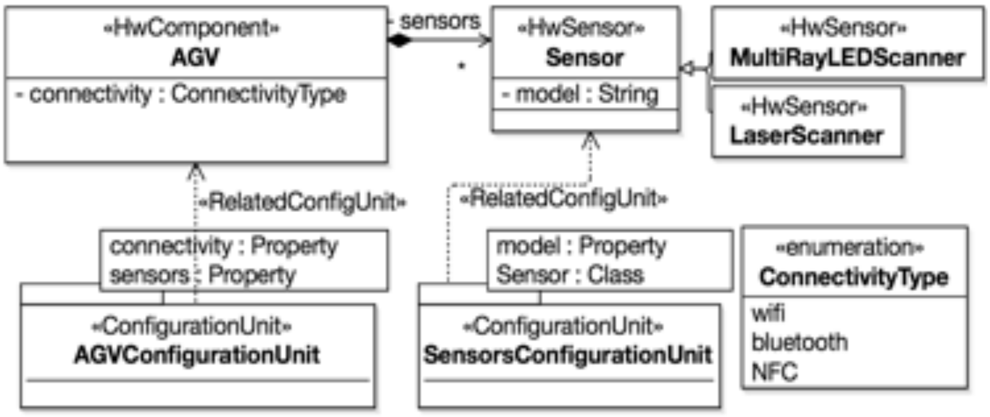


Figure A-4. An excerpt of SimPL for AGV

3.3 Procedure of the Study

Figure A-5 describes the procedure that we followed to conduct the study. First, we constructed a conceptual model for defining data types in mathematics and then we validated the data types with MARTE [22] and SysML [23], as these two standards are often used for modeling embedded systems and therefore can be used for modeling CPSs. In the third step, we defined a set of basic VP types (Section 4.1), based on the mathematical basic data types. We used basic data types for defining the basic VP types, as configuring a VP always requires assigning/selecting a value to/for a basic type variable. In the fourth step, we derived a set of modeling requirements (Section 5) based on knowledge collected from the literature and our experience of conducting industry-oriented research in the field of CPS PLE [4]. In the fifth step, we constructed a conceptual model for CPS, which is used to systematically derive the CPS-specific VP types (Step 6, more details in Section 4.2). In Step 7, we modeled the MHS case study with the selected VMTs, followed by the evaluation of the selected VMTs (Step 8, details in Section 0), based on the basic VP types, CPS-specific VP types, and the set of modeling requirements.

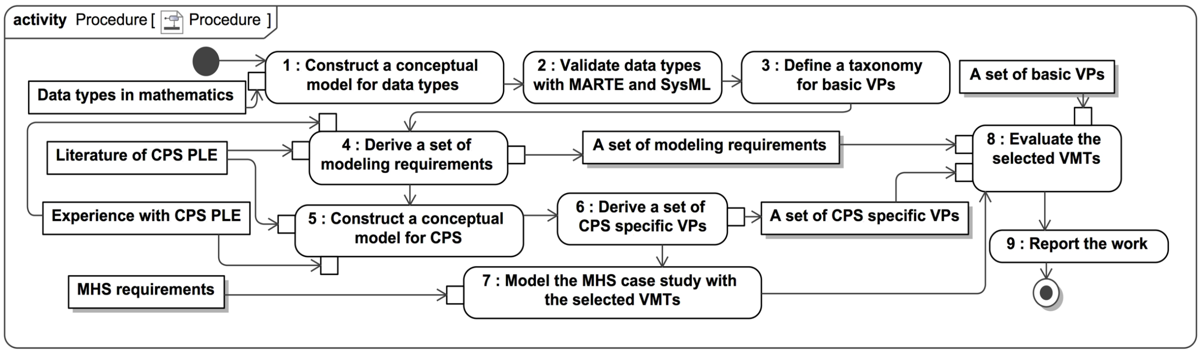

activity Procedure [Procedure]
Data types in mathematics
1 : Construct a conceptual model for data types
2 : Validate data types with MARTE and SysML
3 : Define a taxonomy for basic VPs
A set of basic VPs
Literature of CPS PLE
4 : Derive a set of modeling requirements
A set of modeling requirements
8 : Evaluate the selected VMTs
Experience with CPS PLE
5 : Construct a conceptual model for CPS
6 : Derive a set of CPS specific VPs
A set of CPS specific VPs
9 : Report the work
MHS requirements
7 : Model the MHS case study with the selected VMTs

Figure A-5. Procedure of the study

4 Basic and CPS-specific Variation Point Types

4.1 Basic Variation Point Types

Based on the basic data types in mathematics, we constructed a conceptual model to classify them, as shown in Figure A-6. A *Variable* can be a *VariationPoint* or a *Non-configurableVariable*, which represents the configurable and non-configurable variable in CPS PLE. Each *Variable* has a *Type*, which is classified into two categories: *Atomic* (taking a single value at a given point of time) and *Composite* (composed of more than one atomic type, where each atomic type variable takes exactly one value at a given point in time). Atomic types are further classified into *Quantitative* types (taking numeric values) and *Qualitative* types (taking non-numeric values). *Quantitative* types can be *Discrete* (taking countable values) or *Continuous* (taking uncountable values). *Integer* is the concrete *Discrete* type, whereas *Real* is the concrete *Continuous* type. *Qualitative* types are categorized into *String*, *Binary* and *Categorical* that is further classified into *Ordinal* and *Nominal*.

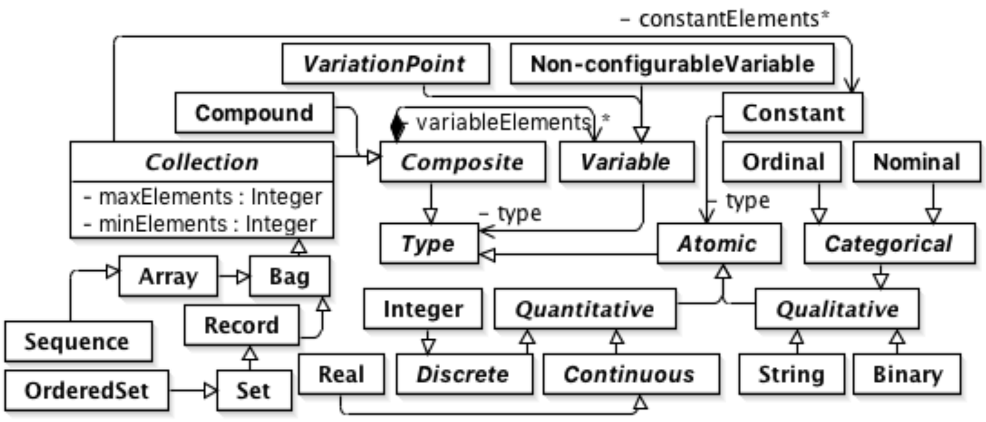


Figure A-6. Basic data types

A *Composite* data type combines several variables and/or constants, which is classified as: *Compound* and *Collection*. *Compound* takes only variables (e.g., complex numbers in SysML containing two variables realPart and imaginaryPart [23]) whereas *Collection* takes *Variables* and/or *Constants* (e.g., collection of colors). Attributes *minElements* and *maxElements* of *Collection* specify the minimum and maximum numbers of elements in a collection. As shown in Figure A-6, we have classified *Collection* into six types (i.e., *Bag*, *Array*, *Record*, *Set*, *OrderedSet* and *Sequence*) based on three properties: homogeneity, uniqueness and order. The homogeneity, uniqueness, and order properties of each collection type are specified as OCL constraints (Appendix A). Table A-2 summarizes the six types of *Collection* along with their properties.

Table A-2. Collection types

Collection	Hom.	Uni.	Ord.
Bag	No	No	No
Record	No	Yes	No
Set	Yes	Yes	No
OrderedSet	Yes	Yes	Yes
Array	Yes	No	No
Sequence	Yes	No	Yes

To validate the conceptual model of the basic data types, we mapped the data types defined in the MARTE Value Specification Language-VSL [22] and SysML [23] to the basic data types presented in Figure A-6. We used MARTE and SysML for validation because these two modeling languages can be used for modeling CPSs [24, 25]. During the validation, we do not include the extended data types provided in MARTE, as they are defined by extending the data types used in our mapping. In case of SysML we include all the data types. Results of the mapping are presented in Table A-3, from which one can see that each data type in MARTE and SysML has a correspondence in our basic data type classification, which suggests that our classification of the basic data types is complete.

Table A-3. Mapping MARTE and SysML data types to the basic data types

MARTE	SysML	Basic data types
Integer	Integer	Integer
UnlimitedNatural	UnlimitedNatural	Integer
Boolean	Boolean	Binary
String	String	String
Real	Real	Real
DateTime	Complex	Compound
EnumerationType	Enumeration	Ordinal/Nominal
	ControlValue	Nominal/Ordinal
IntervalType	UnitAndQuantityKind	Compound
TupleType		Compound
ChoiceType		Compound
CollectionType		Collection

In Figure A-7, we present a classification of basic VP types where one basic VP type is defined corresponding to each basic data type presented in Figure A-6. A *VariationPoint* can be a *CompositeVP* or an *AtomicVP*. An *AtomicVP* can come with any of the six concrete types: *StringVP*, *BinaryVP*, *NominalVP*, *OrdinalVP, IntegerVP, and RealVP* corresponding to *String*, *Binary*, *Nominal*, *Ordinal*, *Integer*, and *Real* respectively. A *CompositeVP* can be *CompoundVP* or *CollectionVP*, which are defined corresponding to *Compound* and *Collection* data types respectively. As shown in Figure A-7, a *CompositeVP* may have several *AtomicVP*s and/or *CompositeVP*s depending on the number of *variableElements* (Figure A-6) involved in the *Composite* data type. *CollectionVP* may have two additional *IntegerVP*(s), i.e., *lowerLimitVP* and *upperLimitVP* corresponding to the minimum and maximum numbers of the elements in the collection.

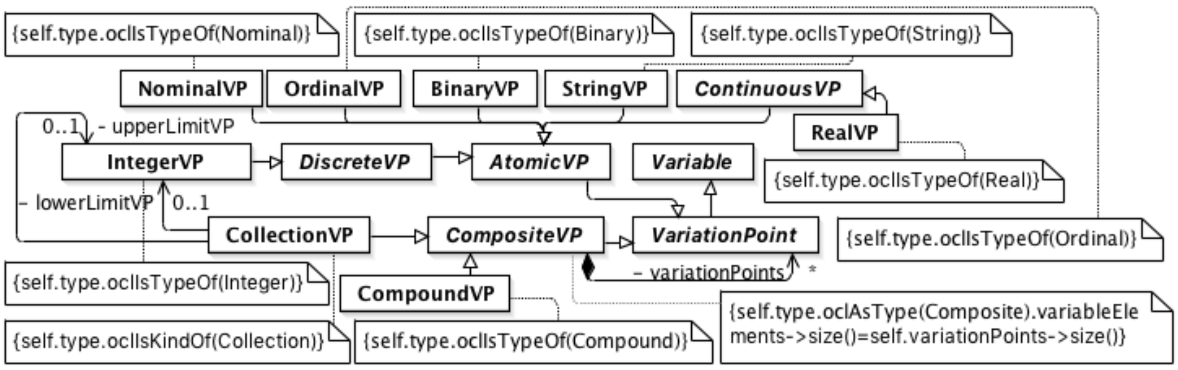

{self.type.oclIsTypeOf(Nominal)}
{self.type.oclIsTypeOf(Binary)}
{self.type.oclIsTypeOf(String)}
NominalVP
OrdinalVP
BinaryVP
StringVP
ContinuousVP
RealVP
0..1
- upperLimitVP
IntegerVP
DiscreteVP
AtomicVP
Variable
{self.type.oclIsTypeOf(Real)}
- lowerLimitVP
0..1
CollectionVP
CompositeVP
VariationPoint
{self.type.oclIsTypeOf(Ordinal)}
- variationPoints
*
{self.type.oclIsTypeOf(Integer)}
CompoundVP
{self.type.oclAsType(Composite).variableElements->size()=self.variationPoints->size()}
{self.type.oclIsKindOf(Collection)}
{self.type.oclIsTypeOf(Compound)}

Figure A-7. Classification of the basic VP types

4.2 CPS-specific Variation Point Types

In this section, first we present a conceptual model for CPS (Figure A-8), based on which we then derive a set of CPS-specific VP types (Table A-4). As shown in Figure A-8, a CPS can be defined as a set of physical components (e.g., human heart, engine), interfacing components (e.g., sensor, actuator, network), and cyber components (with deployed software), which are integrated together to accomplish a common goal.

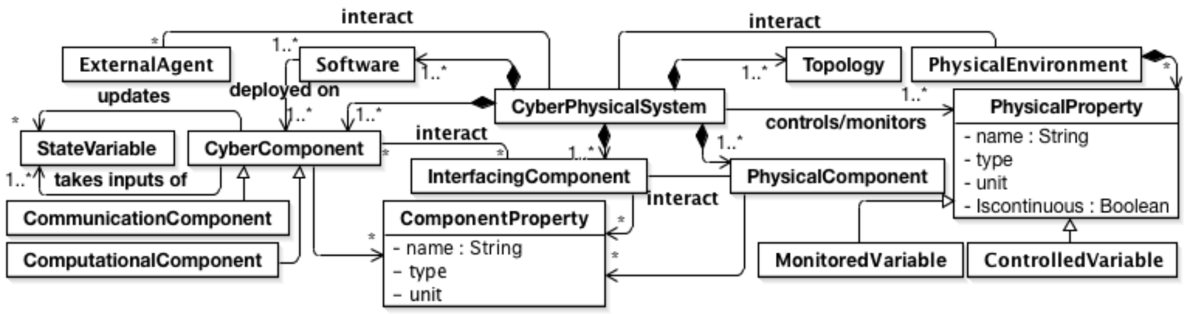


Figure A-8. A CPS conceptual model

A CPS can have one or more topologies, which define how various components are integrated. A CPS controls and monitors a set of physical properties. A *CyberComponent* can either be a *CommunicationComponent* or *ComputationalComponent*, which takes values of *StateVariables* as input and updates their values when needed. Each component in CPS has several component properties. CPS may interact with *PhysicalEnvironment* and *ExternalAgent*s (e.g., external systems). Both *PhysicalProperty* and *ComponentProperty* have attributes *name*, *type*, and *unit* to specify the name, type (e.g., descriptive, numeric, Boolean), and unit of a specific property. *PhysicalProperty* has an extra Boolean attribute *isContinuous* to specify either it is a continuous or a discrete type of property.

In Table A-4, the first column represents the CPS concepts used to derive CPS-specific VP types and the second column shows the derived CPS-specific VP types. The last column presents the basic VP type corresponding to a particular CPS-specific VP type.

PhysicalProperty and **ComponentProperty:** Descriptive-VP, DiscreteMeasurement-VP, ContinuousMeasurement-VP, BinaryChoice-VP, PropertyChoice-VP, MeasurementUnitChoice-VP, and MeasurementPrecision-VP are defined for physical properties and/or component properties of CPS. Descriptive-VP is a *StringVP*, which requires setting a value in order to configure it. It can be defined for a textual *ComponentProperty* such as ID of a sensor.

DiscreteMeasurement-VP and ContinuousMeasurement-VP are *IntegerVP* and *RealVP* respectively. Both these two types of VPs can be defined for numeric component properties (e.g., data transmission interval of a sensor) or physical properties (e.g., length and weight of a physical component) of CPS. BinaryChoice-VP is a *BinaryVP*, which can be defined for Boolean physical properties (e.g., the presence of a magnetic field) and component properties (e.g., whether a sensor keeps the events' log). PropertyChoice-VP is a *NominalVP* or an *OrdinalVP*, which requires selecting one value from a list of pre-defined values. For example, a *ComponentProperty* can be connectionType, which can be configured as wired, 3G, or Wi-Fi, which can be captured as a PropertyChoice-VP. MeasurementUnitChoice-VP is an *OrdinalVP*, which is derived from the *unit* of *PhysicalProperty* and *ComponentProperty*. For example, one can select meter, centimeter or millimeter as a unit for length (a *PhysicalProperty*). MeasurementPrecision-VP is a *RealVP*, which is related to the degree of measurement precision for a *PhysicalProperty* or *ComponentProperty*.

Table A-4. CPS-specific VP types

CPS Concept	CPS-Specific VP Type	Basic VP Type
CP	Descriptive-VP	StringVP
CP, PP	DiscreteMeasurement-VP	IntegerVP
CP, PP	ContinuousMeasurement-VP	RealVP
CP, PP	BinaryChoice-VP	BinaryVP
CP, PP	PropertyChoice-VP	NominalVP/OrdinalVP
CP, PP	MeasurementUnitChoice-VP	OrdinalVP
CP, PP	MeasurementPrecision-VP	RealVP
CP, PP, COM	Multipart/Compound-VP	CompoundVP
COM	ComponentCardinality-VP	IntegerVP
COM	ComponentCollectionBoundary-VP	IntegerVP
COM	ComponentChoice-VP	NominalVP/OrdinalVP
COM	ComponentSelection-VP	CollectionVP
Topology	TopologyChoice-VP	NominalVP
Deployment	AllocationChoice-VP	NominalVP
Interact	InteractionChoice-VP	NominalVP
Constraint	ConstraintSelection-VP	CollectionVP

*CP=ComponentProperty, PP =PhysicalProperty, COM=Physical, Interfacing, or Physical Component

Component: ComponentCardinality-VP, ComponentCollectionBoundary-VP, ComponentChoice-VP, and ComponentSelection-VP are derived from the different types of CPS components: *CyberComponent*, *InterfacingComponent*, *PhysicalComponent*. ComponentCardinality-VP is an *IntegerVP*, which is related to varying number of instances of a CPS component (e.g., number of temperature sensors). ComponentCollectionBoundary-VP is an *IntegerVP*, which is related to the upper limit and/or the lower limit of a collection of CPS components. For example, the maximum and minimum numbers of sensors supported by a controller. ComponentChoice-VP is a *NominalVP*/*OrdinalVP*, which is about selecting a particular type of CPS component such as selecting a speedometer sensor from several speedometers with various specifications. ComponentSelection-VP is a *CollectionVP*, which is about selecting a subset of CPS components from a collection of CPS components such as selecting sensors for a product from available sensors.

Multipart/Compound-VP is a *CompoundVP*, which can be specified for a *PhysicalProperty*, *ComponentProperty*, or a component (Physical, Cyber, or Interfacing) that requires configuring several constituent VPs involved in it. As in the domain of CPS, it is common that different properties do not give complete meaning unless they are combined together. For example, length is a *PhysicalProperty*, which is meaningless without a unit. Hence, we need a Compound-VP type, which involves two VPs including length and its unit. A Compound-VP can also be defined for a component (e.g., sensor), which contains several other VPs defined for its properties.

Topology: TopologyChoice-VP is a *NominalVP*, which is related to selecting a topology from several alternatives. For example, how *CyberComponent* (e.g., controller) is connected with *InterfacingComponent*s (e.g., sensors and actuators).

Deployment: AllocationChoice-VP is a *NominalVP*, which is about the deployment of software on a *CyberComponent* (e.g., controller). For example, the same version of software can be deployed on different controllers or different versions of software can be deployed on the same controller.

Interaction: InteractionChoice-VP is a *NominalVP*, which is about the interaction (presented as association named interact in Figure A-8), of two CPS components (e.g., *CyberComponent* and *InterfacingComponent*) or interaction of CPS with an external agent, which can be for example an external system.

Constraint: ConstraintSelection-VP is a *CollectionVP*, which is about selecting a subset of constraints in order to support the configuration of a specific product, from a set of constraints defined for the corresponding CPS product line.

5 Modeling Requirements

In addition to capturing different types of VPs, a VMT should also accommodate some modeling requirements to enable automation of configuring CPS products. These requirements (Table A-5) are derived from the literature and our experience of working with industry [4].

Table A-5. Modeling requirements

ID	Name	Description
R_1	VP binding time	Support different binding times for a VP (e.g., pre-deployment, deployment, and post-deployment phases).
R_2	Linkage between VP and the base	Provide a mechanism to relate a VP to the corresponding base model element.
R_3	Separation of Concerns	Provide a mechanism to realize the principle of separation of concerns to enable multi-staged and cross-disciplinary configuration of CPS.
R_4	Variability dependency	Capture dependencies between a VP and a variant, two VPs, and two variants.
R_5	Ordering	Specify constraints on the order of configuration steps.
R_6	Inference	Specify constraints that can be used to configure VPs automatically.
R_7	Conformance	Specify conformance rules for ensuring the correctness of configuration data.
R_8	Consistency	Specify consistency rules for checking the consistency of the configuration data and variability models.
R_9	Multidisciplinary	Model *Software*, *PhysicalComponent*, *InterfacingComponent*, *CyberComponent*, and *PhysicalEnvironment* elements of CPS.

In Table A-5, R_1 is related to support different binding times of a VP, as a VP can be configured at three different phases [26]: the pre-deployment phase, the deployment phase and the post-deployment phase. Requirements R_2 focuses on a traceability mechanism to link the variability model and its base whereas R_3 is related to realizing the separation of concerns principle in the product line model. R_4-R_8 are relevant to different types constraints that a VMT should be able to capture for enabling automation of the configuration process in CPS PLE [3]. In [3], a constraint classification was presented and we extended it by adding two more categories: inference and conformance. These constraints are needed to facilitate different functionalities of an interactive, multi-step and multi-staged configuration solution, such as consistency checking, decision inferences. R_9 is related to modeling different types of configurable elements of CPSs.

6 Evaluation

The purpose of the evaluation is to compare the selected four VMTs with the aim to help modelers to select an appropriate VMT or propose a new one if necessary for CPS PLE, which can capture different types of VPs (Section 4) and meet the modeling requirements (Section 5). Corresponding to this goal, we pose the following research questions: **RQ1**: To what extent can each selected VMT capture the basic VPs? **RQ2**: To what extent can each selected VMT capture the CPS-specific VPs? **RQ3**: To what extent does a selected VMT comply with the modeling requirements? We answer RQ1, RQ2 and RQ3 in Section 6.1, Section 6.2, and Section 6.3, respectively.

6.1 Evaluation Based on Basic VP Types (RQ1)

To answer RQ1, we evaluate the selected VMTs based on the basic VP types. In Table A-6, the first column represents the basic VP type and the second column indicates if a basic VP type is required by the MHS case study, whereas columns 3-6 show how each selected VMT supports each basic VP type.

Table A-6. Evaluation based on the basic VP types (RQ1)

Basic VP Type	MHS	VMT			
		FM	CBFM	SimPL	CVL
IntegerVP	Yes	No	One At/F, G & F Cardinality	Attribute-VP, Cardinality-VP	Multiplicity, ParametricVP
RealVP	Yes	No	One At/F	Attribute-VP	ParametricVP
StringVP	Yes	No	One At/F	Attribute-VP	ParametricVP
BinaryVP	Yes	OF, Alt. F	One At/F, OF, Alt. G, F-Cardinality	Attribute-VP, Cardinality-VP, Type-VP, Topology-VP	ChoiceVP (ObjectSubstitution, SlotAssignment, ObjectExistence, SlotValueExistence, LinkExistence), Multiplicity, ParametricSlotAssignment
NominalVP	Yes	Alt. G	Alt. G	Attribute-VP, Type-VP, Topology-VP	Group of SlotAssignment (i.e., ChoiceVP) with group Multiplicity (1,1), ParametricObjectSubstitution (i.e., ParametricVP).
OrdinalVP	Yes	Alt. G	Alt. G		
CompoundVP	Yes	No	No	Configuration Unit	CompositeVP, VClassifier with several Repeatable-VP(s).
CollectionVP	Yes	No	Alt. G, OR G	Cardinality-VP	VClassifier with configurable Multiplicity, group of SlotAssignment (i.e., ChoiceVP).

*F=feature, OF=optional feature, G=group, At=attribute, Alt=Alternative, /= per, &= and

As one can see from Table A-6, modeling the MHS case study requires all the basic VP types. However, FM supports only three out of eight basic VP types: *BinaryVP*, *NominalVP* and *OrdinalVP*. Optional feature and alternative-group with two features of FM map to *BinaryVP*s. In FM, alternative-group corresponds to *NominalVP*s and *OrdinalVP*s, but FM does not differentiate *NominalVP* from *OrdinalVP*. CBFM provides support for all the basic VP types except for *CompoundVP*. Corresponding to *RealVP*s and *StringVP*s, CBFM provides attributes (one attribute per feature) of Real and String respectively. However, for *IntegerVP*s, it offers feature and group cardinalities together with Integer attributes. For *BinaryVP*, CBFM has optional features, alternative-groups, feature cardinalities (0..1), and Boolean attributes. Similar to FM, CBFM also provides alternative-groups, which map to *NominalVP*s and *OrdinalVP*s and CBFM does not differentiate these two types. For *CollectionVP*, CBFM provides alternative-groups and or-groups.

Both SimPL and CVL support all the basic VP types. In SimPL, Attribute-VP defined with Real and String attributes map to *RealVP*s and *StringVP*s. *IntegerVP*s can map to Attribute-VPs defined on Integer attributes or Cardinality-VP. To support *BinaryVP*, SimPL provides Attribute-VP defined on attributes of the binary type, Cardinality-VP with two options, Type-VP with two types, and Topology-VP with two topologies. Cardinality-VP, Type-VP, and Topology-VP offered by SimPL can be mapped to *NominalVP*s and *OrdinalVP*s. SimPL does not differentiate *NominalVP* and *OrdinalVP*. To support *CompoundVP*, SimPL defines «ConfigurationUnit», which can be applied on packages, to organize a set of relevant VPs. In SimPL, *CollectionVP* corresponds to Cardinality-VP.

To support *RealVP* and *StringVP*, CVL provides ParametricVP. For *IntegerVP* it provides ParametricVP and cardinalities. For *BinaryVP*, CVL has different types of ChoiceVPs (i.e., ObjectSubstitution, SlotAssignment, ObjectExistence, SlotValueExistence, and LinkExistence) along with multiplicity and ParametricSlotAssignment (i.e., ParametricVP). In CVL, both *NominalVP*s and *OrdinalVP*s can be mapped to SlotAssignments (i.e., ChoiceVP) with group multiplicity (1..1) or ParametricObjectSubstitution (i.e., ParametricVP). Similar to all the other VMTs, CVL does not differentiate *NominalVP* and *OrdinalVP*. In CVL, *CompoundVP* maps to CompositeVP and a VClassifier with several RepeatableVP(s) can also be used to model *CompoundVP*s. For *CollectionVP*, CVL has VClassifier with the multiplicity other than (1..1) and a group of SlotAssignment (i.e., ChoiceVP).

To summarize, both SimPL and CVL support all the basic VP types whereas FM and CBFM provide partial support. None of the selected four VMTs differentiate NominalVP and OrdinalVP.

6.2 Evaluation Based on the CPS-Specific VP Types (RQ2)

To answer RQ2, we evaluate the selected four VMTs based on the CPS-specific VP types (Section 4.2) and VPs modeled for the MHS case study. In Table A-7, the first column represents the CPS-specific VP types and the second column indicates if a particular CPS-specific VP type is required by the MHS case study. Columns 3-6 are related to the four VMTs to signify if they support a particular CPS-specific basic VP type. The seventh column shows the number of VPs in the MHS case study corresponding to a particular CPS-specific VP type, whereas columns 8-11 show the number of VPs modeled using the four VMTs.

As one can see from Table A-7, our case study (MHS) contains VPs corresponding to all the CPS-specific VP types. FM does not cater majority of the CPS-specific VP types and only supports fully or partially three out of 16 CPS-specific VP types: BinaryChoice-VP, PropertyChooice-VP, and ComponentChoice-VP.

CBFM supports six of 16 CPS-specific VP types: ComponentCardinality-VP, ComponentCollectionBoundary-VP, MeasurementPrecision-VP, PropertyChoice-VP, ComponentChoice-VP, and ComponentSelection-VP. It provides partial support for three CPS-specific VP types (i.e., Descriptive-VP, DiscreteMeasurement-VP, and ContinuousMeasurement-VP) because CBFM allows adding only one attribute for each feature. BinaryChoice-VP is also partially supported, as it can be captured using optional feature or cardinality but CBFM does not allows adding Boolean attribute. The remaining six CPS-specific VP types are not supported by CBFM.

Both SimPL and CVL support Descriptive-VP, DiscreteMeasurement-VP, ContinuousMeasurement-VP, ComponentSelection-VP, ComponentCardinality-VP, ComponentCollectionBoundary-VP, BinaryChoice-VP, MeasurementPrecision-VP, MeasurementUnitChoice-VP, PropertyChoice-VP, ComponentChoice-VP, and Compound-VP. SimPL also supports TopologyChoice-VPs, which cannot be captured using CVL. The remaining three CPS-specific VP types (i.e., AllocationChoice-VP, InteractionChoice-VP, and ConstraintSelection-VP) are not catered by either SimPL or CVL.

As shown in Table A-7, none of the selected VMTs supports all the CPS-specific VP types. SimPL supports 81%, FM supports only 15%, CVL caters 75%, and CBFM covers 50% of the total CPS-specific VP types. Using SimPL and CVL we were able to model 96% and 86%, whereas with FM and CBFM, we could model only 19% and 55% of total VPs in our case study.

Table A-7. Evaluation of VMTs based on the CPS-specific VP types and VPs (RQ2)

CPS-Specific VP Type	VP Types Coverage					VP Coverage				
	MHS	**FM**	**CBFM**	**SimPL**	**CVL**	**MHS**	**FM**	**CBFM**	**SimPL**	**CVL**
Descriptive-VP	Yes	No	Partial	Yes	Yes	34	0	4	34	34
DiscreteMeasurement-VP	Yes	No	Partial	Yes	Yes	23	0	5	23	23
ContinuousMeasurement-VP	Yes	No	Partial	Yes	Yes	51	0	18	51	51
ComponentCardinality-VP	Yes	No	Yes	Yes	Yes	42	0	42	42	42
ComponentCollectionBoundary-VP	Yes	No	Yes	Yes	Yes	42	0	42	42	42
MeasurementPrecision-VP	Yes	No	Yes	Yes	Yes	2	0	2	2	2
BinaryChoice-VP	Yes	Partial	Partial	Yes	Yes	3	0	0	3	3
PropertyChoice-VP	Yes	Yes	Yes	Yes	Yes	82	82	82	82	82
ComponentChoice-VP	Yes	Yes	Yes	Yes	Yes	12	12	12	12	12
TopologyChoice-VP	Yes	No	No	Yes	No	9	0	0	9	0
AllocationChoice-VP	Yes	No	No	No	No	3	0	0	0	0
InteractionChoice-VP	Yes	No	No	No	No	15	0	0	0	0
MeasurementUnitChoice-VP	Yes	No	No	Yes	Yes	59	0	18	59	59
ConstraintSelection-VP	Yes	No	No	No	No	1	0	0	0	0
ComponentSelection-VP	Yes	No	Yes	Yes	Yes	42	0	42	42	42
Multipart/Compound-VP	Yes	No	No	Yes	Yes	64	0	0	64	26
Total (count)	**16**	**2.5**	**8**	**13**	**12**	**484**	**94**	**267**	**465**	**418**
Coverage (%)	**100%**	**15%**	**50%**	**81%**	**75%**	**-**	**19%**	**55%**	**96%**	**86%**

6.3 Evaluation Based on the Modeling Requirements (RQ3)

Table A-8 summarizes the results of our evaluation of the four VMTs in terms of modeling requirements (Section 5) with MHS. In Table A-8, the first two columns are used to identify the requirements and the third column indicates if a requirement is required by MHS. Columns 4-7 signify if the VMTs support a particular requirement.

None of the selected VMTs except for CVL allows specifying the binding time (R_1) of a VP to enable its configuration in different phases. CVL and SimPL support linking a VP to the corresponding base model element explicitly (R_2), which is however not supported by FM and CBFM, as they do not have separate base models. FM and CBFM do not support the separation of concerns (R_3) and CVL supports partially as it models variabilities separately from the base model. SimPL supports R_3 as it provides hardware, software and allocation views in addition to the variability view. For MHS, we captured all the four views defined in SimPL. But, it still requires a view for specifying environment elements and corresponding VPs.

Table A-8. Results for the evaluation of the VMTs based on the modeling requirements (RQ3)

ID	Name	MHS	FM	CBFM	CVL	SimPL
R_1	VP binding times	Yes	No	No	Yes	No
R_2	Linkage between VP and the base	Yes	No	No	Yes	Yes
R_3	Separation of Concerns	Yes	No	No	Partial	Yes
R_4	Variability dependencies	Yes	Partial	Partial	Partial	Yes
R_5	Ordering	Yes	No	No	Depends on base modeling language	Yes
R_6	Inference	Yes	No	No		Yes
R_7	Conformance	Yes	No	No		Yes
R_8	Consistency	Yes	No	No		Yes
R_9	Multidisciplinary	Yes	No	No		Partial

R_4-R_8 are related to capturing different types of constraints to enable automation in CPS PLE. FM and CBFM provide partial support for capturing variability dependencies such as requires and excludes, but they are unable to capture other complex constraints such as consistency rules. In the case of CVL, it uses the Basic Constraint Language [8] for capturing simple propositional and arithmetic constraints but it is unable to capture all the types of constraints discussed in Section 5. If the base model is modeled in UML, then OCL can be integrated with CVL, thereby allowing the specification of all the types of constraints. SimPL is based on UML and OCL, which makes it possible to capture all the types of constraints.

MHS is a multidisciplinary system, which contains *Software*, *CyberComponent*, and different types of *PhysicalComponent* and *InterfacingComponent* interacting with *PhysicalEnvironment* but none of the selected VMTs explicitly model these multidisciplinary elements of CPS (R_9). SimPL supports all, except for *PhysicalEnvironment* elements. In case of CVL, it depends on the DSL used for modeling the base model, which may or may not have the capability of modeling different elements of CPS.

7 Threats to validity

One threat to validity of our study is the selection of the VMTs. Since it is not practically feasible to evaluate all existing VMTs, we therefore selected four representative VMTs. Another threat to validity is the completeness of the basic and CPS-specific VP types and modeling requirements. Note that our approach for deriving the basic VP types is systematic, which to certain extent ensures their completeness. In addition, we validated them using SysML and MARTE, which are two existing standards often used for embedded system modeling. We derived CPS-specific VP types based on thorough domain analyses and our experience in working with industry. We also verified that the MHS case study covers all the CPS-specific VP types.

8 Conclusion

In this paper, we present a set of basic and CPS-specific VP types that need to be supported by a VMT in the context of CPS PLE. Moreover, we present a set of modeling requirements, which need to be catered to enable the automation of configuration in CPS PLE. Based on the proposed basic and CPS-specific VP types and modeling requirements, we evaluated four VMTs: feature model, cardinality based feature model, CVL, and SimPL, with a real-world case study. Results of our evaluation show that the selected four VMTs cannot capture all the VP types and none of the four VMTs meets all the requirements. This necessitates the extension of an existing technique or proposal of a new one to facilitate CPS PLE. The proposed VP types and modeling requirements can be used as evaluation criteria to select a suitable VMT or develop a new one if necessary.

Acknowledgement

This work was supported by the Zen-Configurator project funded by the Research Council of Norway (grant no. 240024/F20) under the category of Young Research Talents of the FRIPO funding scheme. Tao Yue and Shaukat Ali are also supported by the EU Horizon 2020 project U-Test (http://www. u-test.eu/) (grant no. 645463), the RFF Hovedstaden funded MBE-CR (grant no. 239063) project, the Research Council of Norway funded MBT4CPS (grant no. 240013/O70) project, and the Research Council of Norway funded Certus SFI (grant no. 203461/O30).

References

1. Cyber-Physical Systems (CPSs). Available from: http://cyberphysicalsystems.org/.
2. Rawat, D.B., J.J. Rodrigues, and I. Stojmenovic, Cyber-Physical Systems: From Theory to Practice. 2015: CRC Press.
3. Nie, K., et al. Constraints: the core of supporting automated product configuration of cyber-physical systems. in Proceeding of International Conference on Model-Driven Engineering Languages and Systems (MODELS). 2013. Springer.
4. Yue, T., S. Ali, and B. Selic. Cyber-physical system product line engineering: comprehensive domain analysis and experience report. in Proceedings of the 19th International Conference on Software Product Line. 2015. ACM.
5. Kang, K., Cohen, Sholom., Hess, James., Novak, William., & Peterson, A., Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021), in Secondary Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021), Secondary Kang, K., Cohen, Sholom., Hess, James., Novak, William., & Peterson, A., Editor. 1990. RN. Available From:
6. Czarnecki, K., S. Helsen, and U. Eisenecker, Staged configuration using feature models, in Software Product Lines. 2004, Springer. p. 266-283.
7. Behjati, R., et al., SimPL: a product-line modeling methodology for families of integrated control systems. Information and Software Technology, 2013.
8. Haugen, O., Common Variability Language (CVL). OMG Revised Submission, 2012.
9. Berger, T., et al. A survey of variability modeling in industrial practice. in Proceedings of 7th International Workshop on Variability Modelling of Software intensive Systems. 2013. ACM.
10. Galster, M., et al., Variability in software systems-A systematic literature review. IEEE Transactions on Software Engineering, , 2014. **40**(3): p. 282-306.
11. Chen, L., M. Ali Babar, and N. Ali, Variability management in software product lines: A systematic review, in 13th International Software Product Line Conference. 2009. p. 81-90.
12. Arrieta, A., G. Sagardui, and L. Etxeberria, A comparative on variability modelling and management approach in simulink for embedded systems. V Jornadas de Computación Empotrada, ser. JCE, 2014.
13. Djebbi, O. and C. Salinesi. Criteria for comparing requirements variability modeling notations for product lines. in 4th International Workshop on Comparative Evaluation in Requirements Engineering. 2006. IEEE.
14. Eichelberger, H. and K. Schmid, A systematic analysis of textual variability modeling languages, in Software Product Line Conference. 2013, ACM. p. 12-21.
15. Sinnema, M. and S. Deelstra, Classifying variability modeling techniques. Information and Software Technology, 2007. **49**(7): p. 717-739.
16. Czarnecki, K., et al. Cool features and tough decisions: a comparison of variability modeling approaches. in 6th international workshop on variability modeling of software intensive systems. 2012. ACM.
17. Berger, T., et al., Variability modeling in the real: a perspective from the operating systems domain, in International conference on Automated software engineering. 2010, ACM. p. 73-82.
18. www.zen-tools.com/SAM2016.html. Available from: www.zen-tools.com/SAM2016.html.
19. http://www.pure-systems.com/. Available from: http://www.pure-systems.com.
20. http://modelbased.net/tools/ct-cvl/. Available from: http://modelbased.net/tools/ct-cvl/.

21. Safdar, S.A., M.Z. Iqbal, and M.U. Khan, Empirical Evaluation of UML Modeling Tools–A Controlled Experiment, in European Conference on Modeling Foundations and Applications. 2015, Springer: Italy. p. 33-44.
22. The UML MARTE profile, http://www.omgmarte.org/.
23. OMG, Systems Modeling Language (SysML) v1.4, http://sysml.org/. 2015.
24. Selic, B. and S. Gérard, Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. 2013: Elsevier.
25. Derler, P., E.A. Lee, and A.S. Vincentelli, Modeling Cyber–Physical Systems. Proceedings of the IEEE Special issue on CPS, 2012. **100**(1): p. 13-28.
26. Murguzur, A., et al. Context variability modeling for runtime configuration of service-based dynamic software product lines. in Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools. 2014. ACM.

Appendix A: OCL Constraints

Homogeneity: context Array, Set (Sequence, OrderedSet)(**self**.constantElements->size()=0 **and self**.variableElements->select(a|a.**oclIsKindOf**(Collection))->size()=0 **and self**.variableElements->forAll(a,b| a.type=b.type))**or** (**self**.variableElements->size()=0 **and self**.constantElements->forAll(a,b| a.type=b.type)) **or** (**self**.constantElements->size()=0 **and self**.variableElements->size()=**self**.variableElements->select(a:Variable|a.type.**oclIs KindOf**(Collection))->size() **and self**.variableElements->forAll(v1, v2|(v1.type.**oclAsType**(Collection).constant Elements->size()=0 **and** v1.type.**oclAsType**(Collection).variableElements->forAll(v3:Variable | v3.type = v2.type.**oclAsType**(Collection).variableElements->asSequence()->first().type)) **or** (v1.type.**oclAsType**(Collection).variableElements->size()=0 **and** v1.type.**oclAs Type**(Collection).constantElements->forAll(v3:Constant| v3.type=v2.type.**oclAsType** (Collection).constantElements->asSequence()->first().type))))	**Uniqueness: context** Record (Set, OrderedSet) **self**.variableElements->select (**self**.variableElements ->forAll(a,b| a=b))->isEmpty() **and self**.constant Elements->select (**self**.constantElements->forAll(a,b| a=b))->isEmpty()
	Order: context Sequence **self**.variableElements->asSet()->size() >1 **implies self**.variableElements->asSequence()->reverse() <> **self**.variableElements->asSequence() **and self**.constantElements->asSet()->size() >1 **implies self**.constantElements->asSequence()->reverse() <> **self**.constantElements->asSequence() **context** OrderedSet **self**.variableElements->asOrderedSet()->reverse() <> **self**.variableElements->asOrderedSet() **and self**.constantElements->asOrderedSet()->reverse() <> **self**.constantElements->asOrderedSet()

Paper B

A Framework for Automated Multi-Stage and Multi-Step Product Configuration of Cyber-Physical Systems

Safdar Aqeel Safdar, Hong Lu, Tao Yue, Shaukat Ali, Kunming Nie

Published in the Journal of Software and Systems Modeling (SoSyM), 2020.

Abstract

Product Line Engineering (PLE) has been employed to large-scale Cyber-Physical Systems (CPSs) to provide customization based on users' needs. A PLE methodology can be characterized by its support for capturing and managing the abstractions as commonalities and variabilities and the automation of the configuration process for effective selection and customization of reusable artifacts. The automation of a configuration process heavily relies on the captured abstractions and formally specified constraints using a well-defined modeling methodology. Based on the results of our previous work and a thorough literature review, in this paper, we propose a conceptual framework to support multi-stage and multi-step automated product configuration of CPSs, including a comprehensive classification of constraints and a list of automated functionalities of a CPS configuration solution. Such a framework can serve as a guide for researchers and practitioners to evaluate an existing CPS PLE solution or devise a novel CPS PLE solution. To validate the framework, we conducted three real-world case studies. Results show that the framework fulfills all the requirements of the case studies in terms of capturing and managing variabilities and constraints. Results of the literature review indicate that the framework covers all the functionalities concerned by the literature, suggesting that the framework is complete for enabling the maximum automation of configuration in CPS PLE.

Keywords: Cyber-Physical Systems; Product Line Engineering; Automated Configuration; Multi-Stage and Multi-Step Configuration Process; Constraint Classification; Variability Modeling; Real-World Case Studies

1 Introduction

Cyber-Physical Systems (CPSs) are highly connected large-scale systems that combine digital cyber technologies with physical processes where embedded computers and networks monitor and control physical processes using sensors and actuators [1-5]. These systems are increasingly becoming an essential part of daily life, which are applied in diverse domains such as communication, logistics, and healthcare [6, 7]. To cater different needs of users, CPSs require customizations and thus, many CPS producers opt for Product Line Engineering (PLE) methodologies [6, 8]. PLE methodologies enhance the reusability and consequently are expected to improve the overall quality of produced CPSs and the productivity of the development process, and speed up time-to-market [9-12]. A PLE methodology can be characterized by its support for capturing and managing the abstractions in the domain engineering phase, and automation of product configuration for effective selection and customization of reusable artifacts in the application engineering phase.

A systematic domain analysis of the CPS PLE industrial practices is reported in [13], which highlights key characteristics of CPS PLE: 1) CPSs are large-scale, heterogeneous, and hierarchical systems; (2) the hardware topology can vary from one product to another; (3) the generic software code base might be instantiated and configured differently for each product, mostly based on the hardware topology; and (4) there are many dependencies among configurable parameters, particularly across the software code base and the hardware topology. Several challenges in CPS PLE were also reported in [13] such as lacking automation and guidance for product configuration and expensive debugging of configuration data. In general, cost-effectively supporting CPS PLE, especially enabling automation of product configuration, is an industrial challenge.

In CPS PLE, a large number of reusable components (e.g., software, hardware, or network components) are typically configured by different domain experts in different phases of the product development lifecycle, working at different organizations or different departments of the same organization. This demands following a particular configuration process comprising a set of configuration tasks performed sequentially or concurrently, which allows users to configure a CPS incrementally in a multi-stage and multi-step manner [14], in the sense that experts from various domains (e.g., hardware and software engineers) configure a CPS at different stages of a CPS development process and different steps within a stage. Moreover, the correctness of product configuration needs to be ensured with well-formedness, conformance, and consistency checking. Thus, an automated configuration solution with at least these correctness checking functionalities is highly appreciated. Such a solution heavily relies on a large number of constraints that should be formally specified with a well-defined constraint specification language (e.g., Object Constraint Language-OCL [15]) to facilitate, e.g., inferring configuration decisions automatically and optimization of configuration orders according to user preferences.

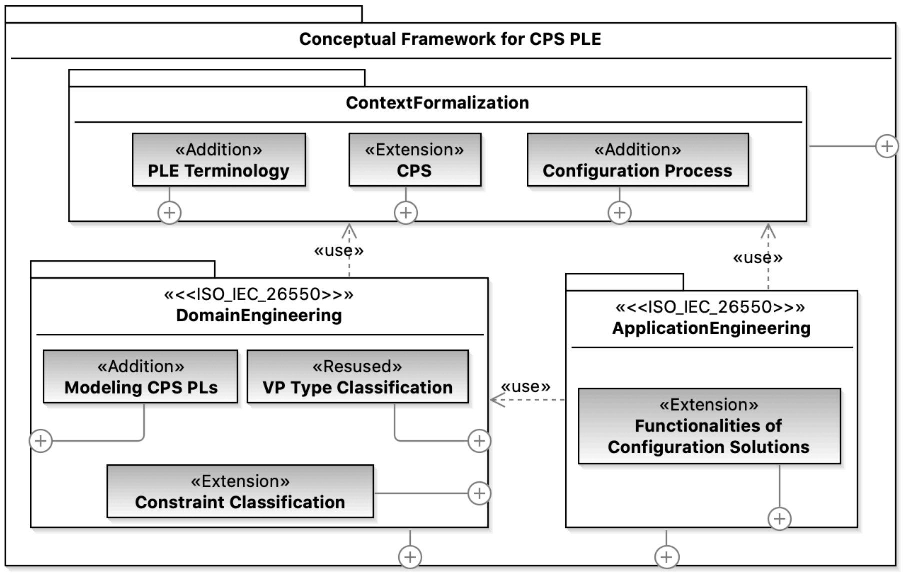

Conceptual Framework for CPS PLE
ContextFormalization
«Addition» PLE Terminology
«Extension» CPS
«Addition» Configuration Process
«use»
«use»
«<<ISO_IEC_26550>>» DomainEngineering
«Addition» Modeling CPS PLs
«Resused» VP Type Classification
«Extension» Constraint Classification
«use»
«<<ISO_IEC_26550>>» ApplicationEngineering
«Extension» Functionalities of Configuration Solutions

Figure B-1. An overview of the proposed conceptual framework for CPS PLE[2]

In our previous work [6], we proposed a classification of constraints for supporting CPS PLE, where we gave a textual description of four types of constraints and discussed how these constraints can facilitate five types of automation of configuration (i.e., collaborative configuration, decision inference, reverting decision, decision ordering and consistency checking). In another work of ours [16], we proposed a classification of variation point (VP) types to capture variabilities of CPS Product Lines (PLs). In this paper, we extend the above-mentioned two works [6, 16] and propose a complete conceptual framework to support multi-stage and multi-step configuration of CPSs based on knowledge collected from the existing literature and

[2] Note: All the conceptual models are available at http://zen-tools.com/Framework/ConceptualFramework.html.

our experience of conducting industry-oriented research in the field of CPS PLE [13]. To the best of our knowledge, the proposed framework is the first complete framework that covers activities in the domain engineering (i.e., capturing constraints and abstractions in form of commonalities and variabilities) and application engineering (i.e., configuration process and tool to enable automation of configuration) of CPS PLs. The framework does not only clarify the problem of supporting multi-stage and multi-step automated configuration of CPSs but also serves as a guide to researchers and practitioners to evaluate an existing CPS-specific PLE solution or devise a new one. Figure B-1 provides an overview of the framework, where we use three stereotypes «*Addition*», «*Extension*», and «*Reused*» to differentiate this work from our previous works [6, 16].

The key contributions of the paper are as follow:

- *ContextFormalization* of the framework has three conceptual models and a set of OCL constraints used to formalize PLE based on the PLE ISO/IEC standard for Product Line Engineering and Management [11], CPSs, and multi-stage and multi-step configuration process.
- *DomainEngineering* is about supporting domain engineering of CPS PLs where we formalize concepts related to modeling of CPS PLs such as models, model elements, and views. We also present the classifications of VP types and constraints. The VP types are from our previous work [16], whereas for the constraint classification we extend our previously proposed constraint classification [6] by adding four new types of constraints.
- *ApplicationEngineering* is for supporting application engineering of CPS PLs where we present 14 possible functionalities of an automated configuration solution and provide their formal definitions. Five of the 14 functionalities were presented in [6].
- We evaluate the framework by performing three representative case studies of CPS PLs and an extensive literature review. With the three case studies, we evaluate the VP types, constraint types, and views. The functionalities and configuration process are validated using 11 configuration tools and existing literature reporting configuration automation techniques.

Evaluation results show that the framework has all the necessary VP, constraint, and view types required to capture and manage the variabilities and constraints of selected case studies. In total, three case studies have 2161 VPs, 3943 constraints, and 40 views that can be modeled using the framework. Furthermore, the results for evaluating the functionalities based on 11 configuration tools and literature on automation of configuration show 92% coverage, which means 13 out of 14 functionalities are covered by at least one of the existing tools or techniques in the literature. This demonstrates that the framework covers all the necessary VP types to capture the variabilities of CPSs, constraint types to capture the constraints essential for enabling automation of configuration, and view types to manage the inherent complexity of CPSs and provides support for multi-stage multi-step configuration. Moreover, it also shows that all the important functionalities of an automated configuration solution are covered by the framework.

The rest of the paper is organized as follows: Section 2 introduces three real-word applications used for evaluation. Section 3 presents details related to *ContextFormalization* where we discuss PLE terminologies, CPSs, and configuration process. In Sections 4 and 5, we present details related to *DomainEngineering* and *ApplicationEngineering* of CPS PLs, respectively. The validation of the framework is presented in Section 6. Section 7 summarizes the literature review and Section 8 concludes the paper.

2 Real-World Applications

In the following sections, we discuss three real-world CPS PLs used for the validation of the framework for supporting multi-stage and multi-step automated configuration of CPS PLs.

2.1 Material Handling System

The first case study is a PL of Material Handling Systems (MHSs) developed with the inspiration when we were collaborating with ULMA Handling System[3] in the context of an EU Horizon 2020 project U-Test[4]. ULMA produces a large variety of MHSs worldwide [17]. It consists of several sub-systems such as Automatic Guided Vehicle (AGV), Automatic Storage Retrieval System (ASRS), and Automatic Identification and Data Collection (AIDC). AGV is an automatic transport system that uses unmanned vehicles to transport different types of loads without human intervention. It is typically used in warehouse, production, and logistics for the safe movement of goods to minimize labor cost and material damage. ASRS is an automated system for inventory management, which is used to place and retrieve the loads from pre-defined locations in the warehouse. AIDC is used to identify, verify, record, and track products. To summarize, MHS is an integrated, large-scale, hierarchal, and highly customizable system of systems where each subsystem of MHS involves a large number of variabilities. Such a complex system is a representative of CPS PLs, which makes it suitable for this study.

2.2 Video Conferencing System

The second case study is a PL of commercial Video Conferencing Systems (VCSs) called *Saturn* developed by Cisco Systems[5], Norway, which had a long-term collaboration with Simula Research Laboratory under Certus-SFI [18]. In total, *Saturn* consists of 20 subsystems such as audio and video subsystems. Each subsystem can run in parallel to the subsystem implementing the core functionality dealing with establishing video conferences. The *Saturn* PL consists of various PLs of hardware codecs (called endpoints[6]) including C-Series, MX-Series, and SX-Series and PLs of software[7] running on these endpoints (e.g., TC-Series, CE-Series). Both hardware codec and software PLs consist of several products. For example, C-Series (i.e., a PL of codecs) have four endpoints C20, C40, C60, and C90 where C20 has minimum hardware and the lowest performance in the C-Series PL. Similarly, TC-Series (i.e., a software PL) has 10 versions of software that can be installed on different endpoints, where each software version has hundreds of configurable parameters (e.g., default protocol and encryption). In summary, *Saturn* is an

[3] www.ulmahandling.com
[4] www.cordis.europa.eu/project/id/645463
[5] www.cisco.com
[6] https://www.cisco.com/c/en/us/products/collaboration-endpoints/product-listing.html
[7] https://software.cisco.com/download/release.html?mdfid=286271155&flowid=71282&softwareid=280886992&release=CE-console-v8.1.0&relind=AVAILABLE&rellifecycle=&reltype=latest

integrated, large-scale, and highly customizable system of systems where each subsystem involves a large number of variabilities.

For the validation purpose, we selected three VCS member products from *Saturn* where three software versions TC 7.0, TC 7.2, and SE 8.0 are installed on C60, MX300, and SX20 respectively. TC 7.0 and TC 7.2 belong to one PL (i.e., TC series of software) whereas SE 8.0 belongs to another PL (i.e., SE series of software). Similarly, C60, MX300, and SX20 belong to three different PLs. These three products have hundreds of configurable parameters (e.g., call rate, default protocol), which are to be configured at the post-deployment time.

2.3 Subsea Production System

The third case study is a PL of Subsea Production Systems (SPSs) in which software controls and monitors the operation of electrical and mechanical instruments. An SPS has hundreds of control modules and thousands of instruments [13]. In the SPS PL, the hardware topology can vary from one product to another, with each topology being a specific configuration of the generic family design. Hardware is configured based on customer requirements, environmental settings, and different regulations and standards. Different products in the SPS PL share the same software code base configured differently for each product, mainly based on the hardware topology. For example, the number of mechanical and electrical instruments as well as their properties (e.g., resolution of a sensor) affect the number and values of runtime objects in the software configured for a specific product. Such dependencies between the hardware and software should be captured and accounted for during the configuration process. Software and hardware variabilities occur at different levels of abstraction and are usually resolved by various domain experts in different phases of the product development lifecycle. For example, high-level hardware decisions (e.g., the number of wells) are made by domain experts after tendering and front-end engineering design phases whereas low-level variabilities (e.g., the operating range of a device) are usually configured by engineers during the configuration, testing, or operation phases. In summary, SPS is an integrated, large-scale, and highly customizable, thus, an example of CPS PLs.

3 Context Formalization

In this section, we discuss the PLE terminology in Section 3.1 followed by concepts related to CPSs in Section 3.2 whereas, in Section 3.3, we formalize the multi-stage and multi-step configuration process.

3.1 PLE Terminologies

We constructed a conceptual model as shown in Figure B-2 to clarify several key PLE concepts and their relationships according to the PLE ISO/IEC standard for Product Line Engineering and Management [11]. Note that, in total, we have used 16 (out of 29) concepts from the standard. The definitions of the concepts in Figure B-2 are provided in Table B-15 in Appendix A along with a running example from the SPS case study (Figure B-3), which is modeled with notations of UML class diagram and the UML profile of the SimPL methodology [19] to capture the commonalities and variabilities of a PL.

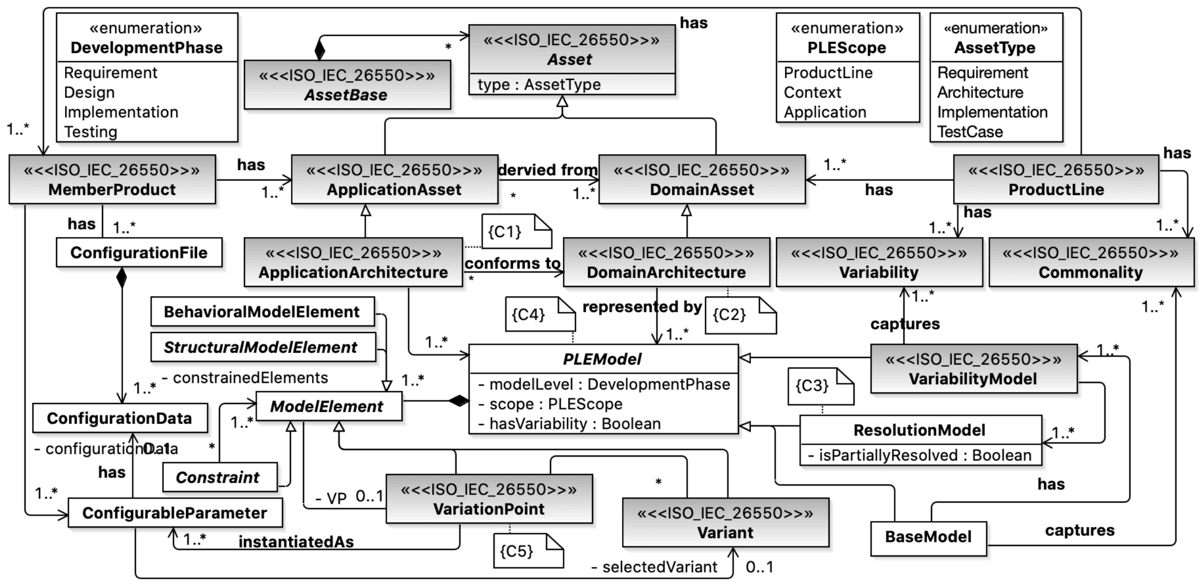
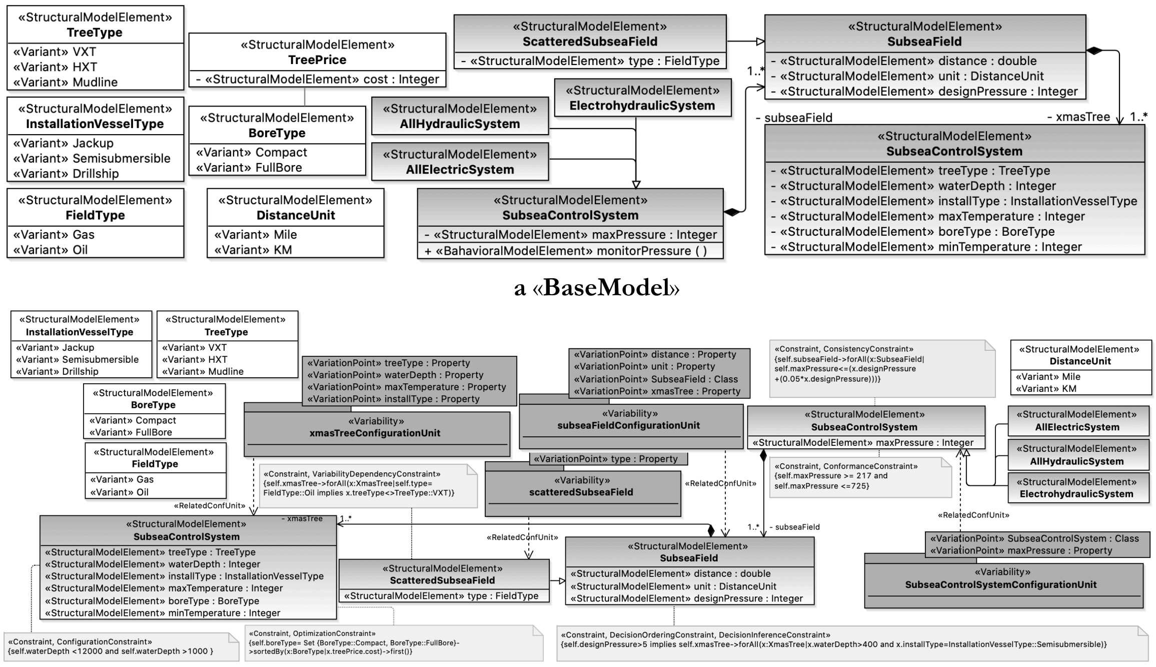

«enumeration» DevelopmentPhase
Requirement
Design
Implementation
Testing
«<<ISO_IEC_26550>>» AssetBase
«<<ISO_IEC_26550>>» Asset
type : AssetType
has
«enumeration» PLEScope
ProductLine
Context
Application
«enumeration» AssetType
Requirement
Architecture
Implementation
TestCase
«<<ISO_IEC_26550>>» MemberProduct
«<<ISO_IEC_26550>>» ApplicationAsset
dervied from
«<<ISO_IEC_26550>>» DomainAsset
«<<ISO_IEC_26550>>» ProductLine
ConfigurationFile
«<<ISO_IEC_26550>>» ApplicationArchitecture
conforms to
«<<ISO_IEC_26550>>» DomainArchitecture
«<<ISO_IEC_26550>>» Variability
«<<ISO_IEC_26550>>» Commonality
{C1}
{C2}
{C3}
{C4}
{C5}
represented by
captures
BehavioralModelElement
StructuralModelElement
- constrainedElements
PLEModel
- modelLevel : DevelopmentPhase
- scope : PLEScope
- hasVariability : Boolean
«<<ISO_IEC_26550>>» VariabilityModel
ConfigurationData
- configurationData
ModelElement
ResolutionModel
- isPartiallyResolved : Boolean
Constraint
«<<ISO_IEC_26550>>» VariationPoint
- VP
«<<ISO_IEC_26550>>» Variant
BaseModel
ConfigurableParameter
instantiatedAs
- selectedVariant

Figure B-2. A conceptual model for PLE[8]


«StructuralModelElement» TreeType
«Variant» VXT
«Variant» HXT
«Variant» Mudline
«StructuralModelElement» TreePrice
- «StructuralModelElement» cost : Integer
«StructuralModelElement» ScatteredSubseaField
- «StructuralModelElement» type : FieldType
«StructuralModelElement» SubseaField
- «StructuralModelElement» distance : double
- «StructuralModelElement» unit : DistanceUnit
- «StructuralModelElement» designPressure : Integer
«StructuralModelElement» InstallationVesselType
«Variant» Jackup
«Variant» Semisubmersible
«Variant» Drillship
«StructuralModelElement» BoreType
«Variant» Compact
«Variant» FullBore
«StructuralModelElement» AllHydraulicSystem
«StructuralModelElement» AllElectricSystem
«StructuralModelElement» ElectrohydraulicSystem
- subseaField
- xmasTree
«StructuralModelElement» SubseaControlSystem
- «StructuralModelElement» treeType : TreeType
- «StructuralModelElement» waterDepth : Integer
- «StructuralModelElement» installType : InstallationVesselType
- «StructuralModelElement» maxTemperature : Integer
- «StructuralModelElement» boreType : BoreType
- «StructuralModelElement» minTemperature : Integer
«StructuralModelElement» FieldType
«Variant» Gas
«Variant» Oil
«StructuralModelElement» DistanceUnit
«Variant» Mile
«Variant» KM
«StructuralModelElement» SubseaControlSystem
- «StructuralModelElement» maxPressure : Integer
+ «BehavioralModelElement» monitorPressure ( )
«Variability» xmasTreeConfigurationUnit
«Variability» subseaFieldConfigurationUnit
«Variability» scatteredSubseaField
«Variability» SubseaControlSystemConfigurationUnit
«RelatedConfUnit»

a «BaseModel»

b «VariabilityModel»

Figure B-3. Running example (an excerpt of the SPS case study modeled using UML class diagram and SimPL methodology) *

* *ConfigUnits* with template parameters (in dark grey color) show captured variabilities. Variabilities corresponding to attributes and cardinalities are represented as "Property" type template parameters and "Class" type template parameters for variabilities related to subclasses.

AssetBase is a repository containing a set of *DomainAsset*s and *ApplicationAsset*s where an *Asset* can be of four types: *Requirement*, *Architecture*, *Implementation*, and *TestCase*. A *ProductLine* has *DomainAsset*s whereas a *MemberProduct* has *ApplicationAsset*s. *DomainArchitecture* is a *DomainAsset*

[8] C1-C5 are OCL constraints provided in Appendix B.

and *ApplicationArchitecture* is an *ApplicationAsset*. Both *DomainArchitecture* and *ApplicationArchitecture* are represented by one or more *PLEModel*s. *PLEModel* is characterized by *modelLevel* to indicate the phase of development life cycle (i.e., *Requirement*, *Design*, *Implementation*, and *Testing*) to which *PLEModel* belongs. *PLEModel* is also characterized by *scope* and *hasVariability* to indicate different *scope*s (i.e., *ProductLine*, *Application*, and *Context*) and the presence of *Variability* in *PLEModel*. *Context* represents the environment in which the system operates and it consists of external agents (i.e., users, external systems and/or cloud services) and physical environment [3]. A *PLEModel* can be of three types: *BaseModel*, *VariabilityModel*, and *ResolutionModel*. *ResolutionModel* is characterized by *isPartiallyResolved* to indicate if the *ResolutionModel* has unresolved variabilities. A *ProductLine* has *Commonality* and *Variability*, which are captured and managed using *BaseModel* and *VariabilityModel* respectively. A *PLEModel* representing the *DomainArchitecture* of a *ProductLine* is either *BaseModel* or *VariabilityModel* whereas a *PLEModel* representing the *ApplicationArchitecture* of a *MemberProduct* is *ResolutionModel*. A *PLEModel* is composed of one or more *ModelElement*s where a *ModelElement* can be *StructuralModelElement*, *BehavioralModelElement*, *VariationPoint*, *Variant*, or *Constraint*. A *ConfigurableParameter* may have one *ConfigurationData*, which represents the configuration decision made to configure a *ConfigurableParameter*. A *MemberProduct* has one or more *ConfigurationFile*s where each *ConfigurationFile* contains one or more Co*nfigurationData*.

3.2 Cyber-Physical System

Figure B-4 presents a conceptual model for CPS and the definitions of the concepts presented in Figure B-4 are provided in Table B-16 in Appendix A. As shown in Figure B-4, a CPS constitutes a set of *PhysicalComponent*s, *CyberComponent*s with deployed *Software*, and *InterfacingComponent*s, which are combined using a particular *Topology* to achieve a common goal. A CPS monitors and controls a set of *PhysicalProperty*. A *CyberComponent* can either be a *ComputationalComponent* or a *CommunicationComponent*, which takes values of *StateVariables* as input and updates their values if required. Both *CyberComponent* and *InterfacingComponent* can have several *ComponentProperty*. Similarly, a *PhysicalComponent* can have several *PhysicalProperty*. Both *PhysicalProperty* and *ComponentProperty* have attributes *name*, *type*, and *unit* to specify the name, type (e.g., String, Integer, Binary), and unit of a specific property. *PhysicalProperty* has an extra Boolean attribute *isContinuous* to specify whether it is a continuous or a discrete type of property.

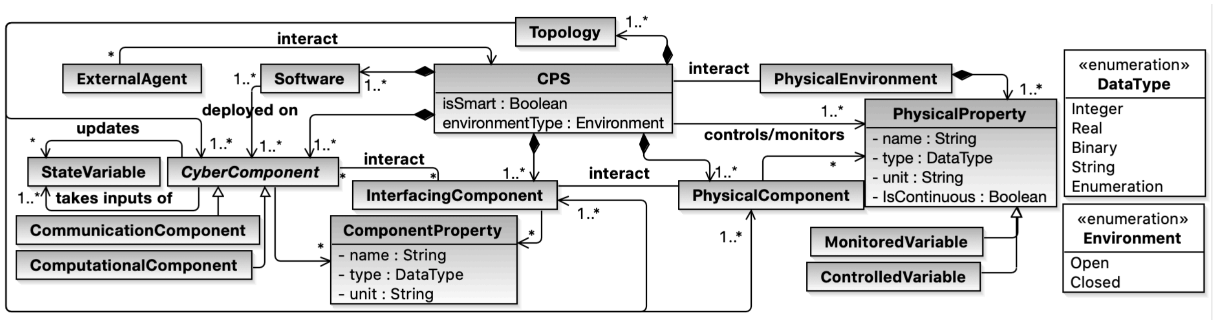


Figure B-4. A conceptual model for CP [16]

A CPS may interact with *PhysicalEnvironment* and *ExternalAgent*s. *PhysicalEnvironment* has at least one *PhysicalProperty*. A CPS can be implemented with the assumption of a closed world where everything is predefined and fixed or an open world where new *Variant*s and/or *VariationPoint*s can be added or removed at any time. This characteristic of CPS is specified using an enumerated attribute *environmentType* with two possible values *Closed* and *Open*. A CPS can also have autonomous behavior, which makes it "smart". This characteristic is specified by a Boolean attribute *isSmart*. Considering the above-mentioned two aspects, we can classify CPSs into four categories: smart closed CPSs, smart open CPSs, typical closed CPSs, and typical open CPSs. In this study, we focus on typical closed CPSs, which are referred as CPSs in the rest of the paper for the sake of simplicity.

3.3 Configuration Process

In Figure B-5, we present a conceptual model for the configuration process. The definitions of the concepts in the conceptual model are provided in Table B-17 in Appendix A.

As shown in Figure B-5, *ConfigurationSolution* enforces a *ConfigurationProcess* to perform *ProductConfiguration*. *ProductConfiguration* can be performed at pre-deployment, deployment, and/or post-deployment time. *ConfigurationProcess* has one or more *ConfigurationStage*s where each *ConfigurationStage* has at least one *ConfigurationStep*. *ConfigurationProcess* is characterized by *IsMultiStage*, *IsInteractive*, and *isIncremental* to show if the *ConfigurationProcess* is a multi-stage, interactive (i.e., requires input from the *Stakeholder*s and provides feedback to the *Stakeholder*s), and incremental (i.e., the configuration is performed incrementally in multiple *ConfigurationStage*s) process. *ConfigurationStage* has a Boolean attribute *IsMultiStep* to indicate if a *ConfigurationStage* contains more than one *ConfigurationStep*. For each *ConfigurationStage*, there is at least one *Stakeholder* who gives input to its *ConfigurationStep*s for making *ConfigurationDecision*s and gets feedback. A *ConfigurationStep* has one or more *ConfigurationDecision*s where each *ConfigurationDecision* is either inferred automatically (*isInferred*) or made manually by the *Stakeholder*s. *ConfigurationDecision*s are represented as *ConfigurationData*. *ConfigurationData* has attributes *isAutoGenerated*, *status*, *type*, *value*, and *parameterID* to specify if the data is generated automatically, its evaluation status (i.e., *Valid*, *Invalid*, *Unknown*), type (e.g., *Integer*, *Real*, *Boolean*), value (i.e., assigned/selected variant), and unique identifier for the corresponding *ConfigurableParameter*.

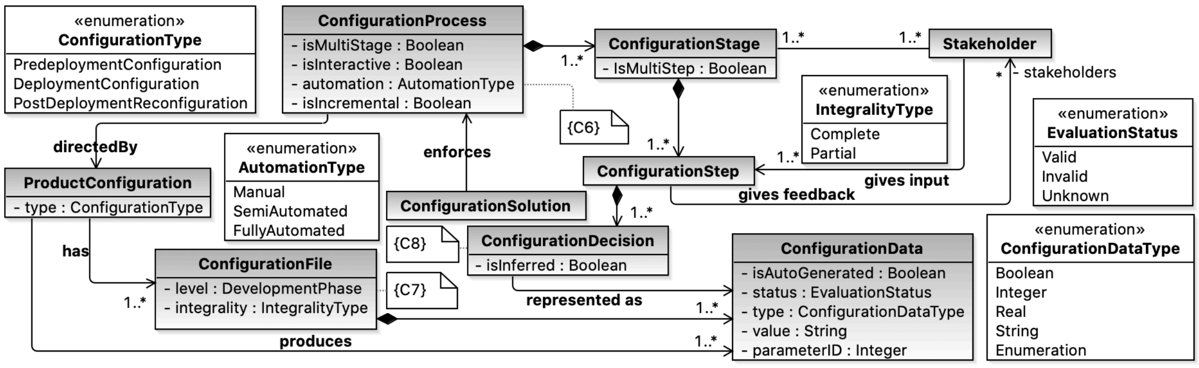


Figure B-5. A conceptual model for the configuration process[9]

CPS PLs involve various components from multiple domains (e.g., Mechanics, Software, and Electronics) and different domain experts (i.e., *Stakeholder*s) are responsible for configuring these components. Thus, *ConfigurationDecision*s for various domains are divided into multiple *ConfigurationStage*s to facilitate different domain experts. In most of the cases, *ConfigurationDecision*s

[9] C6-C8 are OCL constraints provided in Appendix B.

for a particular domain (e.g., Software) are made at various points in time by one or more *Stakeholder*s collaborating together. Therefore, *ConfigurationDecision*s within one *ConfigurationStage* can be divided into multiple *ConfigurationStep*s. In Figure B-6, we have provided a simplified example of the multi-stage and multi-step configuration process for the running example presented in Figure B-3.

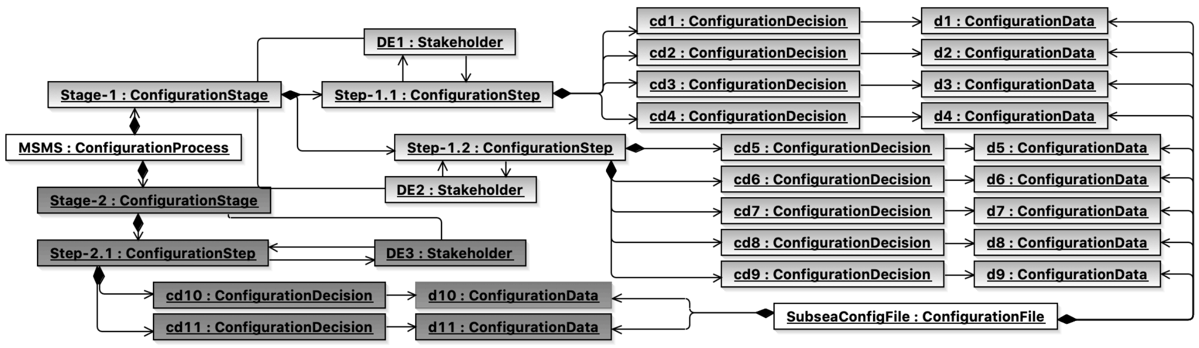

DE1 : Stakeholder
Stage-1 : ConfigurationStage
Step-1.1 : ConfigurationStep
MSMS : ConfigurationProcess
Step-1.2 : ConfigurationStep
DE2 : Stakeholder
Stage-2 : ConfigurationStage
Step-2.1 : ConfigurationStep
DE3 : Stakeholder
cd1 : ConfigurationDecision
cd2 : ConfigurationDecision
cd3 : ConfigurationDecision
cd4 : ConfigurationDecision
cd5 : ConfigurationDecision
cd6 : ConfigurationDecision
cd7 : ConfigurationDecision
cd8 : ConfigurationDecision
cd9 : ConfigurationDecision
cd10 : ConfigurationDecision
cd11 : ConfigurationDecision
d1 : ConfigurationData
d2 : ConfigurationData
d3 : ConfigurationData
d4 : ConfigurationData
d5 : ConfigurationData
d6 : ConfigurationData
d7 : ConfigurationData
d8 : ConfigurationData
d9 : ConfigurationData
d10 : ConfigurationData
d11 : ConfigurationData
SubseaConfigFile : ConfigurationFile

Figure B-6. Exemplifying multi-stage and multi-step configuration process for running example (UML object diagram for the conceptual model of the configurationprocess) *

*Note that two different colors are used to show two different configuration stages.

In Figure B-6, *MSMS* is a *ConfigurationProcess* containing two *ConfigurationStage*s (*Stage-1* and *Stage-2*) for configuring hardware and software of the subsea system presented in Figure B-3. *Stage-1* has two *ConfigurationStep*s *Step-1.1* and *Step-1.2* whereas *Stage-2* has only one *ConfigurationStep Step-2.1*. *Stakeholders DE1* and DE2 are two domain experts who make *ConfigurationDecision*s *cd1-cd4* and *cd5-cd9* in *Step-1.1* and *Step-1*.2 respectively. Similarly, *Stakeholder DE3* makes *ConfigurationDecision*s *cd10-cd11* in *Step-2.1* of *Stage-2*. In total, we have 11 *ConfigurationDecision*s in the *MSMS* for the running example, as we have 11 variabilities in the example (Figure B-3). All the *ConfigurationDecision*s (*cd1-cd11*) are represented as *ConfigurationData* (*d1-d11*) in a *ConfigurationFile* (i.e., *SubseaConfigFile* in Figure B-6). Note that in Figure B-6, we did not instantiate the attributes of different concepts (e.g., *ConfigurationData*, *ConfigurationProcess*) for the sake of simplicity.

4 Domain Engineering of CPS Product Lines

We present concepts related to the modeling of CPS PLs in Section 4.1 and a classification of *VariationPoint* types in Section 4.2 to capture various types of variabilities in CPS PLE. In Section 4.3, we present a classification of constraints in CPS PLE.

4.1 Modeling CPS Product Lines

Figure B-7 presents a conceptual model used to discuss the concepts related to the modeling of CPS PLs for capturing and managing the commonalities and variabilities of CPS PLs to support multi-stage and multi-step automated configuration of CPSs. The conceptual model is constructed as a UML class diagram and formalized using OCL constraints. The definitions of the concepts presented in Figure B-7 are provided in Table B-18 in Appendix A.

As shown in Figure B-7, a *ModelingLanguage* has a set of *MetaModelElement*s defining the *ModelingLanguage*. *PLEModel*s are developed using a *ModelingLanguage* where a *PLEModel* is composed of one or more *ModelElement*s of type *Constraint*, *StructuralModelElement*, *BehavioralModelElement*, *VariationPoint*, or *Variant*. *StructuralModelElement*s represent structural elements of CPSs (e.g., sensor, actuator, software component, or property), which can be of three types: *SoftwareStructuralModelElement*, *HardwareStructuralModelElement*, and *ContextStructuralModelElement*. *BehavioralModelElement* represents behavioral elements of CPSs corresponding to which behavioral variabilities can be defined. For example, *Variability* corresponding to *Interaction*. *Interaction* is a type of *BehavioralModelElement*, which describes how two or more components (i.e., source and target components) interact or communicate with each other [20]. *Interaction* is characterized by *isDirect*, *isHomogeneous*, and *direction*. *isDirect* indicates whether the *Interaction* is direct between the source and target components or it involves intermediate components. *isHomogeneous* shows if all interacting components are of the same type and *direction* indicates if the communication between the source and target components is *Unidirectional* or *Bidirectional*. According to [5, 21], CPS has three logical levels, i.e., application level, infrastructure level, and integration level. Based on these three levels we have classified the *Interaction*s into three categories (i.e., *ApplicationLevelInteraction*, *InfrastructureLevelInteraction*, and *IntegrationLevelInteraction*) as shown in Figure B-7.

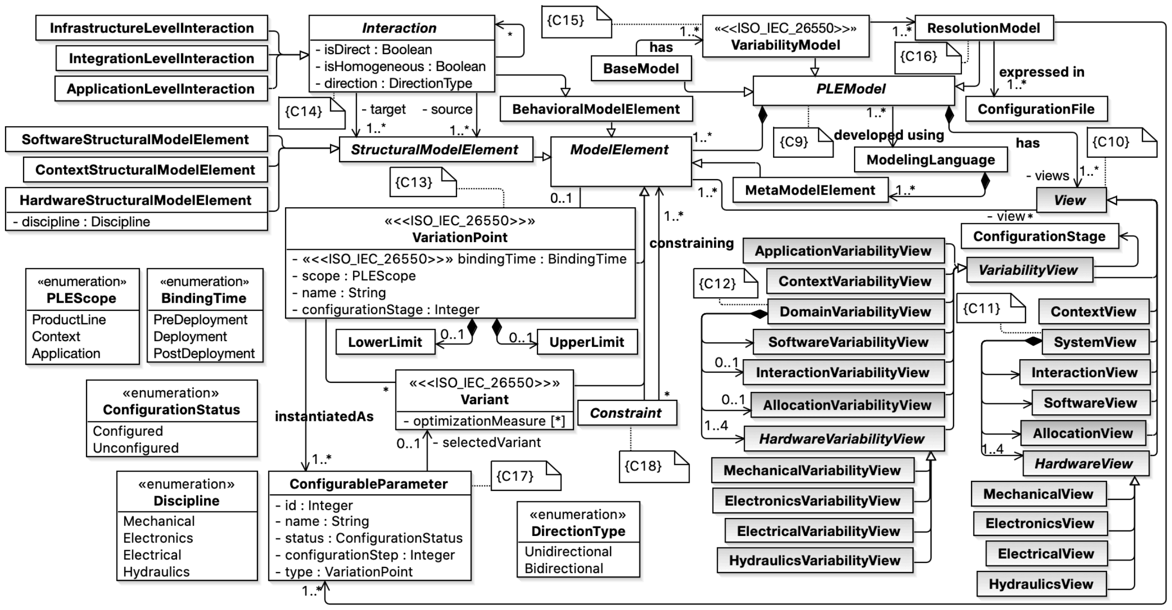


Figure B-7. A conceptual model for modeling CPS product lines[10]

VariationPoint is used to capture the *Variability* corresponding to a *StructuralModelElement* or a *BehavioralModelElement* where an instance of *VariationPoint*, i.e., *ConfigurableParameter* can be configured with more than one *Variant*s. *VariationPoint* is characterized by *type*, *scope*, and *bindingTime*. *Scope* indicates the scope of the *VariationPoint* (i.e., *ProductLine*, *Product*, and *Context*) whereas *type* represents the type of the *VariationPoint*. Types of the *VariationPoint* are discussed in detail in Section 4.2.

BindingTime specifies the time to resolve a *VariationPoint* by binding its instance with one of its *Variant*s. Since in CPS PLE, *ConfigurationDecision*s are made during the design/development phase (e.g., hardware designs, software features), at deployment time (e.g., hardware/software topologies software parameterization, deployment of software components to specific hardware), and after deployment (e.g., software parameterization at startup or runtime,

[10] C9-C18 are OCL constraints provided in Appendix B.

activation/deactivation of software features). Thus, we have classified the *bindingTime* into *PreDeployment, Deployment*, and *PostDeployment*, independent of the technologies and approaches used. Several existing studies [22, 23] discuss various binding times specific to the software development lifecycle such as compile time, link time, load time, initialization time, and runtime that can be mapped to our generic binding times. Most of them except runtime can be mapped to *Deployment* whereas runtime can be mapped to *PostDeployment*.

A finite set of *Variant*s corresponding to a *VariationPoint* can be specified as a pre-defined list whereas the infinite number of *Variant*s (e.g., for a *VariationPoint* corresponding to a Real type variable) can be denoted by specifying the *UpperLimit* and *LowerLimit*. *ConfigurableParameter* is characterized by *id*, *name*, *status* (i.e., *Configured*, *Unconfigured*), *configurationStep*, and *type* (i.e., type of *VariationPoint*). Optionally, a *Variant* can also be characterized by *optimizationMeasure*s (e.g., cost, performance, energy consumption) that assist the configuration optimization. Different types of *Constraint* are discussed in Section 4.3.

Separation of concerns is considered as an important aspect of software engineering. It becomes more important in the case of complex, highly hierarchal, large-scale, and multi-disciplines CPSs that involve different *Stakeholder*s from diverse domains such as Mechanics, Electronics, and Software. To support the separation of concerns and handle the CPSs more efficiently, modeling CPSs requires multi-views, which can also help in reducing the configuration complexity [24].

As shown in Figure B-7, *PLEModel* can have one or more *View*s showing different *ModelElement*s and their relationships. A *View* can be *SystemView* to show the commonalities of CPS PL or *VariabilityView* to represent variabilities of CPS PL. *SystemView* is a composite view containing one *SoftwareView*, one to four *HardwareView*s, one *AllocationView*, and one *InteractionView*. *HardwareView* is an abstract view, which can be *MechanicalView*, *ElectricalView*, *ElectronicsView*, or *HydraulicsView*. A *View* can also be *ContextView* to show the *ContextStructuralModelElement*s and their relationships. A *VariabilityView* is an abstract view, which can be *SoftwareVariabilityView*, *HardwareVariabilityView*, *AllocationVariabilityView*, *InteractionVariabilityView*, *DomainVariabilityView*, *ContextVariabilityView*, *and ApplicationVariabilityView*. *DomainVariabilityView* is a composite view containing one *SoftwareVariabilityView*, one to four *HardwareVariabilityView*, and optionally one *AllocationVariabilityView* and *InteractionVariabilityView*. A *HardwareVariabilityView* is an abstract view, which can be *MechanicalVariabilityView*, *ElectricalVariabilityView*, *ElectronicsVariabilityView*, or *HydraulicsVariabilityView*. Note that for each concrete *VariabilityView* (e.g., *MechanicalVariabilityView*), we have one *ConfigurationStage* to resolve the variabilities in one or more *ConfigurationStep*s.

4.2 Classification of Variation Point Types

In Section 4.1, we present a set of basic VP types, followed by the discussion on CPS-specific VP types in Section 4.2.2.

4.2.1 Basic Variation Point Types

Based on the basic data types in mathematics, we constructed a conceptual model to classify them, as shown in Figure B-8. A *Variable* can be a *VariationPoint* or a *Non*-configurable *Variable*, which represents the configurable and non-configurable variables in CPS PLE. Each *Variable* has a *Type*, which is classified as *Atomic* and *Composite*. A *Variable* of *Atomic Type* takes a single value at a given point in time whereas a *Variable* of *Composite Type* is composed of more than one *Atomic Type Variable*s. *Atomic Type* is further classified as *Quantitative* and *Qualitative* where they take numeric and non-numeric values, respectively. A *Quantitative Type* can be *Discrete* taking countable values or *Continuous* taking uncountable values. *Integer* is the concrete *Discrete* type and *Real* is the concrete *Continuous* type. *Qualitative Type* is classified as *String*, *Binary*, and *Categorical* that is further classified into *Ordinal* and *Nominal*.

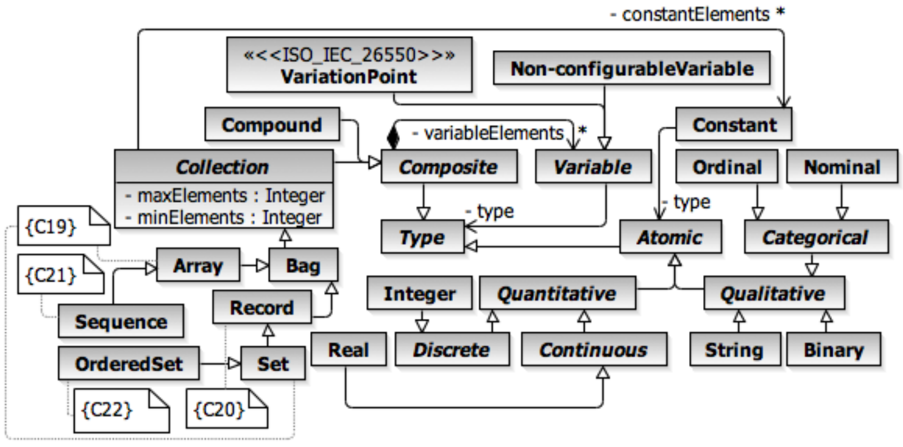


Figure B-8. Basic data types [16][11]

A *Variable* of *Composite* Type combines several *Variable*s and/or *Constant*s, which is classified as *Compound* and *Collection*. *Compound* takes only *Variable*s, e.g., complex numbers in SysML containing two *Variable*s realPart and imaginaryPart [25], whereas *Collection* takes *Variable*s and/or *Constant*s, e.g., a collection of colors. Attributes *minElements* and *maxElements* of *Collection* specify the minimum and maximum numbers of elements in a collection. As shown in Figure B-8, we have classified *Collection* into six types *Bag*, *Array*, *Record*, *Set*, *OrderedSet*, and *Sequence* based on three properties: homogeneity, uniqueness, and order. The homogeneity, uniqueness, and order properties of each Collection type are specified as OCL constraints (Appendix B). Table B-1 summarizes the six types of *Collection* along with their properties.

Table B-1. Collection types [16]

Collection Type	Homogeneity	Uniqueness	Order
Bag	No	No	No
Array	Yes	No	No
Record	No	Yes	No
Set	Yes	Yes	No
OrderedSet	Yes	Yes	Yes
Sequence	Yes	No	Yes

To validate the conceptual model of the basic data types, we mapped the data types defined in the MARTE Value Specification Language-VSL [26] and SysML [25] to the basic data types presented in Figure B-8. We used MARTE and SysML for validation because these two modeling languages can be used for modeling CPSs [1, 27]. During the validation, we do not include the extended data types provided in MARTE, as they are defined by extending the data types used in

[11] C19-C22 are OCL constraints provided in Appendix B.

our mapping. In case of SysML, we include all the data types. Results of the mapping are given in Table B-2, where one can see that each data type in MARTE and SysML has a correspondence in our basic data type classification, which suggests that our classification of the basic data types is complete.

Table B-2. Mapping MARTE and SysML data types to the basic data types [16]

MARTE	SysML	Basic Data Types
Integer	Integer	Integer
UnlimitedNatural	UnlimitedNatural	Integer
Boolean	Boolean	Binary
String	String	String
Real	Real	Real
DateTime	Complex	Compound
EnumerationType	Enumeration	Ordinal/Nominal
-	ControlValue	Ordinal/Nominal
IntervalType	UnitAndQuantityKind	Compound
TupleType	-	Compound
ChoiceType	-	Compound
CollectionType	-	Collection

In Figure B-9, we present a classification of basic VP types where one basic VP type is defined corresponding to each basic data type presented in Figure B-8. A *VariationPoint* can be a *CompositeVP* or an *AtomicVP*. An *AtomicVP* can come with any of the six concrete types: *StringVP*, *BinaryVP*, *NominalVP*, *OrdinalVP*, *IntegerVP*, and *RealVP* corresponding to *String*, *Binary*, *Nominal*, *Ordinal*, *Integer*, and *Real* respectively. A *CompositeVP* can be *CompoundVP* or *CollectionVP*, which are defined corresponding to *Compound* and *Collection* data types respectively. As shown in Figure B-9, a *CompositeVP* may have several *AtomicVP*s and/or *CompositeVP*s depending on the number of *variableElements* (Figure B-8) involved in the *Composite* data type. *CollectionVP* may have two additional *IntegerVP*s, i.e., *lowerLimitVP* and *upperLimitVP* corresponding to the minimum and maximum numbers of the elements in the collection.

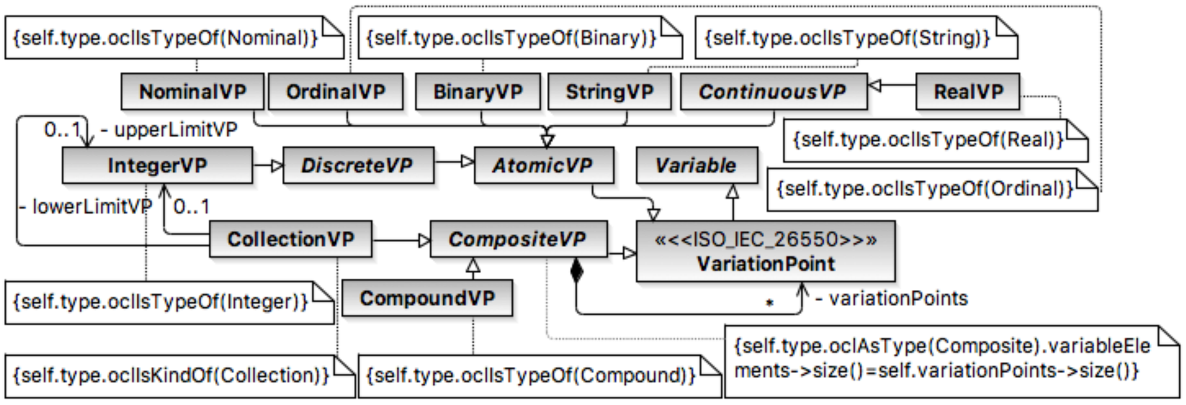

{self.type.oclIsTypeOf(Nominal)}
{self.type.oclIsTypeOf(Binary)}
{self.type.oclIsTypeOf(String)}
NominalVP
OrdinalVP
BinaryVP
StringVP
ContinuousVP
RealVP
0..1
- upperLimitVP
{self.type.oclIsTypeOf(Real)}
IntegerVP
DiscreteVP
AtomicVP
Variable
{self.type.oclIsTypeOf(Ordinal)}
- lowerLimitVP
0..1
CollectionVP
CompositeVP
«<<ISO_IEC_26550>>»
VariationPoint
{self.type.oclIsTypeOf(Integer)}
CompoundVP
*
- variationPoints
{self.type.oclAsType(Composite).variableElements->size()=self.variationPoints->size()}
{self.type.oclIsKindOf(Collection)}
{self.type.oclIsTypeOf(Compound)}

Figure B-9. Classification of the basic VP types [16]

4.2.2 CPS-specific Variation Point Types

Based on the conceptual model of CPS presented in Figure B-4, we derive a set of CPS-specific VP types (Table B-3). In Table B-3, the first column represents the CPS concepts used to derive CPS-specific VP types and the second column shows the derived CPS-specific VP types. The last column presents the basic VP type corresponding to a particular CPS-specific VP type. The precise definitions of CPS-specific VP types are provided in Table B-19 in Appendix A.

Table B-3. CPS-specific VP types [16]

CPS Concept	CPS-Specific VP Type	Basic VP Type
CP	Descriptive-VP	StringVP
CP, PP	DiscreteMeasurement-VP	IntegerVP
CP, PP	ContinuousMeasurement-VP	RealVP
CP, PP	BinaryChoice-VP	BinaryVP
CP, PP	PropertyChoice-VP	NominalVP/OrdinalVP
CP, PP	MeasurementUnitChoice-VP	OrdinalVP
CP, PP	MeasurementPrecision-VP	RealVP
CP, PP, COM	Multipart/Compound-VP	CompoundVP
COM	ComponentCardinality-VP	IntegerVP
COM	ComponentCollectionBoundary-VP	IntegerVP
COM	ComponentChoice-VP	NominalVP/OrdinalVP
COM	ComponentSelection-VP	CollectionVP
Topology	TopologyChoice-VP	NominalVP
Deployment	AllocationChoice-VP	NominalVP
Interact	InteractionChoice-VP	NominalVP
Constraint	ConstraintSelection-VP	CollectionVP

*CP=ComponentProperty, PP=PhysicalProperty, COM= *CyberComponent*, *InterfacingComponent*, or *PhysicalComponent*

As shown in Table B-3, seven VP types: Descriptive-VP, DiscreteMeasurement-VP, ContinuousMeasurement-VP, BinaryChoice-VP, PropertyChoice-VP, MeasurementUnitChoice-VP, and MeasurementPrecision-VP are defined to capture the variabilities corresponding to *PhysicalProperty* and/or *ComponentProperty* of CPS. Similarly, ComponentCardinality-VP, ComponentCollectionBoundary-VP, ComponentChoice-VP, and ComponentSelection-VP are defined to capture the variabilities related to *CyberComponent*s, *InterfacingComponent*s, and *PhysicalComponent*s. Multipart/Compound-VP can be specified for a *PhysicalProperty*, *ComponentProperty*, *CyberComponent*, *InterfacingComponent*, or *PhysicalComponent* that requires configuring several constituent VPs involved in it. This is very useful when different properties do not give complete meaning unless they are combined together. For example, length is a *PhysicalProperty*, which is meaningless without a unit. Hence, we need a Compound-VP type, which involves two VPs length and its unit. A Compound-VP can also be defined for a component (e.g., sensor), which contains several other VPs defined for its properties. TopologyChoice-VP, AllocationChoice-VP, and InteractionChoice-VP are defined to capture the variabilities related to *Topology* of CPSs, *Software* deployment, and *Interaction*. ConstraintSelection-VP is defined to select a subset of constraints.

4.3 Classification of Constraints in PLE

Constraints play a crucial role in the *ConfigurationProcess* of CPS PLE. To enable the automation of configuration for CPS PLs, we need to capture different types of constraints. In our previous work [6], we proposed a classification of constraints in PLE that we extend further in this paper by adding four new types of constraints (i.e., *WellFormednessConstraint*, *ConformanceConstraint*, *DecisionInferenceConstraint*, and *OptimizationConstraint*) as shown in Figure B-10. The rationale behind extending the constraint classification is to differentiate between different constraints and support an automated configuration solution enriched with more functionalities compared with previous work [6]. For example, we added *WellFormednessConstraint*, *ConformanceConstraint*, and

OptimizationConstraint, which facilitate *WellFormednessChecking*, *ConformanceChecking*, and *ConfigurationOptimization* respectively. Moreover, we also added *DecisionInferenceConstraint* to the classification to differentiate between the *VariabilityDependencyConstraint*s that support and do not support *DecisionInference*.

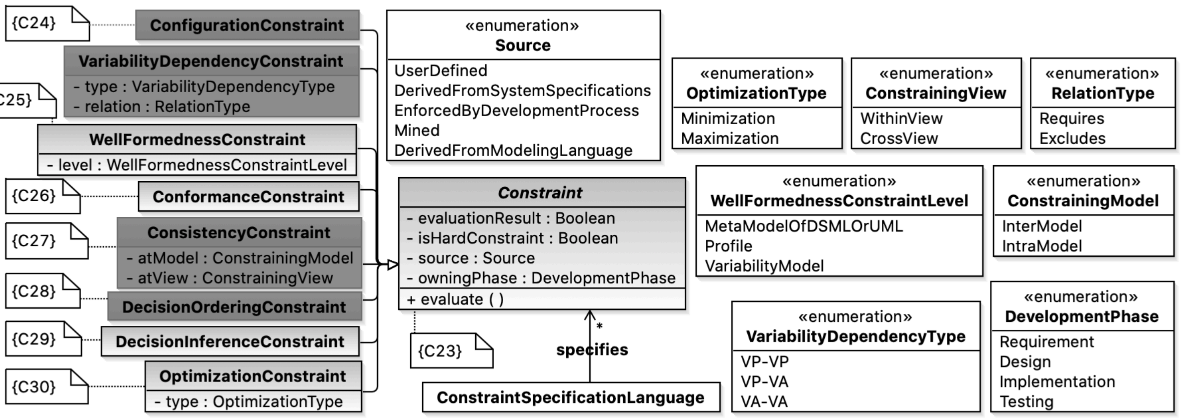

{C24}
ConfigurationConstraint
VariabilityDependencyConstraint
- type : VariabilityDependencyType
- relation : RelationType
{C25}
WellFormednessConstraint
- level : WellFormednessConstraintLevel
{C26}
ConformanceConstraint
{C27}
ConsistencyConstraint
- atModel : ConstrainingModel
- atView : ConstrainingView
{C28}
DecisionOrderingConstraint
{C29}
DecisionInferenceConstraint
{C30}
OptimizationConstraint
- type : OptimizationType
«enumeration» Source
UserDefined
DerivedFromSystemSpecifications
EnforcedByDevelopmentProcess
Mined
DerivedFromModelingLanguage
Constraint
- evaluationResult : Boolean
- isHardConstraint : Boolean
- source : Source
- owningPhase : DevelopmentPhase
+ evaluate ( )
{C23}
specifies
*
ConstraintSpecificationLanguage
«enumeration» OptimizationType
Minimization
Maximization
«enumeration» ConstrainingView
WithinView
CrossView
«enumeration» RelationType
Requires
Excludes
«enumeration» WellFormednessConstraintLevel
MetaModelOfDSMLOrUML
Profile
VariabilityModel
«enumeration» ConstrainingModel
InterModel
IntraModel
«enumeration» VariabilityDependencyType
VP-VP
VP-VA
VA-VA
«enumeration» DevelopmentPhase
Requirement
Design
Implementation
Testing

Figure B-10. Constrain classification[12]

As shown in Figure B-10, *Constraint* is a general concept characterized by *evaluationResult*, and *owningPhase* (i.e., *Requirement*, *Design*, *Implementation*, and *Testing*). A *Constraint* is either a hard constraint or a soft constraint [28], which is specified by a Boolean attribute *isHardConstraint*. Hard constraints cannot be false for a valid *MemberProduct* whereas, on the other hand, soft constraints can be true or false. Furthermore, based on the *source* of the *Constraint*, a constraint can be: 1) *UserDefined* where the constraint is defined by a domain expert, 2) *DerivedFromSystemSpecifications* where the constraint is derived from requirements of the system, 3) *EnforcedByDevelopmentProcess* where the constraint is enforced by the development process used by a particular organization, for example, hardware components should be configured before corresponding software components, 4) *Mined* where the constraint is inferred using machine learning, or 5) *DerivedFromModelingLanguage* where the constraint is derived from the syntax of the modeling language, for example, the constraint imposed due to mandatory feature in the feature model. A *Constraint* can be *ConfigurationConstraint*, *VariabilityDependencyConstraint*, *WellFormednessConstraint*, *ConformanceConstraint*, *ConsistencyConstraint*, *DecisionOrderingConstraint*, *DecisionInferenceConstraint*, or *OptimizationConstraint*. These constraints can be specified using different *ConstraintSpecificationLanguage*s such as OCL. In the rest of this section, we discuss the above-mentioned eight types of *Constraint* in detail. Additionally, we have provided a precise definition of each *Constraint* type using mathematical notations based on set theory, which can be found in Appendix C.

ConfigurationConstraints: *ConfigurationConstraint*s are defined on the *VariationPoint*s, which can be used to configure the *ConfigurableParameter*s with corresponding valid *Variant*s to derive a valid *MemberProduct* from a *ProductLine* [20, 29]. As shown in Figure B-10, *ConfigurationConstraint*s can be hard constraints (e.g., constraints defined by domain experts) as well as soft constraints (e.g., mined constraints [20, 30]) [28], which can come from four different sources: *UserDefined*,

[12] C22-C30 are OCL constraints provided in Appendix B. Also, *Constraint* types presented in dark grey are borrowed from our previous work.

DerivedFromSystemSpecifications, *EnforcedByDevelopmentProcess,* and *Mined.* Furthermore, *ConfigurationConstraint*s can also belong to different phases of the development cycle: *Requirement*, *Design*, *Implementation*, and *Testing*. For example, in the context of the SPS case study (Section 2.3), *ProductConfiguration* starts while requirement specification of a *MemberProduct* during which high-level *ConfigurationDecision*s are often made, e.g., the number of subsea control modules to deploy according to the number of wells to exploit as well as ranges of temperature sensors.

A *ConfigurationConstraint* can be specified for a *ProductLine*, a *MemberProduct*, or *Context* depending on the *scope* (Figure B-7) of *VariationPoint* being constrained. *ConfigurationConstraint*s constraining the *VariationPoint* with the *scope* (Figure B-7) of *ProductLine* are enforced during the pre-deployment time configuration of each *MemberProduct* whereas the ones with the *scope* (Figure B-7) of *MemberProduct* are enforced during the deployment or post-deployment time configuration of a *MemberProduct*. For example, in the VCS case study (Section 2.2), users need to configure several *ConfigurableParameter*s of a *MemberProduct* at the post-deployment time (e.g., call rate and network protocol of C20). To perform the post-deployment time configuration of these *ConfigurableParameter*s, several *ConfigurationConstraint*s need to be specified for the *MemberProduct*s. *ConfigurationConstraint*s constraining the *VariationPoint* with the *scope* (Figure B-7) of *Context* are enforced while resolving the variabilities related to the *Context* of CPS.

```
context XmasTree inv Exp-1:
self.waterDepth <12000 and self.waterDepth >1000
……………………………………………………………………..
context C20 inv Exp-2:
self.callRate >63 and self.callRate <6000
```

Listing 1: Examples of ConfigurationConstraints

The constraint Exp-1 in Listing 1 is a *ConfigurationConstraint* from the SPS case study defined on *XmasTree* class in Figure B-3, which is constraining a *VariationPoint* named *waterDepth* with the scope of *ProductLine*. It states that *waterDepth* can be configured with a value smaller than 12, 000 and larger than 1,000. Similarly, Exp-2 is another *ConfigurationConstraint* from the VCS case study, which is defined for a *MemberProduct* C20 to specify the range for callRate.

VariabilityDependencyConstraints: *VariabilityDependencyConstraint*s are implied restrictions on the relationship (e.g., *Requires* and *Excludes*) of different *VariationPoint*s and their *Variant*s [31, 32]. As shown in Figure B-10, *VariabilityDependencyConstraint*s can be of three types *VP-VP*, *VP-VA*, and *VA-VA* [33]. *VP-VP* implies configuring a *VariationPoint* vp_1 requires configuring another *VariationPoint* vp_2 first. *VP-VA* implies that if one *VariationPoint* is resolved, then another *VariationPoint* should be resolved by binding one specific *Variant*. *VA-VA* implies if one *VariationPoint* is resolved by binding one of its *Variant*s, then another *VariationPoint* should be resolved by binding one of its *Variant*s. *VariabilityDependencyConstraint*s originate from four different sources (i.e., *UserDefined*, *DerivedFromSystemSpecifications*, *DerivedFromModelingLanguage*, *Mined*) and different phases (i.e., *Requirement*, *Design*, *Implementation*, *Testing*) of the development cycle (Figure B-10). *VariabilityDependencyConstraint*s can be hard constraints (e.g., constraints defined by domain experts) as well as soft constraints (e.g., mined constraints [20, 30, 34]). As discussed earlier, *ProductConfiguration* starts by making high-level *ConfigurationDecision*s (e.g., the number of subsea oil and gas wells to exploit for an SPS product) at requirements engineering phase and then proceeds to the design and implementation phases by configuring *ConfigurableParameter*s (e.g., an engineering unit of a sensor and post-deployment

*ConfigurableParameter*s of deployed software). *VariationPoint*s exist in all the development phases, thus, *VariabilityDependencyConstraint*s among them need to be captured.

```
context ScatteredSubseaField inv Exp-3:
self.xmasTree->forAll(x:XmasTree|self.type= FieldType::Oil implies x.treeType<>TreeType::VXT)
```

Listing 2: An Example of VariabilityDependencyConstraint

The constraint Exp-3 in Listing 2 is an example of *VA-VA* type and *Excludes* relation *VariabilityDependencyConstraint* from the SPS case study (Figure B-3). It shows an implied relationship between two *Variant*s (i.e., *Oil* and *VXT*) corresponding to two *VariationPoint*s (i.e., *type* of *ScatteredSubseaField* and *treeType* of *xmasTree*). The constraint implies that for all *xmasTree* of *ScatteredSubseaField* if *type* is *Oil* then *treeType* should not be *VXT*.

WellFormednessConstraints: Generally speaking, *WellFormednessConstraint*s ensure that models are well-formed such that they conform to its modeling language's syntax and additionally defined constraints [35]. In the context of CPS PLE, *WellFormednessConstraint*s can be defined for *VariabilityModel, BaseModel,* and *ResolutionModel.* Usually, these models are developed either using a Domain-Specific Modeling Language (DSML) [36-38] or a UML profile [32] where *WellFormednessConstraint*s are defined on meta-model elements of DSML/UML or/and stereotypes defined in UML profile and validated at one lower level of abstraction. Well-formedness of *VariabilityModel* and *BaseModel* is ensured by modeling tools, however, the well-formedness of *ResolutionModel* needs to be ensured by the configuration engineers, as they develop the *ResolutionModel* by instantiating the *VariationPoint*s during the *ConfigurationProcess*. Thus, we are interested in *WellFormednessConstraint*s for *ResolutionModel. WellFormednessConstraint*s are hard constraints, which can be *UserDefined* or *DerivedFromModelingLanguage.*

```
context ConfigurableParameter inv Exp-4:
self.name<>null and self.type<>null
```

Listing 3: An Example of WellFormednessConstraint

The constraint Exp-4 in Listing 3 is an example of *WellFormednessConstraint*, which ensures that *name* and *type* of a *ConfigurableParameter* are not null.

ConformanceConstraints: *ConformanceConstraint*s are defined on the *VariationPoint*s to ensure that *ConfigurableParameter*s are configured correctly with appropriate *Variant*s from their sets of *Variant*s [9, 10, 39]. In other words, *ConformanceConstraint*s ensure that *ConfigurationData* representing the *ConfigurationDecisions* conform to a set of pre-defined rules. Thus, these constraints eventually enable the configuration engineers to configure a valid *MemberProduct* from the *ProductLine*. *ConformanceConstraint*s are hard constraints, which can be *UserDefined* *DerivedFromSystemSpecifications*, or *DerivedFromModelingLanguage*, belonging to different phases (i.e., *Requirement*, *Design*, *Implementation*, *Testing*) of the development cycle (Figure B-10).

```
context SubseaControlSystem inv Exp-5:
217=< self.maxPressure <=725
```

Listing 4: An Example of ConformanceConstraint

Exp-5 in Listing 4 is a *ConformanceConstraint* from the SPS case study defined on class *SubseaControlSystem* (Figure B-3), stating that the *maxPressure* of a *SubseaControlSystem* can be great or equal to 217 bar(g), and less or equal to 725 bar(g).

ConsistencyConstraints: *ConsistencyConstraint*s are defined on the *VariationPoint*s to ensure that certain conditions are met across different artifacts (e.g., across software and hardware views, across different models) [32, 40, 41]. As shown in Figure B-10, *ConsistencyConstraint* is characterized by *atModel* and *atView*. *atModel* indicates whether constraint is defined on *VariationPoint*s belonging to same model (i.e., *IntraModel*) or different models (*InterModel*) whereas *atView* shows *VariationPoint*s belong to one *View* (i.e., *WithinView)* or multiple *View*s (i.e., *CrossView*). *ConsistencyConstraint*s are hard constraints, which can be either *UserDefined* or *DerivedFromSystemSpecifications*. Note that *ConsistencyConstraint*s ensure a certain property across different artifacts (e.g., two variation points belonging to same/different views/models) and *ConformanceConstraint*s focus on the correctness of *ConfigurationData* corresponding to the *ConfigurableParameter*s of a particular *VariationPoint*. Thus, it is possible that a *ConfigurableParameter* is configured with a variant that conforms to the *ConformanceConstraint*s but violates one or more *ConsistencyConstraint*s.

```
context XmasTree inv Exp-6:
self.subseaField->forAll(x:SubseaField|self.maxPressure <=(x.designPressure + (0.05*x.designPressure)))
```

Listing 5: An Example of ConsistencyConstraint

The constraint Exp-6 in Listing 5 is an example of *IntraModel CrossView ConsistencyConstraint* from the SPS case study (Figure B-3), as the constrained elements belong to two different *View*s (i.e., *maxPressure* and *designPressure* are software and hardware properties respectively) and same design model. It states that *maxPressure* of a *SubseaControlSystem* can be maximum of 5% more than *designPressure* of any *subseaField* of the *SubseaControlSystem*.

DecisionOrderingConstraints: *DecisionOrderingConstraint*s are hard constraints defined to enforce a particular configuration order for the *VariationPoint*s, which can be *UserDefined*, *DerivedFromSystemSpecifications*, or *DerivedFromModelingLanguage* (Figure B-10).

```
context subseaField inv Exp-7:
self.designPressure>5 implies self.xmasTree->forAll(x:XmasTree|x.waterDepth>400 and
x.installType=InstallationVesselType::Semisubmersible)
```

Listing 6: An Example of DecisionOrderingConstraint and DecisionInferenceConstraint

The constraint Exp-7 in Listing 6 has an implied order of configuring three *VariationPoint*s, i.e., *designPressure* of *subseaField* and *installType* and *waterDepth* of *XmasTree*, which can be derived from their dependencies (Figure B-3). Due to implied relationship we need to configure *designPressure VariationPoint* first and then *installType* and *waterDepth*. This will enable configuring the *InstallType VariationPoint* automatically.

DecisionInferenceConstraints: *DecisionInferenceConstraint*s are hard constraints defined on the *VariationPoint*s to infer *ConfigurationDecision*s automatically for one or more *VariationPoint*s [9, 16, 32]. Inferring *ConfigurationDecision*s is only possible when the *VariationPoint*s can be configured with only one *Variant* satisfying the *DecisionInferenceConstraint*s, otherwise, they need to be configured manually. *DecisionInferenceConstraint*s can be *UserDefined*, *DerivedFromSystemSpecifications*, or *DerivedFromModelingLanguage*, belonging to different phases of the development cycle (Figure B-10).

The constraint Exp-7 in Listing 6 is also a *DecisionInferenceConstraint*. If we have configured *designPressure* of *subseaField* with a value greater than 5 then *installType* can be configured automatically with *Semsubmersible*. We cannot configure the *waterDepth* as it has more than one *Variants* (i.e., all values greater than 400) satisfying the constraint.

OptimizationConstraints: *OptimizationConstraints* are soft constraints defined on *VariationPoints* to configure *ConfigurableParameters* with optimal *Variants* in terms of *optimizationMeasures* (e.g., cost, performance, energy consumption). An *OptimizationConstraint* can be defined on one *VariationPoint* (e.g., selecting a temperature sensor with the highest accuracy) or several *VariationPoints* (e.g., selecting different sensors with the lowest overall cost). *OptimizationConstraint* can be *UserDefined* or *DerivedFromSystemSpecifications*, which belong to different phases of the development cycle (Figure B-10).

```
context SubseaControlSystem inv Exp-8:
self.boreType=Set{BoreType::Compact,BoreType::FullBore}->sortedBy(x:BoreType |x.treePrice.cost)->first()
```

Listing 7: An Example of OptimizationConstraint

For example, in the case of the vertical deep-water tree (*VXT XmasTree*) technical *Stakeholders* are interested that tree should 10,000 feet in water (i.e., *waterDepth*=10000) and be able to operate under 350 F temperature (i.e., *maxTemperature*=350). They have a choice between *Compact* and *FullBore XmasTree* (Figure B-3). Both *Compact* and *FullBore XmasTree* support the technical specifications but *FullBore XmasTree* has a higher cost as compared to *Compact XmasTree*. There is an *OptimizationConstraint* enforced by business *Stakeholders* constraining the *VariationPoint boreType*, which implies selecting a low cost *XmasTree*. *VariationPoint boreType* can be configured with *Compact* or *FullBore* but *Compact* is an optimized configuration of *boreType VariationPoint*, which satisfies the *OptimizationConstraint* (i.e., low cost as shown in Listing 7).

5 Application Engineering of CPS Product Lines

This section discusses the third part of the framework (i.e., *ApplicationEngineering* in Figure B-1). Section 5.1. presents various functionalities of a *ConfigurationSolution*, followed by the relationships among functionalities and constraints in Section 5.2.

5.1 Functionalities of the Configuration Solutions

Based on the literature and our experience of conducting industrial research in the field of CPS PLE, we constructed a conceptual model for a *ConfigurationSolution*, as presented in Figure B-11. *ConfigurationSolution* has 14 functionalities: *DecisionInference*, *DecisionOrdering*, *RevertingDecision*, *WellFormednessChecking*, *ConformanceChecking*, *ConsistencyChecking*, *ResolvingViolation*, *CollaborativeConfiguration*, *ImpactAnalysis*, *ConflictDetection*, *ConstraintSelection*, *ConfigurationOptimization*, *RedundancyDetection*, and *IncompletenessDetection*. Note that in Figure B-11, we do not show the relationship among the functionalities, which we discuss in Section 5.2. In this section, we provide textual descriptions of the 14 functionalities, whereas their precise definitions are provided in Appendix D.

DecisionInference: During the *ConfigurationProcess*, various *ConfigurationDecisions* can be made automatically based on previously made *ConfigurationDecisions* and *DecisionInferenceConstraints* [32]. *DecisionInference* functionality enables such automatic inference of *ConfigurationDecisions* by evaluating and solving *DecisionInferenceConstraints*. It reduces the manual configuration effort and errors in *ConfigurationData* caused by human [12, 32] and consequently improves the overall productivity of the *ConfigurationProcess*.

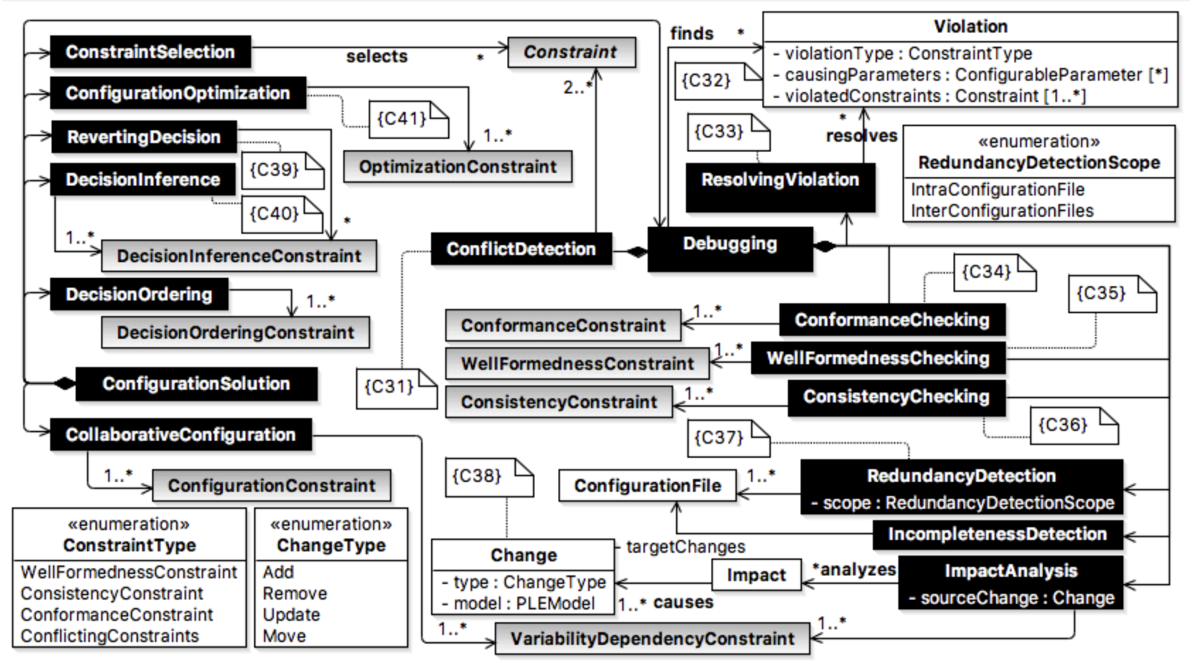


Figure B-11. A conceptual model for the functionalities of a configuration solution[13]

DecisionOrdering: During the *ConfigurationProcess*, the *DecisionOrdering* functionality guides the users in which order the *ConfigurationDecision*s should be made to derive a valid *MemberProduct* from a *ProductLine*. It makes use of *DecisionOrderingConstraint*s to suggest an optimal order of *ConfigurationDecision*s such that the total number of manual *ConfigurationDecision*s is minimized and conflict among *ConfigurationDecision*s can be avoided [42]. Thus, this will improve the productivity of *ConfigurationProcess* and consequently reduce the configuration cost.

RevertingDecision: In practice, users often modify previously made *ConfigurationDecision*s during a *ConfigurationProcess*. Functionality *RevertingDecision* enables users to make these changes on any part of the configuration history. This is not trivial as several *ConfigurationDecision*s are made automatically using *DecisionInference*, thus, rolling back a *ConfigurationDecision* requires re-evaluating and re-solving *DecisionInferenceConstraint*s without violating other constraints (e.g., *ConsistencyConstraint*s).

WellFormednessChecking: *WellFormednessChecking* ensures the correctness of *PLEModel*s by checking their well-formedness against the abstract syntax of *ModelingLanguage* and additionally defined *WellFormednessConstraint*s. As discussed earlier in Section 4.3, modeling tools ensure the well-formedness of *BaseModel* and *VariabilityModel*, however, the well-formedness of *ResolutionModel* needs to be ensured during the *ConfigurationProcess*, as it created by instantiating the *VariationPoint*s and configuring them by selecting/assigning the *Variant*s. Thus, the *ConfigurationSolution* needs to provide *WellFormednessChecking* for *ResolutionModel*s.

ConformanceChecking: *ConformanceChecking* ensures the correctness of *ConfigurationDecision*s made to derive a valid *MemberProduct* from a *ProductLine* according to the user requirements. To

[13] C31-C41 are OCL constraints provided in Appendix B.

do so, it checks whether each configured *ConfigurableParameter* is configured correctly with a valid *Variant* while conforming to a set of predefined *ConformanceConstraint*s [9, 39].

ConsistencyChecking: *ConsistencyChecking* verifies certain conditions and properties across various artifacts (i.e., *PLEModel*s and *View*s) by evaluating a set of pre-defined *ConsistencyConstraint*s with four different scopes (i.e., *IntraModel WithinView, IntraModel CrossView*, *InterModel WithinView*, and *InterModel CrossView*).

CollaborativeConfiguration: CPSs are composed of several subsystems, which involve components from diverse disciplines (e.g., Software. Mechanic, Electronics). Typically, these subsystems and components are developed and configured by different domain experts. *CollaborativeConfiguration* allows users to configure the system part by part and coordinates the *ConfigurationProcess* to derive valid *MemberProduct*s from the *ProductLine* in a multi-stage and multi-step manner [14, 37]. *CollaborativeConfiguration* has three main tasks [44]: 1) splitting the *ConfigurableParameter*s into different groups and assigning them *ConfigurationStage*s and *ConfigurationStep*s such that a group of related *ConfigurableParameter*s can be configured in a *ConfigurationStep* within a *ConfigurationStage* by specific *Stakeholder*s, 2) performing multiple *ConfigurationDecisions* to interactively configure the *ConfigurableParameter*s in each *ConfigurationStep*, and 3) merging *ConfigurationData* from different *ConfigurationStep*s and *ConfigurationStage*s while maintaining the consistency [45].

ConflictDetection: Since the automation of configuration heavily relies on the constraints, it is necessary to ensure the quality of the constraints by identifying the conflicts among different constraints before actual *ConfigurationProcess* starts. Conflicting constraints may lead to ill-formed, incomplete, invalid, and inconsistent *MemberProduc*ts. *ConflictDetection* functionality finds conflicting constraints in a given set of constraints by checking if two or more hard constraints are constraining a *ModelElement* (e.g., *VariationPoint*) that can never be true at the same time.

ResolvingViolation: *WellFormednessConstraint*s, *ConformanceChecking*, *ConsistencyChecking*, and *ConflictDetection* can identify four types of *Violation*s due to violation of *WellFormednessConstraint*, *ConformanceConstraint*, *ConsistencyConstraint*, and *ConflictingConstraints* (Figure B-11). *ResolvingViolation* functionality resolves these *Violation*s automatically where possible [10]. If certain *Violation*s cannot be resolved automatically (e.g., *ConflictingConstraints*), *ResolvingViolation* can guide the users to fix them manually.

ImpactAnalysis: *ImpactAnalysis* is crucial as it helps the *Stakeholder*s to assess the consequence of making certain *Change*s into the *ProductLine*, which will consequently help the *Stakeholder*s in decision making [12]. In Figure B-11, *Impact* represents the consequence of a *sourceChange*, which can be measured in terms of *targetChanges*. A *Change* can be of four types (i.e., *Add*, *Remove*, *Update*, and *Move*), which can occur in a *PLEModel*. *ImpactAnalysis* functionality checks the *Impact* of a *sourceChange* corresponding to a *ModelElement* in a *PLEModel* on other *ModelElement*s in the same or/and different *PLEModel*s.

ConstraintSelection: Usually *ProductLine*s have a large number of constraints and only a subset of constraints is required at a given time. For example, to determine the configuration order of two *VariationPoint*s vp_1 and vp_2, we need only a subset of *DecisionOrderingConstraint*s constraining at least one of vp_1 and vp_2. Checking all the *DecisionOrderingConstraint*s is a wastage of resources (e.g., memory and computation) and can decrease the efficiency and productivity of the *ConfigurationProcess*. Similarly, a subset of constraints for a *MemberProduct* needs to be selected to configure the *ConfigurableParameters* at deployment or/and post-deployment time.

ConstraintSelection functionality selects a subset of constraints related to specific *VariationPoint*s and *ConfigurableParameters* to improve the efficiency and productivity of the *ConfigurationProcess*.

ConfigurationOptimization: In practice, valid configurations do not always mean optimized configurations. For example, selecting a temperature sensor that meets the technical user requirements (e.g., certain temperature range) is a valid configuration but not necessarily optimal in terms of energy consumption. Often, configurations need to be optimized in terms of *optimizationMeasure*s such as cost, performance, and energy consumption according to pre-defined *OptimizationConstraint*s. *ConfigurationOptimization* functionality selects optimal *Variant*s in terms of *optimizationMeasure*s from a set of valid *Variant*s corresponding to a set of *ConfigurableParameter*s such that the maximum *OptimizationConstraint*s are satisfied.

RedundancyDetection: *RedundancyDetection* checks whether multiple *ConfigurationData* are associated with a *ConfigurableParameter* within two scopes, i.e., *IntraConfigurationFile* and *InterConfigurationFile*s. In the case of *IntraConfigurationFile*, it checks within one *ConfigurationFile* representing a *ResolutionModel* of a *MemberProduct* whereas in the case of *InterConfigurationFile*s it checks multiple *ConfigurationFile*s representing a *ResolutionModel* of a *MemberProduct*. This functionality is particularly useful when existing *ConfigurationFile*s are used to derive a *MemberProduct* by completing or modifying the existing configurations. Redundant *ConfigurationData* may lead to inconsistent and invalid *MemberProduct*s.

IncompletenessDetection: *IncompletenessDetection* checks whether a *ConfigurationFile* has *ConfigurationData* with null values for certain *ConfigurableParameters* or un-configured *ConfigurableParameters*, i.e., a *Variant* is not assigned. *ConfigurationFiles* with incomplete *ConfigurationData* can lead to incomplete, inconsistent, and invalid *MemberProduct*s.

5.2 Relationships among the Functionalities of Configuration Solution and Constraints

As shown in Table B-4 and Figure B-11, all the functionalities of a *ConfigurationSolution* except *RedundancyDetection* and *IncompletenessDetection* need to perform different operations (i.e., query, evaluate, and solve [46]) on the different types of constraints. *DecisionInference* and *ConfigurationOptimization* query, evaluate, and solve *DecisionInferenceConstraint*s and *OptimizationConstraints* respectively. *DecisionOrdering*, *WellFormednessChecking*, *ConformanceChecking*, *ConsistencyChecking*, and *ImpactAnalysis* query and/or evaluate the *DecisionOrderingConstraint*s, *WellFormednessConstraint*s, *ConformanceConstraint*s, *ConsistencyConstraint*s, and *VariabilityDependencyConstraint*s respectively. Similarly, *RevertingDecision* also queries and/or evaluates *DecisionInferenceConstraint*s for reverting the automatically inferred *ConfigurationDecision*s. *ResolvingViolation* queries and/or evaluates *WellFormednessConstraint*s and queries, evaluates, and/or solves *ConformanceConstraint*s and *ConsistencyConstraint*s causing inconsistencies whereas *CollaborativeConfiguration* solves and/or queries and evaluates *ConfigurationConstraint*s and *VariabilityDependencyConstraint*s. *ConflictDetection* queries, evaluates, and solves to find the conflicts among different hard constraints whereas *ConstraintSelection* searches the constraints and selects a relevant subset of constraints.

Table B-5 shows the relationships among different functionalities of the *ConfigurationSolution* where a particular value indicates if the functionality presented in a row uses the functionality presented in the column. *DecisionOrdering*, *RevertingDecision*, and *CollaborativeConfiguration* are related to *DecisionInference*. *DecisionInference* can be used to get the significance (or importance) of a

ConfigurationDecision that shows the impact of configuring a *ConfigurableParameter* on the automated resolution of other *ConfigurableParameter*s based on *DecisionInferenceConstraint*s. Hence, *DecisionOrdering* is related to *DecisionInference* as the importance degree of *ConfigurationDecision*s can be used to determine an optimal order. *RevertingDecision* is related to *DecisionInference*, as reverting a *ConfigurationDecision* may also require undoing its subsequent automatically inferred *ConfigurationDecision*s. *CollaborativeConfiguration* uses *DecisionInference*, *DecisionOrdering*, and *RevertingDecision* to infer the *ConfigurationDecision*s automatically based on *DecisionInferenceConstraint*s and *ConfigurationDecision* propagation, to get an optimal order of *ConfigurationDecision*s, and to roll back certain *ConfigurationDecision*s.

Table B-4. Applicability of constraints*

F/C	C_C	C_{VD}	C_{WF}	C_{CF}	C_{CS}	C_{DO}	C_{DI}	C_{OP}
DecisionInference	-	-	-	-	-	-	Q, E, & S	-
DecisionOrdering	-	-	-	-	-	Q & E	-	-
RevertingDecision	-	-	-	-	-	-	Q & E	-
WellFormednessChecking	-	-	Q & E	-	-	-	-	-
ConformanceChecking	-	-	-	Q & E	-	-	-	-
ConsistencyChecking	-	-	-	-	Q & E	-	-	-
ResolvingViolation	-	-	Q & E	Q, E, & S	Q, E, & S	-	-	-
CollaborativeConfiguration	Q, E, & S	Q & E	These are used indirectly because CollaborativeConfiguration uses other functionalities (e.g., *WellFormednessChecking*, *ConformanceChecking*).					
ImpactAnalysis	-	Q & E	-	-	-	-	-	-
ConflictDetection	Q, E, & S							-
ConstraintSelection	Select a subset							
ConfigurationOptimization	-	-	-	-	-	-	-	Q, E, & S
RedundancyDetection	-	-	-	-	-	-	-	-
IncompletenessDetection	-	-	-	-	-	-	-	-

* C_C =*ConfigurationConstraint*s, C_{VD} =*VariabilityDependencyConstraint*s, C_{WF} =*WellFormednessConstraint*s, C_{CF} =*ConformanceConstraint*s, C_{CS} =*ConsistencyConstraint*s, C_{DO} =*DecisionOrderingConstraint*s, C_{DI} =*DecisionInferenceConstraint*s, C_{OP} =*OptimizationConstraint*s, Q=Query, E= Evaluate, S=Solve

DecisionInference and *CollaborativeConfiguration* use *WellFormednessChecking*, *ConformanceChecking*, *ConsistencyChecking*, *ImpactAnalysis*, and *ResolvingViolation* to ensure that *VariationPoint*s are correctly instantiated and configured. *ConfigurationOptimization* also uses *ConformanceChecking*, *ConsistencyChecking*, *ImpactAnalysis*, and *ResolvingViolation* to ensure that none of the *ConformanceConstraint*s and/or *ConsistencyConstraint*s is violated due to *ConfigurationOptimization*. Similarly, *RevertingDecision* can also use *ConsistencyChecking*, *ImpactAnalysis*, and *ResolvingViolation*, as reverting a *ConfigurationDecision* may lead to inconsistent configurations, which need to be fixed using *ResolvingViolation*. *ConstraintSelection* is used by all the functionalities except *RedundancyDetection* and *IncompletenessDetection*, for selecting a subset of constraints. *ConflictDetection*, *ConfigurationOptimization*, *RedundancyDetection*, and *IncompletenessDetection* are used by only *CollaborativeConfiguration* to detect the conflicts among constraints at the start of the *ConfigurationProcess*, optimize the configuration, and detect redundancy and incompleteness in *ConfigurationFile*s. Please note that specifying the exact order in which a functionality uses others is difficult to predict as it depends on the methodology to select the optimized order of functionalities and the context in which they are being used. For example, after every

Table B-5. Use relations among functionalities

Functionality	DI	DO	RD	WFC	CFC	CSC	RV	CC	IA	CD	CS	CO	RD	ID
DecisionInference (DI)	NA	-	-	Yes	Yes	Yes	Yes	-	Yes	-	Yes	-	-	-
DecisionOrdering (DO)	Yes	NA	-	-	-	-	-	-	-	-	Yes	-	-	-
RevertingDecision (RD)	Yes	-	NA	-	-	Yes	Yes	-	Yes	-	Yes	-	-	-
WellFormednessChecking (WFC)	-	-	-	NA	-	-	-	-	-	-	Yes	-	-	-
ConformanceChecking (CFC)	-	-	-	-	NA	-	-	-	-	-	Yes	-	-	-
ConsistencyChecking (CSC)	-	-	-	-	-	NA	Yes	-	-	-	Yes	-	-	-
ResolvingViolation (RV)	-	-	-	-	-	Yes	NA	-	Yes	-	Yes	-	-	-
CollaborativeConfiguration (CC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	NA	Yes	Yes	Yes	Yes	Yes	Yes
ImpactAnalysis (IA)	-	-	-	-	-	-	-	-	NA	-	Yes	-	-	-
ConflictDetection (CD)	-	-	-	-	-	-	-	-	-	NA	Yes	-	-	-
ConstraintSelection (CS)	-	-	-	-	-	-	-	-	-	-	NA	-	-	-
ConfigurationOptimization (CO)	-	-	-	-	Yes	Yes	Yes	-	Yes	-	Yes	NA	-	-
RedundancyDetection (RD)	-	-	-	-	-	-	-	-	-	-	-	-	NA	-
IncompletenessDetection (ID)	-	-	-	-	-	-	-	-	-	-	-	-	-	NA

* Yes= functionality in a row may use the functionality in the column, “-”= two functionalities are not related, NA=not applicable

ConfigurationDecision, *ConformanceChecking* and *ConsistencyChecking* should be invoked, however in which order it depends on the methodology used to implement the *ConfigurationSolution*. Similarly, what functionality should be used next depends on if an inconsistency is found or not.

6 Evaluation

Section 6.1 presents the evaluation design followed by evaluation results in Section 6.2. We provide a discussion based on the evaluation results in Section 6.3 and threats to validity in Section 6.4.

6.1 Evaluation Design

Section 6.1.1 presents the overall goal of the evaluation and formulates research questions. Section 6.1.2 presents the evaluation tasks along with the evaluation metrics corresponding to the research questions defined in Section 6.1.1.

6.1.1 Research Questions

The overall objective of the evaluation is to investigate the capabilities of the framework in terms of providing support for domain engineering and application engineering of CPS PLs. More specifically, we intend to assess if the framework provides the support for capturing and managing commonalities, variabilities, and constraints in the domain engineering phase as well as the support for automation of configuration in the application engineering phase. The objective can be achieved by answering the following research questions:

RQ1. To what extent the framework can capture the variabilities of CPS PLs based on the selected case studies?

RQ2. To what extent the framework can capture the constraints for CPS PLs based on the selected case studies?

RQ3. To what extent the framework is complete for providing different views for CPS PLs based on the selected case studies?

RQ4. To what extent the framework is complete for providing support for automation of configuration based on existing literature?

The first three research questions (RQ1-RQ3) are related to the domain engineering and the fourth (RQ4) is related to the application engineering of CPS PLs. With RQ1 and RQ2, we assess whether the framework fulfills the requirements of the case studies (Section 2) for capturing the variabilities and constraints. RQ3 assesses if the framework provides enough *View* types for managing the *ModelElement*s (e.g., *VariationPoint*s, *Variant*s, *Constraint*s, *Interaction*s) for the case studies. RQ4 aims to assess to what extent the framework provides support for different types of automation for *ProductConfiguration* based on existing literature (both tools and techniques).

6.1.2 Evaluation Tasks and Metrics

As shown in Table B-6, we designed four tasks (T_1-T_4) for addressing RQ1-RQ4. Table B-6 also shows 19 evaluation metrics and their definitions used to answer the research questions (RQ1-RQ4). To answer RQ1, we report the total number of VPs in a case study, the percentage of VPs that can be captured using *VariationPoint* types provided in the framework (Section 4.2.2), and the percentage of *VariationPoint* types required to capture all the VPs in the case study. Similarly, for

RQ2, we present the total number of constraints in a case study, the percentage of constraints that can be captured using *Constraint* types provided in the framework (Section 4.3), and the percentage of *Constraint* types required to capture all the constraints in the case study. For RQ3, we also report the total number of views required by a case study, the percentage of views supported by the framework, and the percentage of *View* types required to model all the views in the case study. Note that corresponding to RQ1-RQ3, we report the above-mentioned metrics for the three case studies individually as well as combined. To answer RQ4, we calculate the percentage of functionalities covered by the existing literature (i.e., tools and techniques).

It is essential to mention the sources used to collect data for calculating the metrics corresponding to RQ1-RQ3. For RQ1, the data were collected from two sources: 1) *VariabilityModel*s for VCS, SPS, and MHS case studies that we developed earlier and 2) documents available on the websites of our industry partners (e.g., for the VCS case study). The *VariabilityModel*s were developed based on the domain knowledge gained by reading documents. Similarly, for RQ2, some of the data were directly collected by counting the number of constraints specified using OCL as part of the *PLEModel*s (e.g., feature models, architecture and design level variability models) that we developed earlier while others were collected as English sentences from publicly available documents (i.e., system specifications, user manuals) and were not formally specified using any particular *ConstraintSpecificationLanguage*. Moreover, some of the data for constraints were also directly collected from the documents provided by the industry partners (e.g., for VCS). For RQ3, we collected data based on types and disciplines of the *ModelElement*s available in the case studies.

Table B-6. Evaluation tasks and metrics*

RQ	Task	Metric	Metric Definition
1	*T₁*: Counting the total number of VPs, VPs that can be captured using the *VariationPoint* types in the framework, and the *VariationPoint* types required to capture all the VPs for the case studies and calculating the values for the corresponding metrics.	VP_i	The total number of VPs in i^{th} case study.
		VPC_i	The percentage of VPs captured in the i^{th} case study using the *VariationPoint* types provided by the framework: $VPC_i = \frac{\#\ of\ VPs\ that\ can\ be\ captured\ for\ i^{th}\ case\ study}{VP_i}$
		$VPTR_i$	The percentage of *VariationPoint* types required by the i^{th} case study: $VPTR_i = \frac{\#\ of\ VariationPoint\ types\ required\ by\ i^{th}\ case\ study}{Total\ \#\ of\ VariationPoint\ types\ in\ the\ framework}$
		VP	The total number of VPs in the three case studies.
		VPC	The percentage of VPs captured in the three case studies using the *VariationPoint* types provided by the framework: $VPC = \frac{\#\ of\ VPs\ that\ can\ be\ captured\ for\ three\ case\ studies}{VP}$
		$VPTR$	The percentage of *VariationPoint* types required by the three case studies: $VPTR = \frac{\lvert\{\bigcup_{i=1}^{i=3} VariationPoint\ types\ required\ by\ i^{th}\ case\ study\}\rvert}{Total\ \#\ of\ VariationPoint\ types\ in\ the\ framework}$
2	*T₂*: Counting the total number of constraints, constraints that can be captured using the *Constraint* types in the framework, and the *Constraint* types required to capture all the constraints for the case studies and calculating the values for the corresponding metrics.	C_i	The total number of constraints in the i^{th} case study.
		CC_i	The percentage of constraints captured for the i^{th} case study using *Constraint* types provided by the framework: $CC_i = \frac{\#\ of\ \text{constraints}\ that\ can\ be\ captured\ for\ i^{th}\ case\ study}{C_i}$
		CTR_i	The percentage of *Constraint* types required by the i^{th} case study: $CTR_i = \frac{\#\ of\ Constraint\ types\ required\ by\ i^{th}\ case\ study}{Total\ \#\ of\ Constraint\ types\ in\ the\ framework}$
		C	The total number of constraints in the three case studies.
		CC	The percentage of constraints captured in the three case studies using the *Constraint* types provided by the framework:

			$CC = \frac{\#\ of\ \text{constraints}\ that\ can\ be\ captured\ for\ three\ case\ studies}{C}$
		CTR	The percentage of *Constraint* types required by the three case studies: $CTR = \frac{\lvert\{\bigcup_{i=1}^{i=3} Constraint\ types\ required\ by\ i^{th}\ case\ study\}\rvert}{Total\ \#\ of\ Constraint\ types\ in\ the\ framework}$
3	*T₃*: Counting the total views required, views that can be modeled using the *View* types in the framework, and the *View* types required to model all the views for the case studies and calculating the values for the corresponding metrics.	V_i	The total number of views required in the i^{th} case study.
		VM_i	The percentage of views modeled for the i^{th} case study with the *View* types provided by the framework: $VM_i = \frac{\#\ of\ \text{views}\ that\ can\ be\ modeled\ for\ i^{th}\ case\ study}{V_i}$
		VTR_i	The percentage of *View* types required by the i^{th} case study: $VTR_i = \frac{\#\ of\ View\ types\ required\ by\ i^{th}\ case\ study}{Total\ \#\ of\ View\ types\ in\ the\ framework}$
		V	The total number of views required in the three case studies.
		VM	The percentage of views modeled for the three case studies using the *View* types provided by the framework: $VM = \frac{\#\ of\ View\ types\ required\ by\ three\ case\ studies}{V}$
		VTR	The percentage of *View* types required by the three case studies: $VTR = \frac{\lvert\{\bigcup_{i=1}^{i=3} View\ types\ required\ by\ i^{th}\ case\ study\}\rvert}{Total\ \#\ of\ View\ types\ in\ the\ framework}$
4	*T₄*: Counting the number of functionalities covered by the literature and calculating the value for the corresponding metric.	FC	Percentage of functionalities covered by the literature: $FC = \frac{\#\ of\ functionalities\ covered\ by\ literature}{Total\ \#\ of\ functionalities\ in\ the\ framework}$

* All the metrics (except FC) with the subscript "i" are for individual case studies whereas without subscript are for the three case studies combined. VP= Total number of VPs, VPC= The percentage of VPs can be captured, VPTR= The percentage of *VariationPoint* types required, C= Total number of constraints, CC= The percentage of constraints can be captured, CTR= The percentage of *Constraint* types required, V= Total number of views, VM= The percentage of views can be modeled, VTR= The percentage of *View* types required, FC= Functionality coverage.

6.2 Evaluation Results

In this section, we present the results of our evaluation and answer the research questions (RQ1-RQ4) defined in Section 6.1.1.

6.2.1 Results for RQ1

Table B-7 summarizes the results of RQ1 based on the evaluation metrics defined in Table B-6. As shown in Table B-7, the three case studies MHS, VCS, and SPS have 476, 1507, and 178 VPs respectively and all of these VPs can be captured using the CPS-specific VP types provided by the framework (Section 4.2.2). Overall the three case studies have 2161 VPs in total. The MHS case study requires all the CPS-specific VP types to capture its VPs (i.e., $VPTR_1$=100%) whereas the other two case studies (i.e., SPS and VCS) require only 12 out of 16 CPS-specific VP types (i.e., $VPTR_2$=75% and $VPTR_3$=75%).

The distribution of VPs across different CPS-specific VP types for the case studies are given in Table B-8. From Table B-8, one can notice that for PropertyChoice-VP, BinaryChoice-VP, Descriptive-VP, and DiscreteMeasurement-VP, we have significantly more VPs than other VP types (Table B-8), which is expected as a CPS has a large number of properties (i.e., *PhysicalProperty* and *ComponentProperty*). Furthermore, it is worth mentioning that not all the case studies have VPs corresponding to all CPS-specific VP types. This can be explained by the fact that all CPSs do not have the same business requirements, complexity, and size. For example, the VCS case study does not have continuous and compound properties; therefore, it does not have VPs corresponding to Continuous-VP and Compound-VP. Similarly, SPS does not have different interaction mechanisms; therefore, it does not need InteractionChoice-VP type. Based on the results of RQ1 (Table B-7 and Table B-8), all the VPs of the case studies can be captured using the CPS-specific VP types provided in the framework. This indicates that the framework has all the necessary VP types required to capture the variabilities of CPS PLs.

Table B-7. Evaluation results for RQ1

Case Study/Metric	**Individual Case Study**			**Overall (Combined All the Three Case Studies)**		
	VP_i	VPC_i	$VPTR_i$	VP	VPC	$VPTR$
MHS	476	100%	100%	2161	100%	100%
VCS	1507	100%	75%			
SPS	178	100%	75%			

Table B-8. The distribution of VPs across CPS-specific VP types for three case studies

CPS-Specific VP type	**Number of VPs in the Case Studies**			
	MHS	**VCS**	**SPS**	**All**
Descriptive-VP	34	206	8	248
DiscreteMeasurement-VP	23	146	37	206
ContinuousMeasurement-VP	51	0	3	54
ComponentCardinality-VP	42	25	7	74
ComponentCollectionBoundary-VP	42	25	7	74
MeasurementPrecision-VP	2	0	4	6
BinaryChoice-VP	3	554	3	560
PropertyChoice-VP	82	454	31	567
ComponentChoice-VP	12	62	13	87
TopologyChoice-VP	9	1	0	10
AllocationChoice-VP	5	3	0	8
InteractionChoice-VP	1	5	0	6
MeasurementUnitChoice-VP	59	0	28	87
ConstraintSelection-VP	5	1	0	6
ComponentSelection-VP	42	25	7	74
Multipart/Compound-VP	64	0	30	94
Total	**476**	**1507**	**178**	**2161**

6.2.2 Results for RQ2

In Table B-9, we present the results of RQ2 based on the evaluation metrics defined in Table B-6. As one can observe from Table B-9 that case studies MHS, VCS, and SPS have 763, 2897, and 283 constraints respectively and all of them can be captured with the *Constraint* types of the framework (Section 4.3). Overall, all the case studies have 3943 constraints and require 6 out of 7 *Constraint* types to capture all the constraints (i.e., CTR_i=86%).

Table B-9. Evaluation results for RQ2

Case Study/Metric	**Individual Case Study**			**Overall (Combined All the Three Case Studies)**		
	C_i	CC_i	CTR_i	C	CC	CTR
MHS	763	100%	86%	3943	100%	86%
VCS	2897	100%	86%			
SPS	283	100%	86%			

In Table B-10, we present the distribution of constraints across different *Constraint* types for the selected case studies. As shown in Table B-10, corresponding to each *Constraint* type except *OptimizationConstraint* and *WellFormednessConstraint*s, there exist one or more constraints in all three case studies. *WellFormednessConstraint*s are not specific to any case study, indeed they are defined as part of modeling methodologies and *ConfigurationSolution*s for facilitating *WellFormednessChecking* to ensure the well-formedness of *PLEModel*s (e.g., *ResolutionModel*). The importance of *WellFormednessConstraint*s can be depicted by the existing tools and techniques supporting *WellFormednessChecking* [35, 47-49]. The case studies do not contain any instance of *OptimizationConstraint*s, as usually they are defined by the business *Stakeholder*s based on the user requirements, standards, and rules and regulations enforced by the governing bodies and we do not have such information. However, these constraints are necessary for *ConfigurationOptimization*, which is supported by existing *ConfigurationSolution*s [50-52]. Based on the results of RQ2 (Table B-9 and Table B-10) and the above discussion, we can conclude that the framework has all the essential *Constraint* types required to capture the constraints for CPS PLs.

Table B-10. The distribution of constraints across CPS-specific VP types for the three case studies

Constraint Type	**Number of Constraints in the Case Studies**			
	MHS	**VCS**	**SPS**	**All**
ConfigurationConstraint	74	876	41	991
VariabilityDependencyConstraint	227	424	49	700
WellFormednessConstraint	WellFormednessConstraints are not specific to the case studies			
ConformanceConstraint	108	1129	49	1286
ConsistencyConstraint	94	424	54	572
DecisionOrderingConstraint	166	22	45	233
DecisionInferenceConstraint	94	22	45	161
OptimizationConstraint	0	0	0	0
Total	**763**	**2897**	**283**	**3943**

6.2.3 Results for RQ3

Table B-11 presents the results of RQ3 based on the evaluation metrics defined in Table B-6. As shown in Table B-11, the MHS case study requires 14 views whereas VCS and SPS both require 13 views. One can observe from Table B-11 that to model all the views, MHS, VCS, and SPS require 82%, 76%, and 76% of *View* types respectively. In total the three case studies require 40 views and all of them are supported by the *View* types provided by the framework (Section 4.1).

Table B-11. Evaluation results for RQ3

Case Study/Metric	**Individual Case Study**			**Overall (Combined All the Three Case Studies)**		
	V_i	VM_i	VTR_i	V	VM	VTR
MHS	14	100%	82%	40	100%	100%
VCS	13	100%	76%			
SPS	13	100%	76%			

Furthermore, in Table B-12, we present the distribution of views across *View* type for the three case studies. Corresponding to all the *View* types, we have one or more views required by the case studies (Table B-12). Note that not all the case studies require all *View* types because all CPSs do not have elements from all the domains (e.g., Hydraulics, Mechanics, Electronics). Thus, depending on the constitution of a CPS, a case study might require various *View* types. For example, VCS does not have elements from the Mechanics and Hydraulics domains, therefore, it does not require *View* types related to these domains (Table B-12). As per the results of RQ2 (Table B-11), all the views required by the case studies are supported by the framework (Table B-11), which suggests that our framework has all the necessary *View* types.

Table B-12. The distribution of views across view types for three case studies

View Type	**Number of Views Required by the Case Studies**			
	MHS	**VCS**	**SPS**	**All**
ContextView	1	1	1	3
SoftwareView	1	1	1	3
InteractionView	1	1	0	2
AllocationView	1	1	1	3
MechanicalView	1	0	1	2
ElectricalView	1	1	1	3
ElectronicsView	1	1	1	3
HydraulicsView	0	0	1	1
ContextVariabilityView	1	1	1	3
ApplicationVariabilityView	0	1	0	1
SoftwareVariabilityView	1	1	1	3
InteractionVariabilityView	1	1	0	2
AllocationVariabilityView	1	1	1	3
MechanicalVariabilityView	1	0	0	1
ElectricalVariabilityView	1	1	1	3
ElectronicsVariabilityView	1	1	1	3
HydraulicsVariabilityView	0	0	1	1
Metric Value	**14**	**13**	**13**	**40**

6.2.4 Results for RQ4

To answer RQ4, we validate the functionalities of *ConfigurationSolution* (Section 5.1) based on existing literature on the automation of configuration. For this, we reviewed 11 mostly-reported configuration tools: Pure::Variants [53], DOPLER [50], Covamof [54], SPLOT [55], Kumbang Configurator [47], FMP [48, 56], Quaestio [49], Zen-Configurator [57], FeatureID [58], C2O Configurator [59], and Gears Tool [60]. Several of these tools (e.g., DOPLER [50], Gears Tool [60], Zen-Configurator [57] that is developed by the authors of this paper) are used in the context of CPS PLE such as communication systems, intelligent traffic systems, industrial automation systems, and aerospace industry. In addition to the above-mentioned tools, we have also referred to the research papers presenting approaches to facilitate different functionalities of an automated configuration solution. Most of these approaches are implemented as part of the above-mentioned 11 tools. Table B-13 summarize the results of RQ4.

Table B-13. Existing tools and techniques related to automation of configuration (RQ4) *

Tool/Technique	**Functionality**													
	DI	**DO**	**RD**	***WFC***	**CFC**	***CSC***	**RV**	**CC**	**IA**	**CD**	**CS**	**CO**	**RD**	**ID**
Configuration Tools														
Pure::Variants [53]	☑	-	☑	-	☑	☑	☑	☑	-	-	-	-	-	-
DOPLER [50]	☑	☑	☑	-	☑	☑	☑	☑	☑	-	☑	☑	-	-
Covamof [54]	☑	☑	-	-	☑	☑	-	-	-	-	-	-	-	-
SPLOT [55]	☑	-	☑	-	☑	☑	☑	-	☑	-	-	-	-	☑
Kumbang Configurator [47]	☑	-	☑	☑	☑	☑	-	-	-	-	-	-	-	☑
FMP [48]	☑	-	☑	☑	☑	☑	☑	-	☑	-	-	-	-	-
Quaestio [49]	☑	☑	☑	☑	☑	☑	-	-	☑	☑	-	-	-	-
Zen-Configurator [57]	☑	☑	-	-	☑	☑	☑	-		-	☑	-	-	-
FeatureID [58]	☑	-	☑	-	☑	☑	-	-	-	-	-	-	-	

C2O Configurator [59]	☑	☑	-	-	☑	☑	☑	-	☑	-	-	-	-	-
Gears Tool [60]	Products are configured automatically based on feature profiles.		-	☑	☑	-	☑	-	-	-	-	-	-	-
Existing Approaches Related to Automation of Configuration														
Marcílio et al. [44, 45, 61]	-	-	-	-	-	-	-	☑	-	-	-	-	-	-
Zhou et al. [52]	-	-	-	-	-	-	-	-	-	-	-	☑	-	-
Alférez et al. [62]	-	-	-	-	-	☑	-	-	-	-	-	-	-	-
Heider et al. [63]	☑	-	-	-	-	-	☑	-	-	-	-	-	-	-
Yue et al. [64]	☑	☑	-	-	-	-	-	-	-	-	-	-	-	-
Lu et al. [9, 39]	-	-	-	-	☑	-	-	-	-	-	-	-	-	-
Lu et al. [10]	-	-	-	-	-	-	☑	-	-	-	-	-	-	-
Vierhauser et al. [40]	-	-	-	-	-	☑	-	-	-	-	☑	-	-	-
Rabiser et al. [65]	-	-	-	-	-	-	-	☑	-	-	-	-	-	-
Lettner et al. [51]	-	-	-	-	-	-	-	-	-	-	-	☑	-	-
Maszo et al. [66]	-	-	-	-	☑	☑	-	-	-	-	-	-	-	-
Heider et al. [67]	-	-	-	-	-	-	-	-	☑	-	-	-	-	-
Czarnecki et al. [35]	-	-	-	☑	-	-	-	-	-	-	-	-	-	-
Hwan et al. [68]	-	-	-	-	-	☑	-	-	-	-	-	-	-	-
Heidenreich [43]	-	-	-	☑	-	-	-	-	-	-	-	-	-	-
Metric Value (FC)	**92%**													

***☑= Partially/Fully Supported, "-"= Not Supported,** DI= *DecisionInference,* DO= *DecisionOrdering,* RD= *RevertingDecision,* WFC= *WellFormednessChecking,* CFC= *ConformanceChecking,* CSC= *ConsistencyChecking,* RV= *ResolvingViolation,* CC= *CollaborativeConfiguration,* IA= *ImpactAnalysis,* CD= *ConflictDetection,* CS= *ConstraintSelection,* CO= *ConfigurationOptimization,* RD= *RedundancyDetection,* ID= *IncompletenessDetection,* FC= Functionality coverage.

DecisionInference: As shown in Table B-13, most of the configuration tools (i.e., Pure::Variants [53], DOPLER [50, 63, 66, 69], Covamof [54, 70], SPLOT [55], Kumbang Configurator [47, 71], FMP [48], Quaestio [49], Zen-Configurator [57, 72], FeatureID [58], and C2O Configurator [59]) provide support of *DecisionInference* for making automatic *ConfigurationDecision*s based on *DecisionInferenceConstraint*s. To provide support for *DecisionInference*, these tools use various solvers (e.g., SAT Solvers, SModelS, and *lparse*). Moreover, some configuration tools (e.g., DOPLER [50]) reuse previously made *ConfigurationDecision*s to derive new products, in case the variability model is evolved (e.g., new variabilities are introduced) over the time [63]. Similarly, Yue et al. [64] proposed a search-based approach in which *ConfigurationDecision*s are performed in an optimal order such that maximum decisions can be inferred, which is implemented in Zen-Configurator.

DecisionOrdering: Table B-13 shows that five configuration tools DOPLER [50, 73], Covamof [54, 70], Quaestio [49], Zen-Configurator [57, 64], and C2O Configurator [59] provide support of *DecisionOrdering* to allow the users to make *ConfigurationDecision*s in an optimized manner. These tools provide support for *DecisionOrdering* by showing the relevant configuration options at any given time while disabling others such that there is no inconsistency or conflict while reducing the manual configuration efforts and increasing the possibility of *ConfigurationDecision* inference. These tools use different approaches such as multi-objective search algorithms and constraint solvers to provide support for *DecisionOrdering*. Usually, configuration tools (e.g., Pure::Variants [53], SPLOT [55], FMP [48]) supporting feature-oriented notations allow users to configure a product in any arbitrary order. Yue et al. [64] proposed a multi-objective search-based *DecisionOrdering* approach to support CPS PLE, which is implemented in Zen-Configurator.

RevertingDecision: From Table B-13, we can notice that all the configuration tools except Covamof [54], Zen-Configurator [57], and C2O Configurator [59] support *RevertingDecision*.

DOPLER [50, 73] and SPLOT [55] support *RevertingDecision* by maintaining the history of *ConfigurationDecision*s made and allowing them to revert step by step using undo or directly go to the un-configured stage using the reset option. Kumbang Configurator [47, 71] and Quaestio [49] allow reverting the last *ConfigurationDecision* made by the user and corresponding inferred *ConfigurationDecision*s. Generally, configuration tools (e.g., Pure::Variants [53], FMP [48], FeatureID [58]) supporting feature-oriented notations allow users to select/unselect any feature at the given time, however, they do not revert the automatically made *ConfigurationDecision*s, unless there is an inconsistency.

WellFormednessChecking: All the variability modeling/configuration tools support *WellFormednessChecking* to ensure the well-formedness of variability models, however, not all the tools support *WellFormednessChecking* for product models (*ResolutionModel*s). As shown in Table B-13, Kumbang Configurator [47, 71], FMP [35, 48], and Quaestio [49] provide support for *WellFormednessChecking* for product models (*ResolutionMode*ls). These tools ensure the well-formedness against the *WellFormednessConstraints* using constraint solvers and evaluators (e.g., OCL solver EsOCL and OCL evaluator Dresden can be used for OCL constraints). In [35], Czarnecki et al. proposed an approach for checking the well-formedness of feature model templates against OCL constraints. Similarly, Heidenreich [43] also checks the well-formedness of different *PLEModel*s constructed using feature models against the defined *WellFormednessConstraints*.

ConformanceChecking: As shown in Table B-13, all the configuration tools provide support for *ConformanceChecking* to ensure that *ConfigurableParameter*s are configured correctly according to *ConformanceConstraint*s, such that a valid product can be derived. Usually, these tools check the conformance on the fly (i.e., after every *ConfigurationDecision* is made). In case a non-conformity is detected, some tools do not let the user go to the next *ConfigurationDecision* (e.g., Quaestio [49]) whereas others allow users to make *ConfigurationDecision*s while showing the non-conformity (e.g., SPLOT [55]). In [9, 39], Lu et al. proposed a model-based approach for incremental *ConformanceChecking*, which is implemented in Zen-Configurator. Similarly, Maszo et al. [66] and Hwan et al. [68] also proposed two approaches for *ConformanceChecking*, which are implemented in DOPLER and FMP respectively.

ConsistencyChecking: Table B-13 indicates that all the configuration tools provide support for *ConsistencyChecking* with different scopes based on *ConsistencyConstraint*s. All the configuration tools check consistency within and across models (i.e., *IntraModel* and *InterModel*), however, some of the configuration tools (e.g., DOPLER [40, 50, 66, 69, 73, 74]) also check the consistency within or across the views. Usually, these tools provide support for *ConsistencyChecking* on the fly and give feedback to users about detected inconsistencies. Furthermore, FMP [48] ensures consistency across the feature model and existing product configurations given the feature model is evolved [68]. FMP [48] also checks if at least one valid product can be derived from the feature model [75]. Alférez et al. [62] proposed an approach to check the consistency between feature models and use case scenarios. In the context of the DOPLER tool, Vierhauser et al. [40] proposed an approach for incremental *ConsistencyChecking* within variability models and across variability models and code. Similarly, Maszo et al. [66] proposed a constraint programming based approach to check the consistency of DOPLER variability models.

ResolvingViolation: All the configuration tools provide support for *ResolvingViolation* to some extent for resolving the inconsistencies. Some of these tools (e.g., Covamof [54, 70], Kumbang

Configurator [47], Quaestio [49], Pure::Variants [53]) report inconsistencies to the users and get their feedback to resolve the inconsistencies manually whereas others (e.g., SPLOT [55], Zen-Configurator [10, 57], FMP [48, 75], DOPLER [50, 63, 74]) attempt to resolve them automatically if possible, otherwise report them to users. To resolve inconsistencies automatically, these tools use either solvers (e.g., SPLOT [55] uses SAT Solver) or Multi-objective search algorithms (e.g., Zen-Configurator [10, 57]). Some of the configuration tools (e.g., FMP [48, 75], DOPLER [50]) provide an option of auto-complete, which resolves the inconsistencies automatically. Furthermore, DOPLER [50] automatically resolves the inconsistencies due to the evolution of variability models [63].

CollaborativeConfiguration: From Table B-13, we can see that three configuration tools Pure::Variants [53], DOPLER [50, 65], and Gears Tool [60] support *CollaborativeConfiguration*. Pure::Variants [53] allows multiple users to configure a product simultaneously. DOPLER [50, 65] and Gears Tool [60] support role-based *CollaborativeConfiguration*, which allows users with different roles (e.g., business *Stakeholder*s, technical *Stakeholder*s) to configure the products. Moreover, Marcílio et al. [44, 45, 61] also proposed an approach to support *CollaborativeConfiguration* for the product lines modeled using feature model.

ImpactAnalysis: Table B-13 shows that five configuration tools DOPLER [50, 67, 73], SPLOT [55], FMP [48, 75], Quaestio [49], and C2O Configurator [59] provide support for *ImpactAnalysis* in different capacities. DOPLER [50] analyzes the impact of each *ConfigurationDecision* made on existing *ConfigurationDecision*s as well as on business values (e.g., in terms of cost) [73]. Moreover, DOPLER also analyzes the impact of changes in the variability model (e.g., adding or replacing a feature) on existing products [67]. SPLOT [55] analyzes the impact of user made *ConfigurationDecision*s, in terms of *ConfigurationDecision*s inferred, the number of checks performed by SAT solver to check the consistency, and time used by the solver. FMP [48] analyzes the impact of a *ConfigurationDecision* on total possible valid configurations left [75]. Quaestio [49] supports *ImpactAnalysis* by showing the impact level (i.e., showing the impact on variability model) for each *ConfigurationDecision* in Fact-Inspector Window. Similarly, C2O Configurator [59] measures the impact of a *ConfigurationDecision* on existing *ConfigurationDecision*s.

ConflictDetection: As shown in Table B-13, none of the configuration tools supports *ConflictDetection* except Quaestio [49]. Quaestio [49] provides support for *ConflictDetection* to discover conflicting constraints. Usually, the configuration tools detect the conflict among the *ConfigurationDecision*s based on the defined constraints, they do not find the conflicting constraints.

ConstraintSelection: None of the reviewed configuration tools except DOPLER [50] and Zen-Configurator [57] provides support of *ConstraintSelection* (Table B-13). DOPLER [50] has *ConstraintSelection* functionality that is used to select a subset of constraints constraining the relevant *VariationPoint*s at a given time, which are further used in *ConsistencyChecking* [40]. This helps to improve the performance of the tool in terms of memory consumption as well as computation cost. Similarly, Zen-Configurator [57] uses *ConstraintSelection* for different functionalities (e.g., *ConformanceChecking*, Non-Conformity resolving).

ConfigurationOptimization: None of the reviewed configuration tools except DOPLER [50] provides support for *ConfigurationOptimization* (Table B-13). DOPLER [50] support *ConfigurationOptimization* by providing calculated business value in terms of optimization measures (e.g., cost, return on investment) and then let users select the appropriate configurations [51].

Similarly, Zhou et al. [52] proposed an approach that uses multi-objective search algorithms to get optimized product configurations in terms of cost.

RedundancyDetection: Table B-13 shows that none of the reviewed configuration tools provides support for *RedundancyDetection* to check the redundancy in configuration files, however, FeatureID [58] checks the redundant constraints.

IncompletenessDetection: As shown in Table B-13, none of the reviewed configuration tools except SPLOT [55] and Kumbang Configurator [47] provides support for *IncompletenessDetection*. SPLOT [55] shows the percentage of configured *ConfigurableParameter*s at a given time whereas Kumbang Configurator [47, 71] indicates if the configuration is complete or incomplete as a Boolean option.

In summary, all the functionalities except *RedundancyDetection* are supported by one or more configuration tools (i.e., *FC*=92%). This indicates that the identified functionalities based on our experience of working with CPS PLs, are quite consistent with what has been reported in the literature and what has been implemented in the tools. However, we noticed that none of the existing tools provide support for all the functionalities. This is because the tools were proposed with specific objectives of the authors. Some functionalities (e.g., *ConformanceChecking*, *ConsistencyChecking*, and *DecisionInference*) are widely considered important, and therefore they have been mostly implemented. However, the least reported functionalities also play an important role in the configuration process of CPS PLE. For example, *ConflictDetection*, *RedundancyDetection*, and *IncompletenessDetection* are important to ensure the correctness of product configuration. Similarly, *ConfigurationOptimization* is important because it helps to achieve business goals (e.g., lower cost, environment friendly products).

Table B-14. Existing configuration tools and their support for multi-stage and multi-step configuration process

Configuration Tool	Support for Multi-Stage and Multi-Step Configuration Process
Pure::Variants [53]	Allows multiple users to configure simultaneously but with explicitly defined stages/steps
DOPLER [50]	Multiple stages based on various roles (business, technical)
Covamof [54]	No
SPLOT [55]	No
Kumbang Configurator [47]	No
FMP [48]	No
Quaestio [49]	No
Zen-Configurator [57]	No
FeatureID [58]	No
C2O Configurator [59]	No
Gears Tool [60]	Multiple stages based on various roles (business, technical)

Furthermore, we have also assessed the existing configuration tools in terms of their support for the multi-stage and multi-step configuration process in Table B-14. As shown in Table B-14, several existing tools provide some support for configuring the products using multiple stages. For example, DOPLER [50, 65] and Gears Tool [60] support role-based configuration where different stakeholders (e.g., business *Stakeholder*s, technical *Stakeholder*s) perform configuration in various stages and steps. Similarly, Pure::Variants [53] allows multiple users to configure a product at the same time, however, it does not explicitly divide the *ConfigurationDecision*s into multiple stages and steps.

6.3 Discussion

In this section, we provide a discussion based on the results presented in Section 6.2. As we discussed earlier in Section 6.2 that the framework contains all the VP types (Section 4.2) required to capture the variabilities of CPS PLs and *View* types (Section 4.1) to manage them efficiently. It also covers the *Constraint* types (Section 4.3), which are essential for enabling different types of automation of configuration for CPS PLE. Furthermore, it specifies different functionalities of a *ConfigurationSolution* to support multi-stage and multi-step automated configuration of CPSs. The framework is comprehensive in the sense that it provides support for 1) domain engineering of CPS PLs to capture abstractions through well-defined VP and *Constraint* types and manage the captured abstractions using various *View* types, and 2) application engineering of CPS PLs by supporting different types of automation for *ProductConfiguration* using a multi-stage and multi-step configuration process. We proposed a generic conceptual framework independent of any modeling methodology or notation (e.g., feature model) and we do not propose any concrete solution (i.e., modeling methodology and *ConfigurationSolution*). However, the framework has several benefits, for example, the framework clarifies the problem of supporting multi-stage and multi-step automated configuration of CPSs. It also serves as a guide to researchers and practitioners for 1) evaluating an existing PLE solution (i.e., modeling methodology and *ConfigurationSolution*) specific to CPSs, 2) devising a new PLE solution for CPSs, and 3) devising a new PLE solution for the new domain other than CPS.

Figure B-12 presents a step-by-step procedure showing how the framework can be used to evaluate an existing PLE solution or propose a new one. For evaluating an existing PLE solution, the first two steps are to evaluate the capabilities of modeling methodology to capture the variabilities of CPSs and constraints based on VP types (Section 4.2) and *Constraint* types (Section 4.3) in the framework respectively. The third step is to evaluate the modeling methodology based on the *View* types (Section 4.1) required to manage the captured variabilities and constraints. The fourth step is to evaluate the *ConfigurationSolution* based on all the functionalities (Section 5.1) and multi-stage and multi-step *ConfigurationProcess* (Section 3.3). For the first three steps, we simply need to check if *VariationPoint* types (Section 4.2), *Constraint* types (Section 4.3), *View* types (Section 4.1) are supported by the modeling methodology being evaluated. For step 4, we need to check if the *ConfigurationSolution* supports all automated functionalities presented in Section 5.1 in addition to allowing product configuration in a multi-stage and multi-step manner.

To devise a new PLE solution for CPS, the first two steps are related to developing a modeling methodology. The first step is to develop (or select/update an existing) a *ModelingLanguage*, which allows capturing the variabilities of CPSs and managing them into different views based on VP types and *View* types specified in the framework. The second step is to develop or select an existing *ConstraintSpecificationLanguage* and update it to provide support for capturing constraints based on the *Constraint* types specified in the framework. The final step is to develop a *ConfigurationSolution* based on the devised modeling methodology, which supports all the functionalities provided in Section 5.1 as well as multi-stage and multi-step *ConfigurationProcess*.

To develop a new PLE solution for a new domain other than CPSs, the framework needs to be updated by updating VP types, *Constraint* types, *View* types, functionalities, and the *ConfigurationProcess* according to the requirements of the new domain. Once the framework is updated, the rest three steps are the same as for developing a modeling methodology and a *ConfigurationSolution* for CPSs.

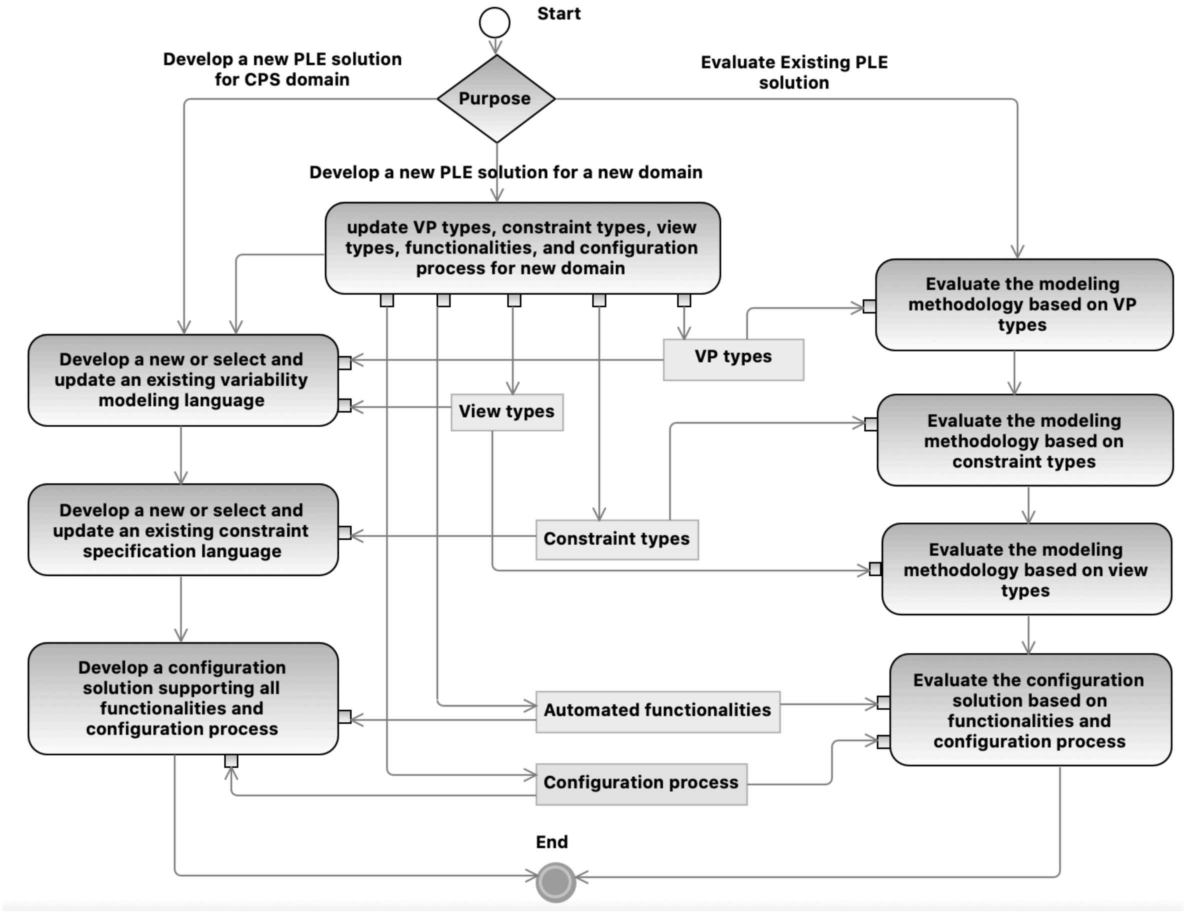

Start
Develop a new PLE solution for CPS domain
Purpose
Evaluate Existing PLE solution
Develop a new PLE solution for a new domain
update VP types, constraint types, view types, functionalities, and configuration process for new domain
Evaluate the modeling methodology based on VP types
VP types
Develop a new or select and update an existing variability modeling language
View types
Evaluate the modeling methodology based on constraint types
Develop a new or select and update an existing constraint specification language
Constraint types
Evaluate the modeling methodology based on view types
Develop a configuration solution supporting all functionalities and configuration process
Automated functionalities
Evaluate the configuration solution based on functionalities and configuration process
Configuration process
End

Figure B-12. Using the framework as a guide to evaluate or propose a PLE solution

6.4 Threats to Validity

Generalization of the results can be questioned with regards to the selection of case studies and configuration tools for the evaluation. To address this, we selected three large-scale real-world case studies as representatives of CPS PLs and evaluated the framework in terms of providing support for domain engineering of CPS PLs (RQ1-RQ3). Similarly, we selected 11 well-known existing configuration tools and existing literature on the automation of configuration to evaluate the framework in terms of providing support for application engineering of CPS PLs (RQ4). The evaluation was performed by reading the literature instead of using all the tools. Thus, it is possible that a certain feature is available in the tool but not reported in the literature. Despite a thorough evaluation, the completeness of the framework cannot be fully ensured as there might be some new requirements (e.g., new VP types or *Constraint* types) in the future. We can only assess the completeness of the framework by evaluating it based on the knowledge collected from existing literature (tools and techniques), real-world case studies, and our experience of conducting industry-oriented research in the field of CPS PLE [13], as we did.

The framework does not provide a concrete modeling methodology or *ConfigurationSolution*; however, it does clarify the problem and lists the requirements for a CPS PLE methodology and *ConfigurationSolution*. Furthermore, several decisions regarding the implementation of

ConfigurationProcess (e.g., dividing the *ConfigurableParameter*s into stages and steps, stages can be defined based on views or based on the phases of development lifecycle, configuring different stages in parallel or a sequence, multiple stakeholders configuring a stage at the same time) are left to the researcher/practitioner designing the configuration solution. This will give the flexibility to the practitioners to implement a *ConfigurationSolution* as per the needs of a given context.

7 Related Work

In this section, we present existing studies on domain and application engineering of product lines, formalizing different aspects of PLE such as configuration process, variability modeling technique, functionalities of a configuration solution.

7.1 Domain Engineering of Product Lines

We discuss existing studies on the domain engineering of PLs in Section 7.1.1 and involved challenges in Section 7.1.2.

7.1.1 Modeling Approaches

To model the commonalities and variabilities of PLs, a large number of variability modeling techniques (VMTs) are available in the literature [76-80]. These VMTs can be categorized into four categories: 1) feature-based VMTs (e.g., [36, 37, 81]), 2) UML based VMTs (e.g., [32, 82, 83]), 3) textual VMTs (e.g., [84]), and 4) other notation (other than UML and feature modeling notation) based VMTs (e.g., [38, 85, 86]). Corresponding to each category, we have discussed some of the VMTs.

Feature-based VMTs are most widely used in industry [87]. A number of tools (e.g., Pure::Variants [53], SPLOT [55], FMP [48], FeatureID [58], Gears Tool [60]) are available to support feature-based VMTs and product configuration. These tools use either basic feature model (FM) [36] or a variation of feature model (e.g., cardinality based feature model (CBFM) [37], Multi-Product Line Feature model [81]). In FM, commonalities are captured as *Mandatory* features and variabilities as *Optional* features and *Alternative* features. Dependencies among features are captured as *Requires* and *Excludes* relationship restrictions. CBFM is the most popular extension of FM, which introduces new concepts such as *Feature Cardinalities*, *Groups* and *Groups Cardinalities*, *Attributes*, and *References*. Multi-Product Line Feature model [81] is an approach proposed to model variability for multiple PLs and their context (e.g., external systems and cloud services) using CBFM notation. Other variations of FM or CBFM (e.g., supported by Pure::Variants [53]) have minor differences (e.g., the number of attributes supported, default values for the attributes).

UML based VMTs use a subset of UML and UML profiles to capture commonalities and variabilities. For example, SimPL is a UML based VMT, which provides notations and guidelines for modeling variabilities and commonalities of integrated control systems (ICS) at the architecture and design level. Several UML modeling tools [88] (e.g., RSA, MagicDraw, and Papyrus) are available for variability modeling with SimPL (or other UML based VMTs) whereas products can be configured using Zen-Configurator [57]. SimPL captures four types of variation points: Attribute-VP, Type-VP, Topology-VP, and Cardinality-VP. Different types of constraints are specified using OCL. Similarly, Clauß [82, 89] proposed a UML profile based on UML class diagram to support variability modeling of PLs and their context (i.e., external agents, systems, and services) at the feature level. The proposed profile supports all the constructs (e.g., *Mandatory*, *Optional*, and *Alternative*, *Requires* and *Excludes* dependencies, *Cardinality*) provided by feature-based VMTs with the help of stereotypes defined in the profile. Additionally, the proposed profile has a stereotype "external" to model external features (i.e., that is not part of the system). It also allows specifying binding time for the variation points. Ziadi et al. [83] proposed another UML profile containing three stereotypes for UML class diagram and five stereotypes for UML sequence diagram. It allows capturing variabilities corresponding to components and their interactions.

Textual VMTs capture the variabilities in the form of text. Dhungana et al. [84] proposed a VMT DOPLER that is independent of any particular domain. It is based on a general variability meta-model that consists of assets and decisions about selecting assets. To apply in a particular domain, the meta-model requires to be extended according to the needs. The meta-model is supported by a meta-tool DecisionKing, which is a part of the DOPLER toolkit. The tool provides support for customization; its implementation can be replaced with domain-specific implementation using plugins. It also provides a model API that can be used in any general tool to create and manipulate models. Similarly, La Rosa et al. [49] also proposed a question-based VMT, which is supported by the Quaestio tool. Several textual notations for feature model (e.g., Feature Description Language [90], Batory [91], Variability Specification Language [92], FAMILIAR [93], Text-based Variability Language [94]) exists in the literature to capture the variabilities in the form of features and groups but using textual notations instead of graphical notations. Some of these textual VMTs (e.g., Batory [91]) support the constructs offered by CBFM whereas others support FM's constructs.

Other notation based VMTs use a completely new notation (e.g., [38, 85, 86]) for capturing variabilities. Common Variability Language (CVL) [38] is a generic VMT that is independent of any domain. CVL defines the orthogonal variability model, i.e., a separate variability model from the base model. The base model can be defined in any Meta-Object Facility (MOF) based language such as MOF based Domain-Specific Language (DSL) or UML corresponding to which variability is defined. An Eclipse-based CVL tool is available that supports CVL partially. Haugen and Øgård [86] have proposed another variability modeling technique for the safety domain, called the Base Variability Model (BVR). BVR is built on the CVL. In BVR some new constructs are added such as *Note*, *Reference*, *Comment*, and *ChoiceOccurrence*, for better expressiveness. An Eclipsed-based BVR tool [95] is available to support BVR VMT. Sinnema et al. [85] proposed COVAMOF to capture variabilities at three levels of abstraction: feature, architecture, and code. COVAMOF offers two views, i.e., the variation point view to give an overview of variabilities in all the abstraction levels of a PL and the dependency view to show the dependencies among variabilities. It captures five types of variation points: *Optional* variation points (selecting zero or one variant), *Alternative* variation points (selecting only one variant), *Optional Variant* variation points (selecting zero or more variant), *Variant* variation points (selecting one or more variant), and *Value* variation points (selecting a value from pre-defined range). Each variation point has two states open and closed. In the case of the open state new variants can be added in the next development phase but a close variation point does not allow adding new variants. It captures logical, numerical, and nominal dependencies among the variabilities. Bühne et al. [96] presented a VMT for requirement engineering that has a tree-like structure similar to the feature model. It captures three types of features, i.e., *Optional*, *Mandatory*, and *Alternative* as well as two types of dependencies, i.e., *Requires* and *Excludes*. Additionally, a concept of assigned requirement artifact is introduced to enable tracking variability through different artifacts.

To summarize, all the feature-based VMTs (e.g., [36, 37, 81]) capture the variabilities such as *Optional* and *Alternative* features, *Cardinality*, and Attributes. Most of these approaches capture the *VariabilityDependencyConstraint*s only, however, some approaches may also allow specifying *ConformanceConstraint*s by specifying the domain of the attributes. Furthermore, there are several textual VMTs (e.g., [90] [91-94]) to capture the variabilities as text, which support the constructs supported by FM and CBFM whereas others capture the variabilities using questionnaires (e.g., [49, 84]). UML based VMTs (e.g., [32, 82, 83]) can capture structural and behavioral variabilities whereas the constraints are captured using OCL. There are also several VMTs (e.g., [38, 85, 86]), which have their own notations to capture the variabilities and constraints.

7.1.2 Key Challenges in Domain Engineering of CPS PLs

On a high level, there are three key challenges in the domain engineering of CPS PLs: capturing abstractions as commonalities and variabilities, specifying constraints, and managing the captured abstractions and constraints efficiently.

Capturing Abstractions: As discussed earlier (Sections 1 and 2), CPSs are large-scale, highly hierarchical, and hybrid systems with complex interactions among different components [1-5]. Thus, unlike traditional software PLs, CPS PLs require capturing variabilities for 1) various domains (e.g., Software, Mechanics, Electronics, Hydraulics), 2) continuous and discrete properties of CPSs, 3) complex interactions among different components, 4) topologies, and 5) software deployment on hardware. Additionally, CPS PLs also require identifying the binding time (e.g., design time) for captured variabilities. This means modeling CPS PLs requires a sophisticated approach that can capture CPS specific variabilities. As discussed in Section 7.1.1, none of the existing modeling approaches can capture all types of variabilities specified in our framework. As discussed in [16], the SimPL methodology [32] can capture more types of variabilities for CPS PLs than any other modeling approach but not all.

Capturing Constraints: To enable the automation of configuration to support application engineering of CPS PLs, various types of constraints (Section 4.3) need to be captured [9, 97]. As discussed in Section 7.1.1, most of the existing modeling approaches only capture *VariabilityDependencyConstraint*s. Some approaches also allow specifying *ConformanceConstraint*s by specifying the possible values of the attributes. UML based approaches (e.g., [32, 82, 83]) capture constraints using OCL, which can capture complex constraints to support CPS PLE.

Managing Abstractions and Constraints: Since CPS PLs are hybrid systems with multiple stakeholders, which often contain a large number of variabilities and constraints (Table B-8 and Table B-10). Thus, they need to be managed efficiently in multiple views to deal with the inherent complexity of the CPS domain and cater multiple stakeholders. None of the existing modeling approaches explicitly support all the views specified in our framework. There are a few modeling approaches that support more than one view. For example, SimPL [32] provides separate views for software, hardware, and the allocation of software to hardware.

The proposed framework addresses the challenges mentioned above by providing various VP types to capture variabilities (Section 4.2), constraint types to capture constraints (Section 4.3), and view types to manage captured variabilities and constraints (Section 4.1).

7.2 Application Engineering of Product Lines

As in RQ4 (Section 6.2.4), we already covered the literature (both tools and techniques) on automation of configuration to support application engineering in general, therefore, we will not discuss it again to avoid repetition. In Section 7.2.1, we present the literature discussing multi-stage and multi-step configuration process and in Section 7.2.2, we discuss the key challenges in the application engineering of CPS PLs.

7.2.1 Configuration Process

Czarnecki et al. [37] introduced the concept of staged configuration for feature models, which can be achieved by stepwise specialization of feature models. This allows various stakeholders to collaborate and derive a product in multiple stages. Furthermore, in [14], Czarnecki et al. proposed to create separate feature models to capture variabilities in each stage to cater different stakeholders. They also discuss the need for decomposing and merging feature models for various stages of the configuration process. Classen et al. [98] formalized the multi-level staged configuration process for feature model notation.

In [99, 100], Chavarriaga et al. proposed to capture variabilities using multiple features models and resolve variabilities using a multi-stage configuration process. Their proposed approach makes use of feature solution graphs (FSGs) to detect and report conflicting configuration decisions in a multi-stage configuration process. In FSGs, feature models for various stages are arranged into pairs. For each pair, configuration decisions made in one stage are propagated into other to evaluate the possible configuration decisions. If certain configuration decisions in one stage do not allow making any configuration decision in the latter stage, configuration decisions causing the problem are marked as conflicting decisions. The article also provides formal semantics for feature models and FSGs. Urli et al. [101] proposed SpineFM for capturing variabilities in the form of multiple feature models and configuration is performed in multiple stages. The tool of SpineFM ensures configuration consistency at each step and propagates configuration decisions to infer other configuration decisions when possible. SpineFM was evaluated with an industrial case study.

Abbasi et al. [102] proposed to divide features of a feature model into multiple subsets to cater multiple stakeholders. Each subset is presented in a view for specific stakeholders to facilitate configurations decisions. To do so, the SPLOT tool [55] was extended to incorporate multiple views. The approach was evaluated with a case study from the aerospace industry.

Schroeter et al. [103] proposed a conceptual configuration management solution to support dynamic multi-stage configuration of cloud-based multi-tenant aware applications. The proposed solution enforces an adaptive staged configuration process that can add and remove stakeholders dynamically and allow reconfiguration of variants when stakeholders' objectives change. Variabilities of functionality and service qualities (i.e., availability, performance, security) of cloud applications are captured with extended feature models. Different aspects of the proposed solution were exemplified using a running example.

Hubaux et al. [104] proposed a textual approach to specify different concerns for various stakeholders corresponding to feature model diagrams. Based on the defined concerns, different views can be generated to present concern-specific configuration options to the stakeholders. The goal is to define criteria to split feature models into different views for various stakeholders to perform configuration in multiple stages. The article proposed three visualizations and illustrated them using an open-source web-based meeting management system. A tool was also developed on top of SPLOT [55].

In [105, 106], White et al. formalized multi-step configuration problems and proposed MUSCLE, which transforms multi-step feature configuration problems into constraint satisfaction problems (CSPs) and generates configurations that meet multi-step constraints with a constraint solver. Furthermore, the authors also discussed mechanisms for optimally deriving configurations that minimize or maximize a property of the configuration process (e.g., cost).

To summarize, several studies exist in the literature that advocate using the multi-stage configuration process where a stage is defined for a particular stakeholder. Furthermore, all these studies discussing the multi-stage configuration process rely on feature model-based notations (e.g., CBFM). Our framework, however, covers concepts of the multi-stage and multi-step configuration process where configuration stages are defined to cater various domain experts (e.g., software engineers) and steps for different phases of a CPS development lifecycle.

7.2.2 Key Challenges in Application Engineering of CPS PLs

CPS PLs involve various domains and stakeholders that require the multi-stage and multi-step configuration process. This makes application engineering of CPS PLs more complex due to, 1) it requires splitting and merging variabilities into various stages and steps to allow collaborative configuration performed by various stakeholders; 2) ensuring a valid product is derived becomes more difficult as it requires detecting different types of inconsistencies emerged due to configuration decisions made in different configuration stages and steps; 3) Inferring configuration decision becomes difficult, as it requires propagating the configuration decisions in multiple configuration stages for every single configuration decision in a specific configuration stage. 4) reverting decisions is more complex, as undoing one configuration decision may require rolling back all inferred decisions in multiple configuration stages; and 5) resolving inconsistencies is more difficult as it requires defining priorities for different domains (or stakeholders) such that stakeholders can change the configuration decisions for low priority domains (e.g., usually software is configured according to hardware configurations) to fix inconsistencies. Overall implementing various kinds of automated functionalities in a configuration solution that supports the application engineering of CPS PLs becomes more challenging.

7.3 Formalization in the Context of PLE

This section presents the existing studies focusing on the formalization of different aspects of PLE such as variability modeling, configuration process, constraints in PLE, and functionalities of a configuration solution, which are discussed as follow:

Several studies exist in the literature that focus on the formalization of, e.g., different functionalities of a configuration solution, semantics of variability modeling language, different types of variabilities, and configuration process. However, all these formalisms focus on specific modeling methodologies (e.g., feature model, SimPL methodology), unlike our work in which we formalize independent of any modeling methodology or notation. Moreover, there does not exist a single study, which covers all aspects, e.g., the configuration process, variation point types, constraint types, models, views, and most importantly the functionalities of a configuration solution.

Several studies formalized feature models and their extensions (e.g., cardinality-based feature model). Czarnecki et al. [107] formalized Cardinality-based feature models by translating feature models into context-free grammar. Moreover, they also formalized the multi-stage configuration process using mathematical notations such as set theory. Similarly, Deursen and Klint [90] formalized the textual notation based feature model using algebraic specifications. In [108], Janota and Kiniry formalized the feature modeling using higher-order logic. In [109], another formalism is presented, where thorough formal semantics for extended feature models and the notion of consistency are provided. The formalism focuses on multi-view and multi-staged feature-based configuration.

Behjati et al. [19] formalized different concepts of SimPL modeling methodology such as reference architecture for a PL, member product, components types, four types of variabilities (i.e., Attribute-VP, Cardinality-VP, Type-VP, and Topology-VP), and configuration process specific to SimPL. The focus of the work [19] is interactive and iterative architecture-level configuration where the authors describe how the configuration state changes as a result of resolving the above-mentioned four types of variabilities. They also proposed a configuration algorithm and implemented it using constraint satisfaction techniques for supporting a semi-automated configuration of CPSs.

Various studies formalized different functionalities of a configuration solution. For example, Lu et al. [9] formalized the conformance checking, decision inference, and decision ordering using OCL constraints for the variability models developed using SimPL modeling notation. Behjati et al. [19] formalized the consistency checking functionality for SimPL modeling methodology based PL and product models. In [35], Czarnecki et al. formalized the well-formedness checking feature models against well-formedness constraints specified using OCL constraints. Similarly, Marcílio et al. [44] formalized the collaborative configuration functionality for feature models.

8 Conclusion

Enabling Product Line Engineering (PLE) for Cyber-Physical Systems (CPSs) is very challenging due to large-scale, intrinsic complexity, and the existence of numerous variabilities and constraints, which requires well-defined modeling methodologies to capture the variabilities and constraints as well as automation of configuration process to derive valid CPS products. In this paper, we present a conceptual framework for supporting the multi-stage and multi-step automated configuration of CPSs. More specifically, we present a classification of constraints and variation points using a UML and OCL based conceptual models. We also presented 14 possible functionalities of an automated configuration solution and provided their formal definitions using mathematical notations and a UML and OCL based conceptual model, independent of any modeling methodology or notation. Furthermore, we also formalized the context of the study (i.e., CPS, general PLE concepts, multi-stage and multi-step configuration process) and different concepts related to modeling of CPS product lines such as models in PLE, model elements, and views. For validation of the framework, we present the results from three real-world case studies and an extensive literature review showing the coverage of variation point types, constraints types, views, and functionalities to facilitate CPS PLE. Such a conceptual framework aims to provide insights to researchers and practitioners from our experience that can help them to systematically devise new modeling methodologies and automated configuration solution for CPS PLE.

Acknowledgement

Safdar Aqeel Safdar and Hong Lu are supported by the Zen-Configurator project funded by the Research Council of Norway (grant no. 240024/F20) under the category of Young Research Talents of the FRIPO funding scheme. Tao Yue and Shaukat Ali are supported by the Co-evolver project (No. 286898/F20) funded by the Research Council of Norway under the FRIPRO program.

References

1. Derler, P., E.A. Lee, and A.S. Vincentelli, Modeling Cyber–Physical Systems. Proceedings of the IEEE Special issue on CPS, 2012. 100(1): p. 13-28.
2. Cyber-Physical Systems (CPSs). Available from: http://cyberphysicalsystems.org/.
3. Rawat, D.B., J.J. Rodrigues, and I. Stojmenovic, Cyber-Physical Systems: From Theory to Practice. 2015: CRC Press.
4. Ma, T., S. Ali, and T. Yue, Modeling foundations for executable model-based testing of self-healing cyber-physical systems. Software & Systems Modeling, 2019. 18(5): p. 2843-2873.
5. Zhang, M., et al., Uncertainty-Wise Cyber-Physical System test modeling. Software & Systems Modeling, 2017: p. 1-40.
6. Nie, K., et al. Constraints: the core of supporting automated product configuration of cyber-physical systems. in Proceeding of International Conference on Model-Driven Engineering Languages and Systems (MODELS). 2013. Springer.
7. Iglesias, A., et al. Product line engineering of monitoring functionality in industrial cyber-physical systems: A domain analysis. in Proceedings of the 21st International Systems and Software Product Line Conference-Volume A. 2017. ACM.
8. Arrieta, A., et al., Search-Based test case prioritization for simulation-Based testing of cyber-Physical system product lines. Journal of Systems and Software, 2019. 149: p. 1-34.
9. Lu, H., et al., Model-based Incremental Conformance Checking to Enable Interactive Product Configuration. Information and Software Technology (IST), 2015. 72: p. 68-89.
10. Lu, H., et al., Nonconformity Resolving Recommendations for Product Line Configuration, in International Conference on Software Testing. 2016, IEEE. p. 57-68.
11. ISO, Software and systems engineering -- Reference model for product line engineering and management. 2013, ISO.
12. Yue, T., S. Ali, and B. Selic. Cyber-Physical System Product Line Engineering: Comprehensive Domain Analysis and Experience Report. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2015. ACM.
13. Yue, T., S. Ali, and B. Selic. Cyber-physical system product line engineering: comprehensive domain analysis and experience report. in Proceedings of the 19th International Conference on Software Product Line. 2015. ACM.
14. Czarnecki, K., S. Helsen, and U. Eisenecker, Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice, 2005. 10(2): p. 143-169.
15. OCL, Object Constraint Language Specification, Version 2.2. 2011, Object Management Group (OMG).
16. Safdar, S.A., et al. Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering. in Proceeding of International Conference on System Analysis and Modeling (SAM). 2016. Springer.
17. ULMA Handling Systems. 2002; Available from: http://www.ulmahandling.com.
18. Ali, S., et al. Empowering Testing Activities with Modeling-Achievements and Insights from Nine Years of Collaboration with Cisco. in Proceeding of International Conference on Model-Driven Engineering and Software Development (MODELSWARD). 2017. Springer.
19. Behjati, R., S. Nejati, and L.C. Briand, Architecture-level configuration of large-scale embedded software systems. ACM Transactions on Software Engineering and Methodology (TOSEM), 2014. 23(3): p. 25.

20. Safdar, S.A., et al. Mining Cross Product Line Rules with Multi-Objective Search and Machine Learning in Proceeding of The Genetic and Evolutionary Computation Conference (GECCO). 2017. Berlin, Germany: ACM.
21. Zhang, M., et al. Understanding uncertainty in cyber-physical systems: a conceptual model. in European Conference on Modelling Foundations and Applications. 2016. Springer.
22. Chakravarthy, V., J. Regehr, and E. Eide. Edicts: implementing features with flexible binding times. in Proceedings of the 7th international conference on Aspect-oriented software development. 2008. ACM.
23. Beuche, D. and J. Weiland. Managing flexibility: Modeling binding-times in simulink. in European Conference on Model Driven Architecture-Foundations and Applications. 2009. Springer.
24. Hotz, L., et al. Evaluation across multiple views for variable automation systems. in Proceedings of the 19th International Conference on Software Product Line. 2015. ACM.
25. OMG, Systems Modeling Language (SysML) v1.4, http://sysml.org/. 2015.
26. The UML MARTE profile, http://www.omgmarte.org/.
27. Selic, B. and S. Gérard, Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. 2013: Elsevier.
28. Bagheri, E., et al. Configuring software product line feature models based on stakeholders' soft and hard requirements. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2010. Springer.
29. Nadi, S., et al., Where do configuration constraints stem from? an extraction approach and an empirical study. IEEE Transactions on Software Engineering (TSE), 2015. 41(8): p. 820-841.
30. Czarnecki, K., S. She, and A. Wasowski. Sample spaces and feature models: There and back again. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2008. IEEE.
31. Bécan, G., et al. Synthesis of attributed feature models from product descriptions. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2015. ACM.
32. Behjati, R., et al., SimPL: a product-line modeling methodology for families of integrated control systems. Information and Software Technology, 2013.
33. Jaring, M. and J. Bosch. A taxonomy and hierarchy of variability dependencies in software product family engineering. in Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International. 2004. IEEE.
34. Safdar, S.A., et al., Using multi-objective search and machine learning to infer rules constraining product configurations. Automated Software Engineering, 2019. 27(1): p. 1-62.
35. Czarnecki, K. and K. Pietroszek, Verifying feature-based model templates against well-formedness OCL constraints, in Proceedings of the 5th international conference on Generative programming and component engineering. 2006, ACM: Portland, Oregon, USA. p. 211-220.
36. Kang, K., Cohen, Sholom., Hess, James., Novak, William., & Peterson, A., Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021), in Secondary Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021), Secondary Kang, K., Cohen, Sholom., Hess, James., Novak, William., & Peterson, A., Editor. 1990. RN. Available From:
37. Czarnecki, K., S. Helsen, and U. Eisenecker, Staged configuration using feature models, in Software Product Lines. 2004, Springer. p. 266-283.
38. Haugen, O., Common Variability Language (CVL). OMG Revised Submission, 2012.
39. Lu, H., et al., Zen-CC: An Automated and Incremental Conformance Checking Solution to Support Interactive Product Configuration, in 25th International Symposium on Software Reliability Engineering. 2014, IEEE. p. 13-22.
40. Vierhauser, M., et al., Flexible and scalable consistency checking on product line variability models, in Proceedings of the IEEE/ACM international conference on Automated software engineering. 2010, ACM: Antwerp, Belgium. p. 63-72.
41. Gomaa, H. and M.E. Shin. A multiple-view meta-modeling approach for variability management in software product lines. in International Conference on Software Reuse. 2004. Springer.
42. El-Sharkawy, S. and K. Schmid. Supporting the effective configuration of software product lines. in Proceedings of the 16th International Software Product Line Conference. 2012. ACM.
43. Heidenreich, F. Towards systematic ensuring well-formedness of software product lines. in Proceedings of the First International Workshop on Feature-Oriented Software Development. 2009. ACM.
44. Mendonça, M., et al., Collaborative Product Configuration. Journal of Software, 2008. 3(2): p. 69.
45. Mendonça, M., T.T. Bartolomei, and D. Cowan. Decision-making coordination in collaborative product configuration. in Proceedings of the 2008 ACM symposium on Applied computing. 2008. ACM.
46. Ali, S., et al. Insights on the use of OCL in diverse industrial applications. in International Conference on System Analysis and Modeling. 2014. Springer.
47. Myllärniemi, V., et al. Kumbang configurator–a configuration tool for software product families. in 19th International Joint Conference on Artificial Intelligence. 2005. Citeseer.

48. Czarnecki, K., et al. fmp and fmp2rsm: eclipse plug-ins for modeling features using model templates. in OOPSLA '05 Companion. 2005. ACM.
49. La Rosa, M., et al., Questionnaire-based variability modeling for system configuration. Software & Systems Modeling, 2009. 8(2): p. 251-274.
50. Rabiser, R., et al. DOPLER, Decision Oriented Product Line Engineering for effective Reuse. Available from: http://ase.jku.at/dopler/.
51. Lettner, D., et al. Supporting end users with business calculations in product configuration. in Proceedings of the 16th International Software Product Line Conference-Volume 1. 2012. ACM.
52. Zhou, C., Z. Lin, and C. Liu, Customer-driven product configuration optimization for assemble-to-order manufacturing enterprises. The International Journal of Advanced Manufacturing Technology, 2008. 38(1): p. 185-194.
53. Pure-Systems. Pure::Variants available at: http://www.pure-systems.com/. 2017].
54. Sinnema, M., et al., Covamof: A framework for modeling variability in software product families, in Software product lines, R.L. Nord, Editor. 2004, Springer Heidelberg. p. 197-213.
55. Mendonca, M., M. Branco, and D. Cowan. SPLOT: software product lines online tools. in Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. 2009. ACM.
56. Antkiewicz, M. and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. in Proceedings of OOPSLA, workshop on eclipse technology eXchange. 2004. ACM.
57. Hong, L., Y. Tao, and A. Shaukat. Zen-Configurator: Interactive and Optimal Configuration of Cyber Physical System Product Lines. [cited 2017; Available from: https://www.simula.no/research/projects/zen-configurator-interactive-and-optimal-configuration-cyber-physical-system.
58. Thüm, T., et al., FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming, 2014. 79: p. 70-85.
59. Nöhrer, A. and A. Egyed, C2O configurator: a tool for guided decision-making. Automated Software Engineering, 2013. 20(2): p. 265-296.
60. Krueger, C.W. Biglever software gears and the 3-tiered spl methodology. in Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. 2007. ACM.
61. De Mendonça, M.F., Collaborative Feature-Based Product Configuration in Software Product Lines. 2007.
62. Alférez, M., et al., Supporting Consistency Checking between Features and Software Product Line Use Scenarios, in Top Productivity through Software Reuse, K. Schmid, Editor. 2011, Springer Berlin Heidelberg. p. 20-35.
63. Heider, W., R. Rabiser, and P. Grünbacher, Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. International Journal on Software Tools for Technology Transfer (STTT), 2012. 14(5): p. 613-630.
64. Yue, T., et al. Search-based decision ordering to facilitate product line engineering of cyber-physical system. in Model-Driven Engineering and Software Development (MODELSWARD), 2016 4th International Conference on. 2016. IEEE.
65. Rabiser, R., P. Grünbacher, and G. Holl, Improving awareness during product derivation in multi-user multi product line environments. 2010.
66. Mazo, R., et al. Using constraint programming to verify DOPLER variability models. in Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. 2011. ACM.
67. Heider, W., et al. Using regression testing to analyze the impact of changes to variability models on products. in Proceedings of the 16th International Software Product Line Conference-Volume 1. 2012. ACM.
68. Hwan, C., P. Kim, and K. Czarnecki. Synchronizing cardinality-based feature models and their specializations. in Model Driven Architecture–Foundations and Applications. 2005. Springer.
69. Dhungana, D., P. Grünbacher, and R. Rabiser, The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering, 2011. 18(1): p. 77-114.
70. Sinnema, M., S. Deelstra, and P. Hoekstra, The COVAMOF derivation process. Reuse of Off-the-Shelf Components, 2006: p. 101-114.
71. Myllärniemi, V., Kumbang Configurator—a tool for configuring software product families. 2005, Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering.
72. Nie, K., T. Yue, and S. Ali. Towards a Search-based Interactive Configuration of Cyber Physical System Product Lines. in Demos/Posters/StudentResearch@ MoDELS. 2013.
73. Rabiser, R., P. Grunbacher, and M. Lehofer. A qualitative study on user guidance capabilities in product configuration tools. in Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on. 2012. IEEE.

74. Dhungana, D., P. Grünbacher, and R. Rabiser, DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling. VaMoS, 2007. 2007: p. 01.
75. Czarnecki, K. and C.H.P. Kim. Cardinality-based feature modeling and constraints: A progress report. in International Workshop on Software Factories. 2005.
76. Chen, L., M. Ali Babar, and N. Ali, Variability management in software product lines: A systematic review, in 13th International Software Product Line Conference. 2009. p. 81-90.
77. Arrieta, A., G. Sagardui, and L. Etxeberria, A comparative on variability modelling and management approach in simulink for embedded systems. V Jornadas de Computación Empotrada, ser. JCE, 2014.
78. Djebbi, O. and C. Salinesi. Criteria for comparing requirements variability modeling notations for product lines. in 4th International Workshop on Comparative Evaluation in Requirements Engineering. 2006. IEEE.
79. Sinnema, M. and S. Deelstra, Classifying variability modeling techniques. Information and Software Technology, 2007. 49(7): p. 717-739.
80. Eichelberger, H. and K. Schmid. A systematic analysis of textual variability modeling languages. in Proceedings of the 17th International Software Product Line Conference. 2013. ACM.
81. Hartmann, H. and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2008. IEEE.
82. Clauß, M. and I. Jena. Modeling variability with UML. in GCSE 2001 Young Researchers Workshop. 2001. Citeseer.
83. Ziadi, T., L. Hélouët, and J.-M. Jézéquel, Towards a UML profile for software product lines, in Software Product-Family Engineering. 2004, Springer. p. 129-139.
84. Dhungana, D., P. Grünbacher, and R. Rabiser, Domain-specific adaptations of product line variability modeling, in Situational Method Engineering: Fundamentals and Experiences. 2007, Springer. p. 238-251.
85. Sinnema, M., et al. Covamof: A framework for modeling variability in software product families. in International Systems and Software Product Line Conference (SPLC). 2004. Springer.
86. Haugen, Ø. and O. Øgård, BVR–Better Variability Results, in System Analysis and Modeling: Models and Reusability. 2014, Springer. p. 1-15.
87. Berger, T., et al. A survey of variability modeling in industrial practice. in Proceedings of 7th International Workshop on Variability Modelling of Software intensive Systems. 2013. ACM.
88. Safdar, S.A., M.Z. Iqbal, and M.U. Khan, Empirical Evaluation of UML Modeling Tools–A Controlled Experiment, in European Conference on Modeling Foundations and Applications. 2015, Springer: Italy. p. 33-44.
89. Clauß, M. Generic modeling using UML extensions for variability. in Workshop on Domain Specific Visual Languages at OOPSLA. 2001.
90. Van Deursen, A. and P. Klint, Domain-specific language design requires feature descriptions. CIT. Journal of computing and information technology, 2002. 10(1): p. 1-17.
91. Batory, D. Feature models, grammars, and propositional formulas. in SPLC. 2005. Springer.
92. Abele, A., et al., The CVM Framework-A Prototype Tool for Compositional Variability Management. VaMoS, 2010. 10: p. 101-105.
93. Acher, M., et al., Familiar: A domain-specific language for large scale management of feature models. Science of Computer Programming, 2013. 78(6): p. 657-681.
94. Boucher, Q., et al. Introducing TVL, a text-based feature modelling language. in Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January. 2010.
95. BVR Eclipse Based Tool. Available from: https://github.com/SINTEF-9012/bvr.
96. Bühne, S., K. Lauenroth, and K. Pohl. Why is it not sufficient to model requirements variability with feature models. in Proceedings of Workshop: Automotive Requirements Engineering (AURE04), Los Alamitos. 2004. Citeseer.
97. Kunming Nie, T.Y., Shaukat Ali., Towards a Search based Interactive Configuration of Cyber Physical System Product Lines, in Demos/Posters/StudentResearch@ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems. 2013. p. 71-75.
98. Classen, A., A. Hubaux, and P. Heymans, A Formal Semantics for Multi-level Staged Configuration. VaMoS, 2009. 9: p. 51-60.
99. Chavarriaga, J., et al., Supporting Multi-level Configuration with Feature-Solution Graphs Formal Semantics and Alloy Implementation, in Secondary Supporting Multi-level Configuration with Feature-Solution Graphs Formal Semantics and Alloy Implementation, Secondary Chavarriaga, J., et al., Editors. 2013. RN. Available From:
100. Chavarriaga, J., et al. Propagating decisions to detect and explain conflicts in a multi-step configuration process. in International Conference on Model Driven Engineering Languages and Systems. 2014. Springer.

101. Urli, S., M. Blay-Fornarino, and P. Collet. Handling complex configurations in software product lines: a tooled approach. in Proceedings of the 18th International Software Product Line Conference-Volume 1. 2014.
102. Abbasi, E.K., A. Hubaux, and P. Heymans. A toolset for feature-based configuration workflows. in 2011 15th International Software Product Line Conference. 2011. IEEE.
103. Schroeter, J., et al. Dynamic configuration management of cloud-based applications. in Proceedings of the 16th International Software Product Line Conference-Volume 2. 2012.
104. Hubaux, A., et al., Supporting multiple perspectives in feature-based configuration. Software & Systems Modeling, 2013. 12(3): p. 641-663.
105. White, J., et al., Evolving feature model configurations in software product lines. Journal of Systems and Software, 2014. 87: p. 119-136.
106. White, J., et al. Automated reasoning for multi-step feature model configuration problems. in Proceedings of the 13th International Software Product Line Conference. 2009. Carnegie Mellon University.
107. Czarnecki, K., S. Helsen, and U. Eisenecker, Formalizing cardinality‐based feature models and their specialization. Software process: Improvement and practice, 2005. 10(1): p. 7-29.
108. Janota, M. and J. Kiniry. Reasoning about feature models in higher-order logic. in Software Product Line Conference, 2007. SPLC 2007. 11th International. 2007. IEEE.
109. Hubaux, A., Feature-based configuration: Collaborative, dependable, and controlled. University of Namur, Belgium, 2012.
110. Cisco C-Series Video Conferencing Systems. 2012; Available from: https://www.cisco.com/c/en/us/products/collaboration-endpoints/telepresence-integrator-c-series/index.html.

9 Appendix A: Concepts Definitions and Examples

Table B-15. Concept definitions of the PLE conceptual model

Concept	Definition and Example
D1: ProductLine	*ProductLine* represents a set of products that share common and variable features while relying on the same domain architecture. For example, C-Series video conferencing systems produced by Cisco Systems [110].
D2: MemberProduct	A particular product belonging to a *ProductLine* is called *MemberProduct*. For example, product C60 from C-Series produced by Cisco Systems [110].
D3: Asset	*Asset* represents an output produced from a domain or an application engineering process. For example, a particular test case.
D4: DomainAsset	*DomainAsset* represents the output of domain engineering processes. For example, all the test cases corresponding to a *ProductLine*.
D5: ApplicationAsset	*ApplicationAsset* represents the output of the application engineering process of a particular *MemberProduct*. For example, a subset of test cases selected for a *MemberProduct*.
D6: AssetBase	*AssetBase* is a repository containing a set of *DomainAsset*s and *ApplicationAsset*s.
D7: DomainArchitecture	*DomainArchitecture* represents the reference architecture for a *ProductLine* that includes architectural structure and common constraints applicable to all the *MemberProduct*s.
D8: ApplicationArchitecture	*ApplicationArchitecture* represents the reference architecture for a *MemberProduct* that includes architectural structure and constraints applicable to a *MemberProduct*.
D9: Commonality	*Commonality* represents a set of features (functional and non-functional) shared by all the *MemberProduct*s within a *ProductLine*. For example, in Figure B-3, *SubseaControlSystem* is shared by all the *MemberProduct*s.
D10: Variability	*Variability* represents a set of features that may vary across the *MemberProduct*s within a *ProductLine*. For example, in Figure B-3, the concrete type of *SubseaControlSystem* is variable across the *MemberProduct*s.
D11: PLEModel	*PLEModel* is a general concept that represents a system at an abstract level, which is composed of different elements. For example, Figure B-3 (a) And Figure B-3 (b) present two *PLEModel*s.
D12: BaseModel	*BaseModel* is constructed to capture the commonalities corresponding to a *ProductLine*. For example, Figure B-3 (A) presents a *BaseModel*.

D13: VariabilityModel	*VariabilityModel* is developed corresponding to a *BaseModel* for capturing the variabilities for the *ProductLine*. For example, Figure B-3 (b) presents a *VariabilityModel* corresponding to the *BaseModel* presented in Figure B-3 (a).
D14: ResolutionModel	*ResolutionModel* is developed corresponding to a *VariabilityModel* by resolving the variabilities for a particular *MemberProduct*. For example, the model produced by resolving the variabilities of the model shown in Figure B-3 (b).
D15: ModelElement	Elements constituting a *PLEModel* representing the structure or behavior of the system are called *ModelElement*s. For example, *XmasTree* in Figure B-3 (a).
D16: StructuralModelElement	*StructuralModelElement* is a *ModelElement* representing a structural component or a property of the system. For example, *XmasTree* in Figure B-3 (a).
D17: BehavioralModelElement	*BehavioralModelElement* is a *ModelElement* representing the behavior of the system. For example, operation *monitorPressure*() in Figure B-3 (a).
D18: VariationPoint	*VariationPoint* is a *ModelElement* representing a *Variability* corresponding to a *DomainAsset* or an *ApplicationAsset* within the context of a *ProductLine*. For example, *treeType* in Figure B-3 (b).
D19: ConfigurableParameter	*ConfigurableParameter* is an instance of *VariationPoint*. For example, an instance of *treeType* (Figure B-3 (b)).
D20: Variant	*Variant (VA)* represents an alternative that can be used to configure a *ConfigurableParameter*. For example, in Figure B-3 (b), *VXT*, *HXT*, and *Mudline* are three *Variant*s of *treeType*.
D21: Constraint	*Constraint* is a *ModelElement*, which imposes certain conditions and limitations on other *ModelElement*s. E.g., in Figure B-3 (b), a constraint is defined on *XmasTree*, which is constraining the values of *waterDepth*.
D22: ConfigurationData	*ConfigurationData* represent configuration decisions made to configure a *ConfigurableParameter*.
D23: ConfigurationFile	*ConfigurationFile* represents a set of *ConfigurationData* corresponding to a *MemberProduct*.

Table B-16. Concept definitions of the CPS conceptual model

Concept	Definition and Example
D24: CPS	*CPS* is a system of systems (e.g., different physical units) that combines digital cyber technologies with physical processes where embedded computers and networks monitor and control physical processes using sensors and actuators often relying on closed feedback loops [1-3].
D25: Software	Software represents a software component of a CPS. For example, a software driver controlling a particular hardware device.
D26: CyberComponent	*CyberComponent* is a component on which a *Software* is deployed. For example, a controller or a computer.
D27: ComputationalComponent	*ComputationalComponent* is a *CyberComponent* responsible for performing computations. For example, a controller.
D28: CommunicationComponent	*CommunicationComponent* is a *CyberComponent* responsible for communication among different components. For example, network devices sending and receiving data.
D29: InterfacingComponent	*InterfacingComponent* is a component used to interact (monitor or manipulate) with the environment in which CPS is operating. For example, a sensor or an actuator.
D30: PhysicalComponent	*PhysicalComponent* represents a physical entity such as an engine or a human heart.
D31: Topology	*Topology* specifies how different components (i.e., *CyberComponent*s, *InterfacingComponent*s, *PhysicalComponent*s) are integrated.
D32: StateVariable	*StateVariable* is a variable showing the state of the CPS. For example, a variable representing the room temperature.
D33: ExternalAgent	*ExternalAgent* represents an external entity such as a human, an external service or an external system.
D34: PhysicalEnvironment	*PhysicalEnvironment* represents the environment in which CPS operates.
D35: PhysicalProperty	*PhysicalProperty* is a property of a *PhysicalComponent* constituting the *PhysicalEnvironment* of the CPS. For example, temperature and humidity level.
D36: ControlledVariable	*ControlledVariable* is a *PhysicalProperty* being controlled. For example, thermostat’s temperature.

D37: MonitoredVariable	*MonitoredVariable* is a *PhysicalProperty* being monitored. For example, room temperature.
D38: ComponentProperty	*ComponentProperty* is a property of an *InterfacingComponent* or a *CyberComponent*. For example, the accuracy of a sensor.

Table B-17. Concept definitions of the configuration process conceptual model

Concept	Definition and example
D39: ProductConfiguration	*ProductConfiguration* is an activity to derive a *MemberProduct* from a *ProductLine*.
D40: ConfigurationProcess	*ConfigurationProcess* is a process followed during the *ProductConfiguration*.
D41: ConfigurationSolution	*ConfigurationSolution* represents a configuration tool that assists to perform *ProductConfiguration*.
D42: ConfigurationStage	*ConfigurationStage* represents a stage in which a set of related *ConfigurableParameters* are configured by specific *Stakeholder*s.
D43: ConfigurationStep	*ConfigurationStep* represents a step in which one or more *ConfigurationDecision*s are made to configure a particular *ConfigurableParameter*.
D44: ConfigurationDecision	*ConfigurationDecision* represents a decision about selecting/assigning a *Variant* to configure a particular *ConfigurableParameter*.
D45: Stakeholder	*Stakeholder* represents a person or a group of persons concerning one or more *ConfigurationDecision*s.

Table B-18. Concept definitions for the conceptual model for modeling CPS product lines

Concept	Definition and example
D46: ModelingLanguage	*ModelingLanguage* represents a modeling language used to develop *PLEModel*s. For example, a Domain-Specific Modeling Language (e.g., Feature model [36]) or UML profiles (e.g., SimPL [19]).
D47: MetaModelElement	*MetaModelElement* is a *ModelElement* constituting the meta-model of a *ModelingLanguage*. For example, a *ModelElement* representing an optional feature in the meta-model of feature model.
D48: SoftwareStructuralModelElement	*SoftwareStructuralModelElement* represents a software component (e.g., a software driver for a particular device) or its property (e.g., an attribute).
D49: HardwareStructuralModelElement	*HardwareStructuralModelElement* represents a hardware component (e.g., network device, sensor, actuator) or its property (e.g., the accuracy of a sensor).
D50: ContextStructuralModelElement	*ContextStructuralModelElement* represents an *ExternalAgent*, a *PhysicalComponent* or a *PhysicalProperty* constituting the *PhysicalEnvironment* with which CPS interacts.
D51: Interaction	*Interaction* is *BehavioralModelElement*, which specifies how different components (e.g., software component, *CyberComponent*, *InterfacingComponent*, *PhysicalComponent*) interact/communicate with each other or how CPS interacts with *ExternalAgents* and *PhysicalEnvironment* [20].
D52: ApplicationLevelInteraction	*ApplicationLevelInteraction* represents the *Interaction* between *ContextStructuralModelElement*s and *SoftwareStructuralModelElement*s or among *SoftwareStructuralModelElement*s belonging to one physical unit of CPS. For example, the *Interaction* of human with a software application for smart buildings.
D53: InfrastructureLevelInteraction	*InfrastructureLevelInteraction* represents the *Interaction* among *HardwareStructuralModelElements* with a direct physical connection among them. For example, the *Interaction* of two *InterfacingComponent*s connected through a network cable.
D54: IntegrationLevelInteraction	*IntegrationLevelInteraction* represents the *Interaction* among

	*SoftwareStructuralModelElement*s, *HardwareStructuralModelElements*, and *ContextStructuralModelElement*s where these components either belong to different physical units of CPS or communicating through information networks. For example, the *Interaction* of software installed on two different *InterfacingComponent*s communicating through the Internet.
D55: View	*View* shows one aspect of the *PLEModel* by displaying the related *ModelElement*s and their relationships.
D56: ContextView	*ContextView* shows the *ContextStructuralModelElement*s and their relationships.
D57: SoftwareView	*SoftwareView* shows the *SoftwareStructuralModelElement*s and their relationships.
D58: InteractionView	*InteractionView* shows the *Interaction*s specifying how different components are interacting or how the CPS is interacting with *ExternalAgents* and *PhysicalEnvironment*.
D59: AllocationView	*AllocationView* shows the deployment of a *Software* on *CyberComponent*s.
D60: SystemView	*SystemView* is a composite view, which is composed of one *SoftwareView*, one to four *HardwareView*s, one *AllocationView*, and one *InteractionView*.
D61: HardwareView	*HardwareView* is an abstract view with four concrete views *MechanicalView*, *ElectricalView*, *ElectronicsView*, and *HydraulicsView* to show the commonalities corresponding to the *HardwareStructuralModelElement*s belonging to different disciplines of CPS.
D62: MechanicalView	*MechanicalView* shows the *HardwareStructuralModelElement*s belonging to mechanical discipline and their relationships.
D63: ElectricalView	*ElectricalView* shows the *HardwareStructuralModelElement*s belonging to electrical discipline and their relationships.
D64: ElectronicsView	*ElectronicsView* shows the *HardwareStructuralModelElement*s belonging to electronics discipline and their relationships.
D65: HydraulicsView	*HydraulicsView* shows the *HardwareStructuralModelElement*s belonging to hydraulics discipline and their relationships.
D66: VariabilityView	*VariabilityView* is an abstract view, which can be *SoftwareVariabilityView*, *HardwareVariabilityView*, *AllocationVariabilityView*, *InteractionVariabilityView*, *DomainVariabilityView*, *ContextVariabilityView*, or *ApplicationVariabilityView*.
D67: ContextVariabilityView	*ContextVariabilityView* shows the variabilities corresponding to the *ContextStructuralModelElement*s that can be resolved at pre-deployment, deployment, or post-deployment time.
D68: ApplicationVariabilityView	*ApplicationVariabilityView* shows the variabilities corresponding to a *MemberProduct* that can be resolved at deployment or post-deployment time only.
D69: SoftwareVariabilityView	*SoftwareVariabilityView* shows the variabilities corresponding to *Software* of CPS that can be resolved at pre-deployment, deployment, or post-deployment time.
D70: InteractionVariabilityView	*InteractionVariabilityView* shows the variabilities corresponding to the *Interaction* that can be resolved at pre-deployment time.
D71: AllocationVariabilityView	*AllocationVariabilityView* shows the variabilities corresponding to the deployment of a *Software* on a *CyberComponent*, which can be resolved at pre-deployment or deployment time.

D72: HardwareVariabilityView	*HardwareVariabilityView* is an abstract view with four concrete views *MechanicalVariabilityView*, *ElectricalVariabilityView*, *ElectronicsVariabilityView*, and *HydraulicsVariabilityView* to show the variabilities corresponding to the *HardwareStructuralModelElement*s belonging to different disciplines of CPS.
D73: MechanicalVariabilityView	*MechanicalVariabilityView* shows the variabilities corresponding to the *HardwareStructuralModelElement*s belonging to mechanical discipline that can be resolved at pre-deployment time.
D74: ElectricalVariabilityView	*ElectricalVariabilityView* shows the variabilities corresponding to the *HardwareStructuralModelElement*s belonging to electrical discipline that can be resolved at pre-deployment time.
D75: ElectronicsVariabilityView	*ElectronicsVariabilityView* shows the variabilities corresponding to the *HardwareStructuralModelElement*s belonging to electronics discipline that can be resolved at pre-deployment time.
D76: HydraulicsVariabilityView	*HydraulicsVariabilityView* shows the variabilities corresponding to the *HardwareStructuralModelElement*s belonging to hydraulics discipline that can be resolved at pre-deployment time.
D77: DomainVariabilityView	*DomainVariabilityView* is a composite view, which is composed of one *SoftwareVariabilityView*, one to four *HardwareVariabilityView*, and optionally one *AllocationVariabilityView* and *InteractionVariabilityView*.

Table B-19. Definitions of CPS-specific VP types

CPS-Specific VP Type	Definition and Example
D78: Descriptive-VP	Descriptive-VP is a *StringVP*, which requires setting a value in order to configure it. It can be defined for a textual *ComponentProperty* such as ID of a sensor or IP address of a sensor.
D79: DiscreteMeasurement-VP	DiscreteMeasurement-VP is an *IntegerVP*, which can be defined for a discrete numeric *ComponentProperty* (e.g., data transmissions per second for a sensor) or *PhysicalProperty* (e.g., the number of heartbeats per second) of CPS.
D80: ContinuousMeasurement-VP	ContinuousMeasurement-VP is a *RealVP*, which can be defined for a continuous numeric *ComponentProperty* (e.g., error in the measurement of a sensor) or *PhysicalProperty* (e.g., length and weight of a *PhysicalComponent*) of CPS.
D81: BinaryChoice-VP	BinaryChoice-VP is a *BinaryVP*, which can be defined for a Boolean *ComponentProperty* (e.g., whether a sensor keeps the events' log) or *PhysicalProperty* (e.g., the presence of a magnetic field) of CPS.
D82: PropertyChoice-VP	PropertyChoice-VP is a *NominalVP* or an *OrdinalVP* that requires selecting one value from a list of pre-defined values. PropertyChoice-VP can be defined for a *ComponentProperty* (e.g., connectionType with three possible values wired, 3G, and Wi-Fi) or a *PhysicalProperty* (e.g., different ranges for humidity level) of CPS.
D83: MeasurementUnitChoice-VP	MeasurementUnitChoice-VP is an *OrdinalVP*, which is derived from the *unit* of *PhysicalProperty* and *ComponentProperty*. For example, one can select meter, centimeter or millimeter as a unit for length.
D84: MeasurementPrecision-VP	MeasurementPrecision-VP is a *RealVP*, which is related to the degree of measurement precision for a *PhysicalProperty* or a *ComponentProperty*. For example, ± 0.0001 is an error in the measurement of a sensor.
D85: Multipart/Compound-VP	Multipart/Compound-VP is a *CompoundVP*, which can be specified for a *PhysicalProperty*, *ComponentProperty*, *CyberComponent*, *InterfacingComponent*, or *PhysicalComponent* that requires configuring several constituent *VariationPoint*s involved in it. For example, a Compound-VP with two *VariationPoint*s length and its unit.
D86: ComponentCardinality-VP	ComponentCardinality-VP is an *IntegerVP*, which is related to varying

	the number of instances of a *CyberComponent*, *InterfacingComponent*, or *PhysicalComponent*. For example, the number of temperature sensors.
D87: ComponentCollectionBoundary-VP	ComponentCollectionBoundary-VP is an *IntegerVP*, which is related to the upper limit and/or the lower limit of a collection of *CyberComponent*, *InterfacingComponent*, and/or *PhysicalComponent*. For example, the maximum and minimum numbers of sensors supported by a controller.
D88: ComponentChoice-VP	ComponentChoice-VP is a *NominalVP* or an *OrdinalVP*, which is about selecting a particular type of *CyberComponent*, *InterfacingComponent*, or *PhysicalComponent*. For example, selecting a speedometer sensor from several speedometers with various specifications.
D89: ComponentSelection-VP	ComponentSelection-VP is a *CollectionVP*, which is about selecting a subset of *CyberComponent*, *InterfacingComponent*, or/and *PhysicalComponent* from a collection of components. For example, selecting sensors for a *MemberProduct* from available sensors.
D90: TopologyChoice-VP	TopologyChoice-VP is a *NominalVP*, which is related to selecting a *Topology* from several alternatives. For example, how controllers are connected with different sensors and actuators.
D91: AllocationChoice-VP	AllocationChoice-VP is a *NominalVP*, which is about the deployment of *Software* on a *CyberComponent* (e.g., controller). For example, the same version of *Software* can be deployed on different controllers or different versions of *Software* can be deployed on the same controller.
D92: InteractionChoice-VP	InteractionChoice-VP is a *NominalVP*, which is about selecting an alternative for the *Interaction* specifying how different components involved in the *Interaction* will interact/communicate with each other.
D93: ConstraintSelection-VP	ConstraintSelection-VP is a *CollectionVP*, which is about selecting a subset of constraints for a specific *MemberProduct* from a set of constraints defined in the *ProductLine*.

10 Appendix B: OCL Constraints

Table B-20. OCL constraints for conceptual model of PLE (Figure B-2)

```
C1: context ApplicationArchitecture inv:
self.pLEModels->forAll(a:PLEModel|a.scope=PLEScope::Application or
a.scope=PLEScope::Context) and self.type=AssetType::Requirement implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Requirement) and
self.type=AssetType::Architecture implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Design) and
self.type=AssetType::Implementation implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Implementation) and
self.type=AssetType::TestCase implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Testing) and self.pLEModels-
>forAll(a:PLEModel|a.oclIsTypeOf(ResolutionModel))
```

Description: The scope of a PLE model representing application architecture should be application and/or context. Also, all the model elements in the PLE model belong to the same development phase (e.g., requirements) as the application architecture.

```
C2: context DomainArchitecture inv:
self.pLEModels->forAll(a:PLEModel|a.scope=PLEScope::ProductLine or
a.scope=PLEScope::Context) and self.type=AssetType::Requirement implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Requirement) and
self.type=AssetType::Architecture implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Design) and
self.type=AssetType::Implementation implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Implementation) and
self.type=AssetType::TestCase implies self.pLEModels-
>forAll(a:PLEModel|a.modelLevel=DevelopmentPhase::Testing) and self.pLEModels-
>forAll(a:PLEModel|a.oclIsTypeOf(VariabilityModel) or a.oclIsTypeOf(BaseModel))
```

Description: The scope of a PLE model representing domain architecture should be product line. Also, all the model elements in the PLE model belong to the same development phase (e.g., requirements) as the domain architecture.

```
C3: context ResolutionModel inv:
self.isPartiallyResolved implies self.hasVariability and not self.isPartiallyResolved
implies not self.hasVariability and not self.isPartiallyResolved implies
self.configurationFiles->forAll(a:ConfigurationFile|a.integrality=IntegralityType::Complete)
and self.configurationFiles->forAll(a:ConfigurationFile|a.level=self.modelLevel)
```

Description: A partially resolved resolution model has unresolved variabilities whereas a fully resolved resolution model has no unresolved variabilities. Furthermore, configuration files for a fully resolved resolution model should be completed and belong to the same development phase as the resolution model itself.

```
C4: context PLEModel inv:
self.oclIsTypeOf(BaseModel) implies not self.hasVariability and
self.oclIsTypeOf(VariabilityModel) implies self.hasVariability and
(self->selectByKind(VariabilityModel)->size()=self-
>selectByKind(BaseModel).variabilityModel->size())and (self->selectByKind(ResolutionModel)-
>size()=self->selectByKind(VariabilityModel).resolvedModel->size())
```

Description: In a product line, variability models are defined for a base model whereas resolution models are defined for variability models. The variabilities are defined in variability models only.

```
C5: context VariationPoint inv:
((self.type.oclIsTypeOf(Integer) or self.type.oclIsTypeOf(Real) or
self.type.oclIsTypeOf(String)) implies self.variants->size()=0) and (not
(self.type.oclIsTypeOf(Integer) or self.type.oclIsTypeOf(Real) or
self.type.oclIsTypeOf(String)) implies self.variants->size()>0)
```

Description: A list of variants is defined for all variation points except for Integer, Real, and String type variation points.

Table B-21.OCL constraints for conceptual model of configuration process (Figure B-5)

```
C6: context ConfigurationProcess inv:
self.isMultiStage implies self.configurationStage->size()>1 and self.isInteractive implies
self->exists(self.configurationStage->exists(self.configurationStage.configurationStep-
>exists(self.configurationStage.configurationStep.stakeholders->size()>0))) and
(self.automation=AutomationType::Manual implies self.productConfiguration.configurationData-
>forAll (d:ConfigurationData|not d.isAutoGenerated)) and
(self.automation=AutomationType::FullyAutomated implies
self.productConfiguration.configurationData->forAll (d:ConfigurationData|d.isAutoGenerated))
and (self.automation=AutomationType::SemiAutomated implies
(self.productConfiguration.configurationData->select(d:ConfigurationData|not
d.isAutoGenerated)->size()>0 and self.productConfiguration.configurationData-
>select(d:ConfigurationData|d.isAutoGenerated)->size()>0)) and self.isInteractive implies
(self.automation=AutomationType::Manual or self.automation=AutomationType::SemiAutomated)
and self.isIncremental implies self.configurationStage->size()>1
```

Description: An interactive and incremental multi-stage configuration process has more than one stage and at least one stakeholder where the configuration is performed manually or semi-automatically. In a fully automated (manually) configuration process, all configuration decisions are made automatically (manually) whereas in a semi-automated configuration process some configuration decisions are made manually and others automatically.

```
C7: context ConfigurationFile inv:
self.integrality=IntegralityType::Partial implies self.configurationData-
>exists(a:ConfigurationData|a.value=null) and self.integrality=IntegralityType::Complete
implies self.configurationData->forAll(a:ConfigurationData|a.value<>null)
```

Description: A partially completed configuration file has configuration data with null values whereas competed ones do not have configuration data with null values.

```
C8: context ConfigurationDecision inv:
self.isInferred implies self.configurationData-
>forAll(a:ConfigurationData|a.isAutoGenerated) and not self.isInferred implies
self.configurationData->forAll(a:ConfigurationData|not a.isAutoGenerated)
```

Description: The configuration data for inferred configuration decisions are automatically generated only.

Table B-22. OCL constraints for conceptual model for modeling CPS product lines (Figure B-7)

```
C9: context PLEModel inv:
 (self.oclIsKindOf(VariabilityModel) or self.oclIsKindOf(ResolutionModel) or
self.oclIsKindOf(BaseModel)) implies self.modelElements-
>selectByType(WellFormednessConstraint)->size()=0 and
self.modelElements->selectByType(MetaModelElement)->size()=0 and
self.oclIsTypeOf(VariabilityModel) implies self.views-
>forAll(a|a.oclIsTypeOf(VariabilityView)) and self.oclIsTypeOf(BaseModel) implies self.views-
>forAll(b|not b.oclIsTypeOf(VariabilityView)) and (self.modelElements-
>selectByKind(SoftwareStructuralModelElement)->size()>0 implies self.views-
>selectByKind(SoftwareView)->size()>0) and (self.modelElements-
>selectByKind(HardwareStructuralModelElement)->size()>0 implies self.views-
>selectByKind(HardwareView)->size()>0) and (self.modelElements-
>selectByKind(ContextStructuralModelElement)->size()>0 implies self.views-
>selectByKind(ContextView)->size()>0) and (self.modelElements->selectByKind(Interaction)-
>size()>0 implies self.views->selectByKind(InteractionView)->size()>0) and
(self.modelElements->selectByKind(SoftwareStructuralModelElement)-
>one(s:SoftwareStructuralModelElement|s.VP->size()>0) implies self.views-
>selectByKind(SoftwareVariabilityView)->size()>0) and (self.modelElements-
>selectByKind(HardwareStructuralModelElement)->one(s:HardwareStructuralModelElement|s.VP-
>size()>0) implies self.views->selectByKind(HardwareVariabilityView)->size()>0) and
(self.modelElements->selectByKind(ContextStructuralModelElement)-
>one(s:ContextStructuralModelElement|s.VP->size()>0) implies self.views-
>selectByKind(ContextVariabilityView)->size()>0) and (self.modelElements-
>selectByKind(Interaction)->one(s:Interaction|s.VP->size()>0) implies self.views-
>selectByKind(InteractionVariabilityView)->size()>0) and (self.modelElements-
>one(s:ModelElement|s.VP->size()>0 and s.VP.scope=PLEScope::Application) implies self.views-
>selectByKind(ApplicationVariabilityView)->size()>0)
```

Description: A PLE model can be a base model, variability model, or resolution model and these models have no well-formedness constraints or meta-model elements. The variability model has variability views and the base model has other non-variability views. If a PLE model has a specific type of model elements (e.g., SoftwareStructuralModelElement, Interaction) then the model should have corresponding views (SoftwareView, InteractionView). Similarly, if we have variation points corresponding to a specific model element then the model should have variability views (e.g., SoftwareVariabilityView, InteractionVariabilityView).

```
C10: context View inv:
self.oclIsKindOf(ContextVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclIsKindOf(ContextStructuralModelElement))) and
(self.oclIsKindOf(ApplicationVariabilityView) or self.oclIsKindOf(SoftwareVariabilityView))
implies (self.modelElements->forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements-
>selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclIsKindOf(SoftwareStructuralModelElement))) and
self.oclIsKindOf(InteractionVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclIsKindOf(Interaction))) and
self.oclIsKindOf(HardwareVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclIsKindOf(HardwareStructuralModelElement))) and
self.oclIsKindOf(AllocationVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclIsKindOf(HardwareStructuralModelElement) or
a.modelElement.oclIsKindOf(SoftwareStructuralModelElement))) and
self.oclIsKindOf(DomainVariabilityView) implies self.modelElements->size()=0 and
not self.oclIsKindOf(VariabilityView) implies self.modelElements->forAll(a|not
a.oclIsKindOf(VariationPoint)) and self.oclIsKindOf(ContextView) implies self.modelElements-
>forAll(a|a.oclIsKindOf(ContextStructuralModelElement)) and self.oclIsKindOf(InteractionView)
implies self.modelElements->forAll(a|a.oclIsKindOf(BehavioralModelElement) or
a.oclIsKindOf(StructuralModelElement)) and self.oclIsKindOf(SoftwareView) implies
self.modelElements->forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement)) and
self.oclIsKindOf(HardwareView) implies self.modelElements-
>forAll(a|a.oclIsKindOf(HardwareStructuralModelElement)) and self.oclIsKindOf(MechanicalView)
implies self.modelElements-
>forAll(a|a.oclAsType(HardwareStructuralModelElement).discipline=Discipline::Mechanical) and
self.oclIsKindOf(ElectricalView) implies self.modelElements-
>forAll(a|a.oclAsType(HardwareStructuralModelElement).discipline=Discipline::Electrical) and
self.oclIsKindOf(ElectronicsView) implies self.modelElements-
>forAll(a|a.oclAsType(HardwareStructuralModelElement).discipline=Discipline::Electronics) and
self.oclIsKindOf(HydraulicsView) implies self.modelElements-
>forAll(a|a.oclAsType(HardwareStructuralModelElement).discipline=Discipline::Hydraulics) and
self.oclIsKindOf(MechanicalVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclAsType(HardwareStructuralModelElement).discipline=D
iscipline::Mechanical)) and
self.oclIsKindOf(ElectricalVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclAsType(HardwareStructuralModelElement).discipline=D
iscipline::Electrical)) and
self.oclIsKindOf(ElectronicsVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclAsType(HardwareStructuralModelElement).discipline=D
iscipline::Electronics)) and
self.oclIsKindOf(HydraulicsVariabilityView) implies (self.modelElements-
>forAll(a|a.oclIsKindOf(VariationPoint)) and self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.modelElement.oclAsType(HardwareStructuralModelElement).discipline=D
iscipline::Hydraulics)) and self.oclIsTypeOf(ApplicationVariabilityView) implies
self.modelElements->selectByKind(VariationPoint)-
>forAll(a:VariationPoint|a.VP.scope=PLEScope::Application and
a.VP.bindingTime<>BindingTime::PreDeployment) and self.oclIsKindOf(AllocationView) implies
self.modelElements->forAll(a|a.oclIsKindOf(HardwareStructuralModelElement) or
a.oclIsKindOf(SoftwareStructuralModelElement))
```

Description: It ensures that correct (with respect to domain, PLE scope, and variability) type of model elements are presented in the views. For example, ContextVariabilityView should have only variation points defined corresponding to ContextStructuralModelElements. Abstract views (e.g., DomainVariabilityView) should not have model elements.

```
C11: context SystemView inv:
self.hardwareView->selectByType(MechanicalView)->size()<2 and self.hardwareView-
>selectByType(ElectronicsView)->size()<2 and self.hardwareView->selectByType(ElectricalView)-
>size()<2 and self.hardwareView->selectByType(HydraulicsView)->size()<2
```

Description: System view can have a maximum of one view for each domain (e.g., Electronics, Hydraulics).

```
C12: context DomainVariabilityView inv:
self.hardwareVariabilityView->selectByType(MechanicalVariabilityView)->size()<2 and
self.hardwareVariabilityView->selectByType(ElectronicsVariabilityView)->size()<2 and
self.hardwareVariabilityView->selectByType(ElectricalVariabilityView)->size()<2 and
self.hardwareVariabilityView->selectByType(HydraulicsVariabilityView)->size()<2
```
Description: Domain variability view can have a maximum of one variability view for each domain (e.g., Electronics, Hydraulics).

```
C13: context VariationPoint inv:
self.scope=PLEScope::Context implies
self.modelElement.oclIsKindOf(ContextStructuralModelElement) and
self.scope=PLEScope::Application implies
self.modelElement.oclIsKindOf(SoftwareStructuralModelElement) and
(self.type.oclIsTypeOf(Integer) or self.type.oclIsTypeOf(Real)) implies (self.lowerLimit-
>size()=1 and self.upperLimit->size()=1) and (not (self.type.oclIsTypeOf(Integer) or
self.type.oclIsTypeOf(Real))) implies (self.lowerLimit->size()=0 and self.upperLimit-
>size()=0)
```
Description: The PLE scope (i.e., context, application, product line) of the variation points should be the same as the corresponding model elements. For integer and real type variation points only, lower and upper limits must be defined.

```
C14: context Interaction inv:
self.isHomogeneous implies ((self.source-
>forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement)) and self.target-
>forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement))) or (self.source-
>forAll(a|a.oclIsKindOf(HardwareStructuralModelElement)) and self.target-
>forAll(a|a.oclIsKindOf(HardwareStructuralModelElement)) and self.source-
>collect(a|a.oclAsType(HardwareStructuralModelElement).discipline)->asSet()=self.target-
>collect(a|a.oclAsType(HardwareStructuralModelElement).discipline)->asSet()and self.source-
>collect(a|a.oclAsType(HardwareStructuralModelElement).discipline)->asSet()->size()=1) or
(self.source->forAll(a|a.oclIsKindOf(ContextStructuralModelElement)) and self.target-
>forAll(a|a.oclIsKindOf(ContextStructuralModelElement)))) and not self.isDirect implies
self.interaction->size()>0 and self.isDirect implies self.interaction->size()=0 and
self.oclIsTypeOf(ApplicationLevelInteraction) implies ((self.source-
>forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement) or
a.oclIsKindOf(ContextStructuralModelElement))) and (self.target -
>forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement) or
a.oclIsKindOf(ContextStructuralModelElement))) and (self.source->union(self.target)-
>includes(SoftwareStructuralModelElement)))and
self.oclIsTypeOf(InfrastructureLevelInteraction) implies ((self.source-
>forAll(a|a.oclIsKindOf(HardwareStructuralModelElement))) and (self.target -
>forAll(a|a.oclIsKindOf(HardwareStructuralModelElement))) and (self.isDirect)) and
self.oclIsTypeOf(IntegrationLevelInteraction) implies ((self.source-
>forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement) or
a.oclIsKindOf(HardwareStructuralModelElement) or
a.oclIsKindOf(ContextStructuralModelElement))) and (self.target -
>forAll(a|a.oclIsKindOf(SoftwareStructuralModelElement) or
a.oclIsKindOf(HardwareStructuralModelElement) or
a.oclIsKindOf(ContextStructuralModelElement))))
```
Description: A homogenous interaction means that both source and target elements are of the same type. Direct interactions do not have intermediate interactions. In application-level interaction, the source and target elements are software or contextual structural elements whereas in infrastructure level interaction they can be only hardware elements. Furthermore, infrastructure level interactions are always direct. In the case of integration level interactions, source and target elements can be software, hardware, or contextual structural elements.

```
C15: context VariabilityModel inv:
self.scope=PLEScope::ProductLine implies self.modelElements-
>forAll(a|a.oclAsType(VariationPoint).scope=PLEScope::ProductLine) and
self.scope=PLEScope::Application implies self.modelElements-
>forAll(a|a.oclAsType(VariationPoint).scope=PLEScope::Application) and
self.scope=PLEScope::Context implies self.modelElements-
>forAll(a|a.oclAsType(VariationPoint).scope=PLEScope::Context)
```
Description: All the variation points in a variability model should have the same scope as the variability model itself (e.g., product line, application or context)

```
C16: context ResolutionModel inv:
self.isPartiallyResolved implies self.hasVariability and not self.isPartiallyResolved implies
not self.hasVariability and self.isPartiallyResolved implies self.configurationFiles-
>exists(a:ConfigurationFile|a.integrality=IntegralityType::Partial) and not
self.isPartiallyResolved implies self.configurationFiles-
>forAll(a:ConfigurationFile|a.integrality=IntegralityType::Complete) and
self.configurationFiles->forAll(a:ConfigurationFile|a.level=self.modelLevel)
```
Description: Partially resolved resolution model must have unresolved variabilities and at least one configuration file that is not complete. Also, all configuration files and resolution model should belong to the same phase of the development lifecycle.

```
C17: context ConfigurableParameter inv:
self.status=ConfigurationStatus::Configured implies (self.selectedVariant->size()=1 and
self.configurationData->size()=1) and self.status=ConfigurationStatus::Unconfigured implies
(self.selectedVariant->size()=0 and self.configurationData->size()=0)
```
Description: Only a configurable parameter that is configured must have one selected variant and one configuration data.

```
C18: context Constraint inv:
self.constrainedElements->forAll(a:ModelElement|not a.oclIsKindOf(Constraint)) and
self.oclIsTypeOf(WellFormednessConstraint) implies self.constrainedElements-
>forAll(a:ModelElement|a.oclIsKindOf(MetaModelElement)) and
not self.oclIsTypeOf(WellFormednessConstraint) implies self.constrainedElements-
>forAll(a:ModelElement|not a.oclIsKindOf(MetaModelElement))
```

Description: All constraints except well-formedness constraints are defined on model elements (except constraint and meta-model elements) whereas well-formedness constraints are defined on meta-model elements.

Table B-23. OCL constraints for conceptual model of basic data types (Figure B-8) [16]

```
C19: context Array, Set (Sequence, OrderedSet) inv:
(self.constantElements->size()=0 and self.variableElements-
>select(a|a.oclIsKindOf(Collection))->size()=0 and self.variableElements-
>forAll(a,b|a.type=b.type)) or (self.variableElements->size()=0 and self.constantElements-
>forAll(a,b|a.type=b.type)) or (self.constantElements->size()=0 and self.variableElements-
>size()=self.variableElements->select(a:Variable|a.type.oclIsKindOf(Collection))->size()and
self.variableElements->forAll(v1, v2|(v1.type.oclAsType(Collection).constant Elements-
>size()=0 and v1.type.oclAsType(Collection).variableElements->forAll(v3:Variable
|v3.type=v2.type.oclAsType(Collection).variableElements->asSequence()->first().type)) or
(v1.type.oclAsType( Collection).variableElements->size()=0 and
v1.type.oclAsType(Collection).constantElements-
>forAll(v3:Constant|v3.type=v2.type.oclAsType(Collection).constantElements->asSequence()-
>first().type))))
```

Description: The elements in the array, set, sequence, and ordered set are homogenous and they can be variables, constants, or collections.

```
C20: context Record (Set, OrderedSet) inv:
self.variableElements->select(self.variableElements->forAll(a,b|a=b))->isEmpty()and
self.constantElements->select(self.constantElements->forAll(a,b|a=b))->isEmpty()
```

Description: The elements in the record, set, and ordered sets are unique.

```
C21: context Sequence inv:
self.variableElements->asSet()->size()>1 implies self.variableElements->asSequence()-
>reverse() <> self.variableElements->asSequence()
```

Description: The elements in the sequence are not unique but they have a specific order.

```
C22: context OrderedSet inv:
self.variableElements->asOrderedSet()->reverse() <> self.variableElements->asOrderedSet()and
self.constantElements->asOrderedSet()->reverse() <> self.constantElements->asOrderedSet()
```

Description: The elements in the ordered set are unique and have a specific order.

Table B-24. OCL constraints for constraints classification (Figure B-10)

```
C23: context Constraint inv:
self.isHardConstraint implies self.evaluationResult and
self.source=InternalTypes::Source::Mined implies not self.isHardConstraint and
self.source=InternalTypes::Source::EnforcedByDevelopmentProcess implies
self.isHardConstraint and self.constrainedElements-
>forAll(a|a.oclIsTypeOf(VPClassification::VPTypes::VariationPoint) or
a.oclIsTypeOf(PLE::Variant) or a.oclIsTypeOf(Modeling::MetaModelElement)) and
self.oclIsTypeOf(WellFormednessConstraint) implies self.constrainedElements-
>forAll(a|a.oclIsKindOf(Modeling::MetaModelElement)) and
not self.oclIsTypeOf(WellFormednessConstraint) implies self.constrainedElements-
>forAll(a|not a.oclIsKindOf(Modeling::MetaModelElement))
```

Description: A hard constraint must be true. Mined constraints are soft constraints whereas constraints enforced by the development process are hard constraints. Well-formedness constraints are defined on meta-model elements whereas other types of constraints are defined on variations points and their variants.

```
C24: context ConfigurationConstraint inv:
not self.source<>Source::DerivedFromModelingLanguage
```

Description: ConfigurationConstraints can be originated from different sources (e.g., user defined, mined, derived from system specifications) except for the modeling language.

```
C25: context WellFormednessConstraint inv:
(self.source=Source::UserDefined or self.source=Source::DerivedFromModelingLanguage) and
self.owningPhase->size()=0 and self.level=WellFormednessConstraintLevel::VariabilityModel
```

Description: WellformednessConstraints are hard constraints derived from the modeling language. Such constraints are applied on variability model and they do not belong to a particular phase of development lifecycle.

```
C26: context ConformanceConstraint inv:
(self.source=Source::UserDefined or self.source=Source::DerivedFromSystemSpecifications or
self.source=Source::DerivedFromModelingLanguage) and self.isHardConstraint
```

Description: ConformanceConstraints are hard constraints which can be either user defined or derived from system specifications and the modeling language.

```
C27: context ConsistencyConstraint inv:
self.isHardConstraint and (self.source=Source::UserDefined or
self.source=Source::DerivedFromSystemSpecifications) and self.owningPhase->size()=0 and
self.atView=ConstrainingView::WithinView implies self.constrainedElements-
>forAll(a:ModelElement, b:ModelElement|a.view=b.view) and
self.atView=ConstrainingView::CrossView implies (self.constrainedElements-
>exists(a:ModelElement, b:ModelElement|a.view<>b.view)) and
self.atModel=ConstrainingModel::IntraModel implies self.constrainedElements-
>forAll(a:ModelElement, b:ModelElement|a.pLEModel=b.pLEModel) and self.
atModel=ConstrainingModel::InterModel implies self.constrainedElements-
>exists(a:ModelElement, b:ModelElement|a.pLEModel<>b.pLEModel)
```

Description: ConsistencyConstraints are hard constraints which are either user defined or derived from system specifications and they do not belong to a particular phase of development lifecycle. Furthermore, the scope of the consistency constraints can be specified based on two

properties ConstrainingView and atModel.
C28: context DecisionOrderingConstraint **inv:** **self**.isHardConstraint **and self**.owningPhase->size()=0 **Description:** DecisionOrderingConstraints are hard constraints which do not belong to a particular phase of development lifecycle.
C29: context DecisionInferenceConstraint **inv:** (**self**.source=Source::EnforcedByDevelopmentProcess) **and self**.isHardConstraint **Description:** DecisionInferenceConstraints are hard constraints enforced by the development process.
C30: context OptimizationConstraint **inv:** **not self**.isHardConstraint **and** (**self**.source=Source::UserDefined **or self**.source=Source::Mined **or self**.source=Source::DerivedFromSystemSpecifications) **Description:** OptimizationConstraints are soft constraints and they are either user defined, mined, or derived from the system specifications.

Table B-25. OCL constraints for functionalities (Figure B-11)

C31: context ConflictDetection **inv:** **self**.constraints->forAll(a|**not** a.**oclIsKindOf**(OptimizationConstraint)) **and self**.constraints->forAll(a:Constraint|a.isHardConstraint) **and self**.debugging.violations ->forAll(a:Violation|a.violationType=ConstraintType::ConflictingConstraints) **Description:** ConflictingConstraints type violation will occur if there is a conflict only when all the constraints are hard constraints and none of them is OptimizationConstraint.
C32: context Violation **inv:** **self**.violationType=ConstraintType::ConflictingConstraints **implies self**.violatedConstraints->size()>1 **and self**.violationType<>ConstraintType::ConflictingConstraints **implies self**.violatedConstraints->size()=1 **and self**.violationType=ConstraintType::ConflictingConstraints **implies self**.causingParameters->size()=0 **and self**.violationType<>ConstraintType::ConflictingConstraints **implies self**.causingParameters->size()>0 **Description:** ConflictingConstraints type violation involves more than one constraint and no configurable parameter whereas other types of violations involve one constraint and at least one configurable parameter causing the violation.
C33: context ResolvingViolation **inv:** **self**.violations->forAll(a:Violation|a.violationType<>ConstraintType::ConflictingConstraints) **Description:** ResolvingViolation ensures that there are no conflicting constraints.
C34: context ConformanceChecking **inv:** **self**.debugging.violations->forAll(a:Violation|a.violationType=ConstraintType::ConformanceConstraint) **Description:** ConformanceChecking checks only violations related to conformance constraints.
C35: context WellFormednessChecking **inv:** **self**.debugging.violations->forAll(a:Violation|a.violationType=ConstraintType::WellFormednessConstraint) **Description:** WellFormednessChecking checks only violations related to well-formedness constraints.
C36: context ConsistencyChecking **inv:** **self**.debugging.violations->forAll(a:Violation|a.violationType=ConstraintType::ConsistencyConstraint) **Description:** ConsistencyChecking checks only violations related to consistency constraints.
C37: context RedundancyDetection **inv:** **self**.scope=RedundancyDetectionScope::InterConfigurationFiles **implies self**.configurationFiles->size()>1 **and self**.scope=RedundancyDetectionScope::IntraConfigurationFile **implies self**.configurationFiles->size()=1 **Description:** RedundancyDetection checks redundancy within one configuration file if scope is IntraConfigurationFile otherwise within more than one configuration files.
C38: context Change **inv:** **self**.model.**oclIsKindOf**(VariabilityModel) **or self**.model.**oclIsKindOf**(ResolutionModel) **Description:** A change occurs either in the variability model or resolution model.
C39: context RevertingDecision **inv:** **self**.debugging.wellFormednessChecking->size()=0 **and self**.debugging.conformanceChecking->size()=0 **and self**.debugging.conflictDetection->size()=0 **and self**.debugging.redundancyDetection->size()=0 **and self**.debugging.incompletenessDetection->size()=0 **Description:** Reverting a configuration decision should not cause any violation (e.g., related to incompleteness, conformance, wellformedness, redundancy, conflicts).
C40: context DecisionInference **inv:** **self**.debugging.resolvingViolation->size()=0 **and self**.debugging.conflictDetection->size()=0 **and self**.debugging.redundancyDetection->size()=0 **and self**.debugging.incompletenessDetection->size()=0 **Description:** Inferred configuration decisions should not cause any violation (e.g., related to incompleteness, redundancy, conflicts).
C41: context ConfigurationOptimization **inv:** **self**.debugging.wellFormednessChecking->size()=0 **and self**.debugging.resolvingViolation->size()=0 **and self**.debugging.conflictDetection->size()=0 **and self**.debugging.redundancyDetection->size()=0 **and self**.debugging.incompletenessDetection->size()=0 **Description:** Configuration optimization should not cause any violation (e.g., related to incompleteness, well-formedness, redundancy, conflicts).

11 Appendix C: Formal Definitions of Constraints

In this section, we present the formal definitions of eight types of *Constraint* presented in Section 4.3. To formalize each type of the *Constraint* in Appendix C and functionalities of an automated *ConfigurationSolution* in Appendix D, we define some basic notations corresponding to the concepts (classes, attributes, and roles) presented in Figure B-2, Figure B-5, and Figure B-7, as follow:

First, we clarify the notations that we used to access the different concepts presented in the conceptual models (Figure B-2, Figure B-5, and Figure B-7) and then we define the notations used to formalize *Constraint* types (and functionalities of *ConfigurationSolution*). Let C_1 and C_2 be two classes in the conceptual model, where C_1 has a relationship (i.e., bidirectional association, unidirectional association, composition, and aggregation) with C_2 with role name r. Class C_1 and C_2 have attribute a_1 and a_2 respectively. Then, to access a_1, r, and a_2 from class C_1, we use notations $(C_1.a_1)$, $(C_1.r)$, and $(C_1.r.a_2)$ respectively. Notation $A \coloneqq B$ represents an assignment operation where B is assigned to A. Similarly, $A = B$ represents equality operation (i.e., A is equal to B) and $A \rightarrow B$ shows implies relationship where A implies B.

Let PL be a *ProductLine* containing nvp *VariationPoint*s, $VP = \{vp_1, vp_2, .., vp_{nvp}\}$. For each vp_i, $VA_i = \{v_{i1}, v_{i2}, .., v_{inv}\}$ is a set of possible *Variant*s and cp_{ij} is a *ConfigurableParameter* (i.e., an instance of *VariationPoint* vp_i). $CP_{iC} = \{cp_{i1}, cp_{i2}, .., cp_{inc}\}$ is a set of configured *ConfigurableParameter*s for vp_i such that $\forall\ cp_{ij} \in CP_{iC}, (cp_{ij}.status = Configured \wedge (cp_{ij}.selectedVariant = v_{is}) \wedge (v_{is} \in VA_i) \wedge$ $(cp_{ij}.cd.status = Valid))$, where $cp_{ij}.cd$ represent the *ConfigurationData* corresponding to cp_{ij} and v_{is} is the *Variant* assigned/selected to configure cp_{ij}. $CP_{iUC} = \{cp_{nic+1}, cp_{nic+2}, .., cp_{inuc}\}$ is a set of un-configured *ConfigurableParameter*s for vp_i such that $\forall\ cp_{ik} \in CP_{iUC}, (\ cp_{ik}.status = Unconfigured)$. If vp_i is not instantiated yet, then $|CP_{iC}| = |CP_{iUC}| = \emptyset$. Let $Instance(vp_i)$ be a function that instantiates vp_i and returns a *ConfigurableParameter* cp_{ij}.

Let P be a partially configured *MemberProduct* of PL and RM is the *ResolutionModel* corresponding to *MemberProduct* P containing ncp *ConfigurableParameter*s such that $RM = \{CP_C \cup CP_{UC}\}$, where $CP_C = \{CP_{1C} \cup CP_{2C} \cup \ldots \cup CP_{nvpC}\}$ and $CP_{UC} = \{CP_{1UC} \cup CP_{2UC} \cup \ldots \cup CP_{nvpUC}\}$. Corresponding to CP_C, CD is a set of *ConfigurationData* such that $|CP_C| = |CD|$. F_i is a *ConfigurationFile* representing a partial or complete *ResolutionModel* RM corresponding to a *MemberProduct* P, which contains in *ConfigurableParameter*s $CP_i = \{cp_1, cp_2, .., cp_{in}\}$ and the corresponding *ConfigurationData* $CD_i = \{cp_1.cd, cp_2.cd, .., cp_{in}.cd\}$. Let $C = \{c_1, c_2, .., c_{nc}\}$ be a set of *Constraint*s in the context of PLE, where each *Constraint* c_i is defined over one or more *VariationPoint*s (or *MetaModelElemenet*s in case of *WellFormednessConstraint*). Let $VAR(c_i)$ be a function that gives a set of elements (e.g., *VariationPoint*s, *MetaModelElemenet*s) constrained by c_i and $VAL(c_i)$ be a function that gives a set of possible values (e.g., a subset of *Variant*s corresponding to a *VariationPoint*(s), configuration order for certain *VariationPoint*s) that satisfy the constraint c_i.

```
$\forall\ c \in C_C,$
    $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c))$
    $\forall\ vp_i \in VP_C,$
        $(cp_{ij} \coloneqq Instance(vp_i)) \wedge \left(CP_{iUC} \coloneqq (CP_{iUC} \cup cp_{ij})\right)$
        $c.evaluationResult = True \rightarrow (\exists v_{is} |\ (v_{is} \in VA_C) \wedge (v_{is} \in VA_i)) \wedge (cp_{ij}.status \coloneqq Configured) \wedge$
        $(cp_{ij}.selectedVariant \coloneqq v_{is}) \wedge (CP_{iC} \coloneqq (CP_{iC} \cup cp_{ij})) \wedge (CP_{iUC} \coloneqq (CP_{iUC} \backslash cp_{ij})) \wedge$
        $(CD \coloneqq (CD \cup cp_{ij}.cd))))|\ (\nexists\ cp_{xy}|\ ((cp_{xy} \in CP_C) \wedge (cp_{xy}.cd.status = Invalid)))$
```

Listing 8: Formal definition of DecisionInferenceConstraints

```
$\forall\ c \in C_{VD},$
    $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c))$
    $IF: (c.type = VP - VP \wedge c.relation = Requires)\ THEN: c.evaluationResult = True \rightarrow (\exists\ vp_i, vp_j|\ ((vp_i, vp_j \in VP_C) \wedge$
        $((cp_{ik}.status = Configured \rightarrow cp_{jl}.status = Configured)|(cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}))))$
4.      $IF: (c.type = VP - VP \wedge c.relation = Excludes)\ THEN: c.evaluationResult = True \rightarrow (\exists\ vp_i, vp_j|\ ((vp_i, vp_j \in VP_C) \wedge$
        $((cp_{ik}.status = Configured \rightarrow cp_{jl}.status \neq Configured)|(cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}))))$
5.      $IF: (c.type = VP - VA \wedge c.relation = Requires)\ THEN: c.evaluationResult = True \rightarrow (\exists\ vp_i, vp_j, v_{js}|\ ((vp_i, vp_j \in VP_C) \wedge$
        $(v_{js} \in VA_C) \wedge (v_{js} \in VA_j) \wedge ((cp_{ik}.status = Configured \rightarrow cp_{jl}.selectedVariant = v_{js})|(cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}))))$
6.      $IF: (c.type = VP - VA \wedge c.relation = Excludes)\ THEN: c.evaluationResult = True \rightarrow (\exists\ vp_i, vp_j, v_{js}|\ ((vp_i, vp_j \in VP_C) \wedge$
        $(v_{js} \notin VA_C) \wedge (v_{js} \in VA_j) \wedge ((cp_{ik}.status = Configured \rightarrow cp_{jl}.selectedVariant \neq v_{js})|(cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}))))$
7.      $IF: (c.type = VA - VA \wedge c.relation = Requires)\ THEN: c.evaluationResult = True \rightarrow (\exists\ vp_i, vp_j, v_{is}, v_{js}|\ ((vp_i, vp_j \in VP_C)$
        $\wedge (v_{is}, v_{js} \in VA_C) \wedge (v_{is} \in VA_i) \wedge (v_{js} \in VA_j) \wedge ((cp_{ik}.selectedVariant = v_{is} \rightarrow cp_{jl}.selectedVariant = v_{js})|(cp_{ik} \in CP_{iC}) \wedge$
        $(cp_{jl} \in CP_{jC}))))$
8.      $IF: (c.type = VA - VA \wedge c.relation = Excludes)\ THEN: c.evaluationResult = True \rightarrow (\exists\ vp_i, vp_j, v_{is}, v_{js}|\ ((vp_i, vp_j \in VP_C)$
        $\wedge (v_{is} \in VA_C) \wedge (v_{js} \notin VA_C) \wedge (v_{is} \in VA_i) \wedge (v_{js} \in VA_j) \wedge ((cp_{ik}.selectedVariant = v_{is} \rightarrow cp_{jl}.selectedVariant \neq v_{js})|$
        $(cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}))))$
```

Listing 9: Formal definition of VariabilityDependencyConstraints

```
1.  $\forall\ c \in C_{WF},$
2.      $(MME_C \coloneqq VAR(c))$
3.      $\forall\ mme_i \in MME_C,$
4.              $(me_{ij} \coloneqq Instance(mme_i))$
5.              $c \rightarrow (\nexists\ me_{ij}|c.evaluationResult = False)$
```

Listing 10: Formal definition of WellFormednessConstraints (New Definition)

```
1.  $\forall\ c \in C_{CF},$
2.      $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c))$
3.      $\forall\ vp_i \in VP_C,$
4.              $c.evaluationResult = True \rightarrow (\nexists\ cp_{ij}|((cp_{ij} \in CP_{iC}) \wedge (cp_{ij}.selectedVariant \notin VA_C) \wedge$
                $(cp_{ij}.cd.status = Invalid)))$
```

Listing 11: Formal definition of ConformanceConstraints

```
1.  $\forall\ c \in C_{CS},$
2.      $(VP_C \coloneqq VAR(c)) \wedge (|VP_C| \geq 2) \wedge (VA_C \coloneqq VAL(c))$
3.      $\forall\ vp_i, vp_j \in VP_C,$
4.              $IF: (c.atModel = IntraModel \wedge c.atView = WithinView) THEN: c.evaluationResult = True \rightarrow (\nexists\ cp_{ik}, cp_{jl}|$
                $((cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}) \wedge (cp_{ik}.selectedVariant \notin VA_C) \wedge (cp_{jl}.selectedVariant \notin VA_C) \wedge$
                $(cp_{ik}.cd.status = Invalid) \wedge (cp_{jl}.cd.status = Invalid) \wedge (vp_i, vp_j$ belong to the same PLEModel and View$)))$
5.              $IF: (c.atModel = IntraModel \wedge c.atView = CrossView) THEN: c.evaluationResult = True \rightarrow (\nexists\ cp_{ik}, cp_{jl}|$
                $((cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}) \wedge (cp_{ik}.selectedVariant \notin VA_C) \wedge (cp_{jl}.selectedVariant \notin VA_C) \wedge$
                $(cp_{ik}.cd.status = Invalid) \wedge (cp_{jl}.cd.status = Invalid) \wedge (vp_i, vp_j$ belong to same PLEModel but different Views$)))$
6.              $IF: (c.atModel = InterModel \wedge c.atView = WithinView) THEN: c.evaluationResult = True \rightarrow (\nexists\ cp_{ik}, cp_{jl}|$
                $((cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}) \wedge (cp_{ik}.selectedVariant \notin VA_C) \wedge (cp_{jl}.selectedVariant \notin VA_C) \wedge$
                $(cp_{ik}.cd.status = Invalid) \wedge (cp_{jl}.cd.status = Invalid) \wedge (vp_i, vp_j$ belong to different PLEModels but same View$)))$
```

```
7.          $IF: (c.atModel = InterModel \wedge c.atView = CrossView) THEN: c.evaluationResult = True \rightarrow$
    $(\nexists cp_{ik}, cp_{jl}|$
            $((cp_{ik} \in CP_{iC}) \wedge (cp_{jl} \in CP_{jC}) \wedge (cp_{ik}.selectedVariant \notin VA_C) \wedge (cp_{jl}.selectedVariant \notin VA_C) \wedge$
            $(cp_{ik}.cd.status = Invalid) \wedge (cp_{jl}.cd.status = Invalid) \wedge (vp_i, vp_j$ *belong to different PLEModels and Views*$)))$
```

Listing 12: Formal definition of ConsistencyConstraints

```
1.  $\forall c \in C_{DO},$
2.      $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c))$
3.      $\forall vp_i, vp_j \in VP_C,$
4.          $IF: c \rightarrow Instance(vp_i).configurationStep < Instance(vp_j).configurationStep$
5.          $THEN: c.evaluationResult = True \rightarrow (\nexists cp_{ik}, cp_{jl}|((cp_{ik}.status = Unconfigured) \wedge$
            $(cp_{jl}.status = Configured)))$
```

Listing 13: Formal definition of DecisionOrderingConstraints

```
1.  $\forall c \in C_{DI},$
2.      $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c))$
3.      $\exists vp_i \in VP_C|(|VA_C \cap VA_i| = 1),$
4.          $(cp_{ij} \coloneqq Instance(vp_i)) \wedge (CP_{iUC} \coloneqq (CP_{iUC} \cup cp_{ij}))$
5.          $c.evaluationResult = True \rightarrow (\exists v_{is}| (v_{is} \in VA_C) \wedge (v_{is} \in VA_i)) \wedge (cp_{ij}.status \coloneqq Configured) \wedge$
            $(cp_{ij}.selectedVariant \coloneqq v_{is}) \wedge (CP_{iC} \coloneqq (CP_{iC} \cup cp_{ij})) \wedge (CP_{iUC} \coloneqq (CP_{iUC} \backslash cp_{ij})) \wedge$
            $(cp_{ij}.cd.isAutoGenerated = True) \wedge (CD \coloneqq (CD \cup cp_{ij}.cd))))| (\nexists cp_{xy}| ((cp_{xy} \in CP_C) \wedge$
            $(cp_{xy}.cd.status = Invalid)))$
```

Listing 14: Formal definition of DecisionInferenceConstraints

```
1.  $\forall c \in C_C,$
2.      $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c))$
3.      $IF: |VP_C| = 1\ THEN:$
4.          $IF: c.type = Minimization\ THEN: c.EvaluationResult = True \rightarrow \forall cp_{ij} \in CP_{iC},$
            $(cp_{ij}.selectedVariant.optimizationMeasure_k \leq \forall v_{il} \in VA_i, v_{il}.optimizationMeasure_k)) \wedge$
            $((\nexists cp_{xy}| ((cp_{xy} \in CP_C) \wedge (cp_{xy}.cd.status = Invalid)))$
5.          $IF: c.type = Maximization\ THEN: c.EvaluationResult = True \rightarrow \forall cp_{ij} \in CP_{iC},$
            $(cp_{ij}.selectedVariant.optimizationMeasure_k \geq \forall v_{il} \in VA_i, v_{il}.optimizationMeasure_k)) \wedge$
            $((\nexists cp_{xy}| ((cp_{xy} \in CP_C) \wedge (cp_{xy}.cd.status = Invalid)))$
6.      $IF: |VP_C| > 1\ THEN:$
7.          $IF: c.type = Minimization\ THEN: c.EvaluationResult = True \rightarrow \forall vp_i \in VP_C, (\forall v_{il} \in VA_i,$
            $(\sum_{i=1}^{i=|VP_C|} \sum_{j=1}^{j=|CP_{iC}|} cp_{ij}.selectedVariant.optimizationMeasure_k \leq$
            $\sum_{i=1}^{i=|VP_C|} \sum_{j=1}^{j=|CP_{iC}|} v_{il}.optimizationMeasure_k) \wedge ((\nexists cp_{xy}| ((cp_{xy} \in CP_C) \wedge (cp_{xy}.cd.status =$
    $Invalid))))$
8.          $IF: c.type = Maximization\ THEN: c.EvaluationResult = True \rightarrow \forall vp_i \in VP_C, (\forall v_{il} \in VA_i,$
            $(\sum_{i=1}^{i=|VP_C|} \sum_{j=1}^{j=|CP_{iC}|} cp_{ij}.selectedVariant.optimizationMeasure_k \geq$
            $\sum_{i=1}^{i=|VP_C|} \sum_{j=1}^{j=|CP_{iC}|} v_{il}.optimizationMeasure_k) \wedge ((\nexists cp_{xy}| ((cp_{xy} \in CP_C) \wedge (cp_{xy}.cd.status =$
    $Invalid))))$

    where $optimizationMeasure_k$ is a particular optimization measure constrained by $c$.
```

Listing 15: Formal definition of OptimizationConstraints

12 Appendix D: Formal Definitions of Functionalities

In this section, we present the formal definitions of 14 functionalities of the *ConfigurationSolution* presented in Section 5.1. To formally define each functionality, we use the following template.

- *Inputs:* The inputs of the function.
- *Outputs:* The outputs of the function.
- *Definition:* A concise and precise definition of the functionality using mathematical notations based on set theory.

1.1 DecisionInference

Inputs: Sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, *ConfigurationData* CD, and *DecisionInferenceConstraint*s C_{DI}.
Outputs: Updated sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, and *ConfigurationData* CD.

```
1.  $\boldsymbol{DI(CP_C, CP_{UC}, CD, C_{DI}) \overset{\text{def}}{=} CP_C, CP_{UC}, CD}$
2.  $\forall\, \boldsymbol{c} \in \boldsymbol{C_{DI}},$
3.      $(\boldsymbol{VP_C} \coloneqq \boldsymbol{VAR(c)}) \wedge (\boldsymbol{VA_C} \coloneqq \boldsymbol{VAL(c)})$
4.      $\forall\, \boldsymbol{vp_i} \in \boldsymbol{VP_C}, \forall\, \boldsymbol{cp_{ij}} \in \boldsymbol{CP_{iUC}}$
5.              $(\boldsymbol{cp_{ij}.status} \coloneqq \boldsymbol{Configured}) \wedge (\boldsymbol{cp_{ij}.selectedVariant} \coloneqq \boldsymbol{v_{is}} | ((\boldsymbol{v_{is}} \in \boldsymbol{VA_C}) \wedge (\boldsymbol{v_{is}} \in \boldsymbol{VA_i})) \wedge (\nexists\, \boldsymbol{cp_{xy}} | ((\boldsymbol{cp_{xy}} \in$
    $\boldsymbol{CP_C}) \wedge$
                $(\boldsymbol{cp_{xy}.cd.status} = \boldsymbol{Invalid}))) \wedge (\boldsymbol{CP_{UC}} \coloneqq (\boldsymbol{CP_{UC}} \backslash \boldsymbol{cp_{ij}})) \wedge (\boldsymbol{CP_C} \coloneqq (\boldsymbol{CP_C} \cup \boldsymbol{cp_{ij}})) \wedge$
                $(\boldsymbol{cp_{ij}.cd.isAutoGenerated} \coloneqq \boldsymbol{True}) \wedge (\boldsymbol{CD} \coloneqq (\boldsymbol{CD} \cup \boldsymbol{cp_{ij}.cd})) \wedge$
6.              $(\boldsymbol{IF}\colon \boldsymbol{cp_{ij}.type} \in \{\boldsymbol{ComponentCardinality - VP, ComponentCollectionBoundary -}$
    $\boldsymbol{VP, ComponentChoice - VP}$
                $\boldsymbol{, ComponentSelection - VP, TopologyChoice - VP, AllocationChoice - VP, InteractionChoice - VP}\}$
                $\boldsymbol{THEN}\colon (\boldsymbol{CP_{UC}} = (\boldsymbol{CP_{UC}} \cup \boldsymbol{CP_{UC-Additonal}}))))$

    where $\boldsymbol{CP_{UC-Additonal}}$ is a set of additional un-configured ConfigurableParameters added due to the configuration of $\boldsymbol{cp_{ij}}$.
```

Listing 16: Formal definition of DecisionInference

1.2 DecisionOrdering

Inputs: Sets of un-configured *ConfigurableParameter*s CP_{UC} and *DecisionOrderingConstraint*s C_{DO}.
Outputs: A sequence of un-configured *ConfigurableParameter*s CP_{UC} in which they should be configured.

```
1.  $\boldsymbol{DO(CP_{UC}, C_{DO}) \overset{\text{def}}{=} Sequence(CP_{UC})} | (\nexists \boldsymbol{cp_{ik}, cp_{jl}} | ((\boldsymbol{cp_{ik}, cp_{jl}} \in \boldsymbol{Sequence(CP_{UC})}) \wedge$
    $(\exists\, \boldsymbol{c} \in \boldsymbol{C_{DO}} | ((\boldsymbol{c.evluationResult} = \boldsymbol{False}) \wedge$
    $(\boldsymbol{vp_i} \in \boldsymbol{VAR(c)}) \wedge (\boldsymbol{vp_j} \in \boldsymbol{VAR(c)})))))$

Where $\mathbf{Sequence(CP_{UC})}$ returns a sequence of elements in $\boldsymbol{CP_{UC}}$.
```

Listing 17: Formal definition of DecisionOrdering

1.3 RevertingDecision

Inputs: A *ConfigurableParameter* cp_{ik} to be reverted and sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, *ConfigurationData* CD, and *DecisionInferenceConstraint*s C_{DI}.
Outputs: Updated sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, and *ConfigurationData* CD.

```
1. $RD(cp_{ik}, CP_C, CP_{UC}, CD, C_{DI}) \overset{\text{def}}{=} CP_C, CP_{UC}, CD$
2. $\forall\, c \in C_{DI},$
3.     $(VP_C \coloneqq VAR(c)) \wedge (VA_C \coloneqq VAL(c)) \mid vp_i \in VAR(c)$
4.     $\forall\, vp_i \in VP_C, \forall\, cp_{jl} \in CP_C \mid ((cp_{jl}.cd.isAutoGenerated = True) \wedge (vp_j \in VP_C))$
5.             $RD(cp_{jl}, CP_C, CP_{UC}, CD, C_{DI})$, Recursively reverting ConfigurationDecisions.
6.     $IF\colon cp_{ik}.type \in \left\{ \begin{matrix} ComponentCardinality - VP, ComponentCollectionBoundary - VP, ComponentChoice - VP, \\ ComponentSelection - VP, TopologyChoice - VP, AllocationChoice - VP, InteractionChoice - VP \end{matrix} \right\}$
       $THEN\colon \forall\, cp_{xy} \in CP_{Additional},$
7.             $IF\colon (cp_{xy}.status = Configured)\ THEN\colon CD \coloneqq (CD \setminus cp_{xy}.cd) \wedge (CP_C \coloneqq (CP_C \setminus cp_{xy}))$
8.             $IF\colon (cp_{xy}.status = Unconfigured)\ THEN\colon CP_{UC} \coloneqq (CP_{UC} \setminus cp_{xy})$
9.     $(CD \coloneqq (CD \setminus cp_{ik}.cd)) \wedge (cp_{ik}.status \coloneqq Unconfigured) \wedge (cp_{ik}.selectedVariant \coloneqq null) \wedge (CP_C \coloneqq$
   $(CP_C \setminus cp_{ik})) \wedge$
       $(CP_{UC} \coloneqq (CP_{UC} \cup cp_{ik}))$
   Where $CP_{Additional}$ is a set of additionally added *ConfigurableParameter*s due to the configuration of $cp_{ik}$
```

Listing 18: Formal definition of RevertingDecision

1.4 WellFormednessChecking

Inputs: A set of *ModelElements* ME (e.g., *ConfigurableParameter*s) corresponding to a *ResolutionModel* and a set of *WellFormednessConstraint*s C_{WF}.
Outputs: A set of ill-formed *ModelElements* ME_{IF}.

```
1. $WFC\ (ME, C_{WF}) \overset{\text{def}}{=} \begin{cases} \emptyset, |ME_{IF}| = 0 \\ ME_{IF}, |ME_{IF}| > 0 \end{cases}$
2. $ME_{IF} = \{\bigcup(\forall me_{ij} \in ME, (me_{ij} \mid (\exists\, c_k \mid ((c_k \in C_{WF}) \wedge (c_k.evaluationResult = False) \wedge (mme_i \in VAR(c_k))))))\}$
```

Listing 19: Formal definition of WellFormednessChecking

1.5 ConformanceChecking

Inputs: A set of configured *ConfigurableParameter*s CP_C and a set of *ConformanceConstraint*s C_{CF}.
Outputs: A set of *ConfigurableParameter*s CP_V violating one or more *ConformanceConstraint*s

```
1. $CFC\ (CP_C, C_{CF}) \overset{\text{def}}{=} \begin{cases} \emptyset, |CP_V| = 0 \\ CP_V, |CP_V| > 0 \end{cases}$
2. $CP_V = \{\bigcup(\forall cp_{ij} \in CP_C, (cp_{ij} \mid (\exists\, c_k \mid ((c_k \in C_{CF}) \wedge (c_k.evaluationResult = False) \wedge (vp_i \in VAR(c_k))))))\}$
```

Listing 20: Formal definition of ConformanceChecking

1.6 ConsistencyChecking

Inputs: A set of configured *ConfigurableParameter*s CP_C and a set of *ConsistencyConstraint*s C_{CS}.
Outputs: A set C_{IC} containing sets of inconsistent *ConfigurableParameter*s (i.e., violating one or more *ConsistencyConstraint*s).

```
1. $CSC\ (CP_C, C_{CS}) \overset{\text{def}}{=} \begin{cases} \emptyset, |CP_{IC}| = 0 \\ CP_{IC}, |CP_{IC}| > 0 \end{cases}$
2. $CP_{IC} = \{\bigcup(\forall cp_{ij} \in CP_C, (cp_{ij} \mid (\exists\, c_k \mid ((c_k \in C_{CS}) \wedge (c_k.evaluationResult = False) \wedge (vp_i \in VAR(c_k))))))\}$
```

Listing 21: Formal definition of ConsistencyChecking

1.7 ResolvingViolation

Inputs: A set of *Violation*s $VL = \{vl_1, vl_2, .., vl_{ni}\}$ identified where $\forall\, vl_j \in VL, vl_j.violationType \in \{WellFormednessConstraint, ConformanceConstraint, ConsistencyConstraint\}$ and sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, and *ConfigurationData* CD.

Outputs: A set of unresolved *Violation*s VL and updated sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, and *ConfigurationData* CD.

```
1.  $RV\ (VL, CP_C, CD, CP_{UC}) \overset{\text{def}}{=} \begin{cases} \emptyset, |VL| = 0 \\ VL, |VL| > 0 \end{cases}$
2.  $\forall\ vl \in VL,$
3.      $IF: vl.violationType = WellFormednessConstraint$
4.      $THEN: \forall\ cp_{ij} \in vl.causingParameters, ((CP_{UC} \coloneqq (CP_{UC} \backslash cp_{ij})) \wedge (cp_{ik} \coloneqq Instance(vp_i)) \wedge\ (CP_{UC} \coloneqq (CP_{UC} \cup cp_{ik})) \wedge$
        $(vp_i \in VAR(vl.violatedConstraints[0])))$
5.      $IF: vl.violationType = ConformanceConstraint$
6.      $THEN: \forall\ cp_{ij} \in vl.causingParameters, ((CD \coloneqq (CD \backslash cp_{ij}.cd)) \wedge (cp_{ij}.selectedVariant \coloneqq Replace(v_{is}, v_{ik}) | ((v_{is} \neq$
    $v_{ik}) \wedge$
        $(v_{ik} \in VA_i) \wedge (v_{ik} \in VAL(vl.violatedConstraints[0])))) \wedge \left(CD \coloneqq (CD \cup cp_{ij}.cd)\right) \wedge$
        $(vp_i \in VAR(vl.violatedConstraints[0])))$
7.      $IF: vl.violationType = ConsistencyConstraint$
8.      $THEN: \forall\ cp_{ij}, cp_{kl} \in vl.causingParameters, (((CD \coloneqq (CD \backslash cp_{ij}.cd)) \wedge (cp_{ij}.selectedVariant \coloneqq Replace(v_{is}, v_{ik})|$
        $((v_{is} \neq v_{ik}) \wedge (v_{ik} \in VA_i) \wedge (v_{ik} \in VAL(vl.violatedConstraints[0])))) \wedge \left(CD \coloneqq (CD \cup cp_{ij}.cd)\right) \wedge$
        $(vp_i \in VAR(vl.violatedConstraints[0]))) \vee ((CD \coloneqq (CD \backslash cp_{kl}.cd)) \wedge (cp_{kl}.selectedVariant \coloneqq Replace(v_{ks}, v_{km})|$
        $\left((v_{ks} \neq v_{km}) \wedge (v_{km} \in VA_k) \wedge \left(v_{km} \in VAL(vl.violatedConstraints[0])\right)\right)) \wedge \left(CD \coloneqq (CD \cup cp_{kl}.cd)\right) \wedge$
        $(vp_k \in VAR(vl.violatedConstraints[0]))))$
9.      $IF: vl.violatedConstraints[0].evaluate() = True$
10.     $THEN: VL \coloneqq (VL \backslash vl)$
```

Listing 22: Formal definition of ResolvingViolation

1.8 CollaborativeConfiguration

Inputs: A set of *VariationPoint*s VP and sets of eight types of constraints (i.e., $C_{DO}, C_{WF}, C_C, C_{VD}, C_{DI}, C_{OP}, C_{CF}, C_{CS}$).
Outputs: A valid configured *MemberProduct* P containing a set of *ConfigurableParameter*s CP and *ConfigurationData* CD.

```
1.  $CC(VP, C_{DO}, C_{WF}, C_C, C_{VD}, C_{DI}, C_{OP}, C_{CF}, C_{CS}) \overset{\text{def}}{=} (P |\ P$ is a configured MemberProduct$)$
2.  $Split(VP) \overset{\text{def}}{=} VP_S |\ ((VP_S = \{VP_{s1}, VP_{s2}, .., VP_{ns}\}) \wedge (ns \geq 2) \wedge (\forall\ VP_{si} \in VP_S, VP_{si} \subseteq VP) \wedge\ (|\{VP_{s1} \cup VP_{s2} \cup .. \cup VP_{ns}\}| = |VP|) \wedge$
    $(\forall\ VP_{si}, VP_{sj}\ \in VP_S, \left((VP_{si} \cap\ VP_{sj}) = \emptyset\right)) \wedge (\forall\ vp_i, vp_j \in VP_{si}, vp_i.configurationStage = vp_j.configurationStage) \wedge$
    $(\nexists\ vp_x \in VP_{si},\ vp_y \in VP_{sj} | vp_x.configurationStage = vp_y.configurationStage))$
3.  $Configure(VP_S, C_{DO}, C_{WF}, C_C, C_{VD}, C_{DI}, C_{OP}, C_{CF}, C_{CS}) \overset{\text{def}}{=} CP_S, CD_S |\ ((CP_S = \{CP_{s1}, CP_{s2}, .., CP_{ns}\})\ \wedge (CD_S =$
    $\{CD_{s1}, CD_{s2}, .., CD_{ns}\})),$
```

where CP_S is a set containing sets of *ConfigurableParameter*s (configured or/and un-configured) for each stage and CD_S is a set containing sets of *ConfigurationData* (for configured *ConfigurableParameter*s) for each stage. Note that $Configure(VP_S, C_{DO}, C_{WF}, C_C, C_{VD}, C_{DI}, C_{OP}, C_{CF}, C_{CS})$ uses all the other functionalities (e.g., *DecisionInference*, *DecisionOrdering*, *ConformanceChecking*) to configure different *VariationPoint*s during different stages.

```
4.  $Merge(CP_S, CD_S, C_{CS}) \overset{\text{def}}{=} CP, CD |\ ((CP = \{CP_{s1} \cup CP_{s2} \cup .. \cup CP_{ns}\})\ \wedge (CD = \{CD_{s1} \cup CD_{s2} \cup .. \cup CD_{ns}\})\ \wedge (\nexists c \in$
    $C_{CS} | c.evaluationResult = False))$
```

where CP and CD are *ConfigurableParameter*s and *ConfigurationData* corresponding to *MemberProduct* P. Note that $Merge(CP_S, CD_S, C_{CS})$ maybe use *ConsistencyChecking* to ensure the consistency across the *ConfigurationDecision*s made during different *ConfigurationStage*s.

Listing 23: Formal definition of CollaborativeConfiguration

1.9 ImpactAnalysis

Inputs: A *sourceChange* ch_s for which *Impact* I needs to be assessed and sets of configured *ConfigurableParameter*s CP_C, un-configured *ConfigurableParameter*s CP_{UC}, *VariationPoint*s VP, and *VariabilityDependencyConstraint*s C_{VD}.
Outputs: *Impact* I with a set of target *Change*s CH (i.e., *Impact*).

```
1.  $IA\ (ch_s, VP, CP_C, CP_{UC}, C_{VD}) \overset{\text{def}}{=} I | I.targetChanges =\ \{ch_1, ch_2, .., ch_{nch}\}$
2.  $\forall ch_i \in I.targetChanges$
3.      $IF: (ch_s.model = ResolutionModel)\ THEN:$
```

```
4.          IF: $(ch_s.type = Remove)$ THEN: $((ch_i.type = Remove \vee ch_i.type = Update) \wedge$
            $(ch_i.model = ResolutionModel) \wedge (ch_i \rightarrow CP_C \coloneqq Change(CP_C, C_{VD}))\ \wedge (ch_i \rightarrow CP_{UC} \coloneqq Change(CP_{UC}, C_{VD}))$
5.          IF: $(ch_s.type = Add\ \vee ch_s.type = Update)$ THEN: $((ch_i.type = Add \vee ch_i.type = Remove \vee$
            $ch_i.type = Update) \wedge\ (ch_i.model = ResolutionModel) \wedge (ch_i \rightarrow CP_C \coloneqq Change(CP_C, C_{VD}))\ \wedge$
            $(ch_i \rightarrow CP_{UC} \coloneqq Change(CP_{UC}, C_{VD})))$
6.          IF: $(ch_s.type = Move)$ THEN: $(\ (ch_i.type = Move \vee ch_i.type = Remove \vee ch_i.type = Update) \wedge$
            $(ch_i.model = ResolutionModel) \wedge (ch_i \rightarrow CP_C \coloneqq Change(CP_C, C_{VD}))\ \wedge (\ ch_i \rightarrow CP_{UC} \coloneqq$
    $Change(CP_{UC}, C_{VD})))$
7.      IF: $(ch_s.model = VariabilityModel)$ THEN:
8.          IF: $(ch_s.type = Remove)$ THEN: $((ch_i.type = Remove \vee ch_i.type = Update) \wedge (ch_i.model =$
    $VariabilityModel \wedge$
            $ch_i.model = ResolutionModel) \wedge (ch_i \rightarrow CP_C \coloneqq Change(CP_C, C_{VD})) \wedge (\ ch_i \rightarrow CP_{UC} \coloneqq$
    $Change(CP_{UC}, C_{VD})) \wedge$
            $(ch_i \rightarrow VP \coloneqq Change(VP, C_{VD})))$
9.          IF: $((ch_s.type = Add\ \vee ch_s.type = Update)$ THEN: $((ch_i.model = VariabilityModel \wedge$
            $ch_i.model = ResolutionModel) \wedge (ch_i \rightarrow CP_C \coloneqq Change(CP_C, C_{VD})) \wedge\ (ch_i \rightarrow CP_{UC} \coloneqq$
    $Change(CP_{UC}, C_{VD})) \wedge$
            $(\ ch_i \rightarrow VP \coloneqq Change(VP, C_{VD})))$
10.         IF: $(ch_s.type = Move)$ THEN: $((ch_i.type = Move \vee ch_i.type = Remove \vee ch_i.type = Update) \wedge$
            $(ch_i.model = VariabilityModel \wedge ch_i.model = ResolutionModel) \wedge (ch_i \rightarrow CP_C \coloneqq Change(CP_C, C_{VD}))\ \wedge$
            $(ch_i \rightarrow CP_{UC} \coloneqq Change(CP_{UC}, C_{VD})) \wedge (\ ch_i \rightarrow VP \coloneqq Change(VP, C_{VD})))$
```

Listing 24: Formal definition of ImpactAnalysis

1.10 ConflictDetection

Inputs: A set of hard constraints C of any type defined in Section 4.3 except *OptimizationConstraint*s corresponding to a *ProductLine* for which conflicting constraints are to be detected.
Outputs: A set of pairs of conflicting constraints S.

```
1.  $CD\ (C) \overset{\text{def}}{=} \begin{cases} \emptyset, |S| = 0 \\ S, |S| > 0 \end{cases}$
2.  IF: $C$ is a set of *DecisionOrderingConstraint*s
3.  THEN: $S = \left\{\bigcup_{x=1,y=2}^{nc} \left(Pair(c_x, c_y)\right| (\exists\ cp_{ik}, cp_{jl} \left| \left(((c_x \wedge c_y) = False\right) \wedge (vp_i \in VAR(c_x)\right) \wedge \left(vp_j \in VAR(c_y)\right) \wedge$
    $\left(c_x \rightarrow\ cp_{ik}.configurationStep < cp_{jl}.configurationStep\right) \wedge \left(c_y \rightarrow\ cp_{jl}.configurationStep <\right.$
    $\left.cp_{ik}.configurationStep\right)))\}$
4.  IF: $C$ *is a set of WellFormednessConstraints*
5.  THEN: $S = \left\{\bigcup_{x=1,y=2}^{nc} \left(Pair(c_x, c_y)\right| (\exists\ cp_{ij} \left| \left(((c_x \wedge c_y) = False\right) \wedge (vp_i \in VAR(c_x)\right) \wedge \left(vp_i \in VAR(c_y)\right)))\}$
6.  IF: $C$ *is a set of ConfigurationConstraints, ConformanceConstraints, or ConsistencyConstraints*
7.  THEN: $S = \left\{\bigcup_{x=1,y=2}^{nc} \left(Pair(c_x, c_y)\right| (\nexists\ cp_{ik}, cp_{jl} \left|((cp_{ik}.status = Configured) \wedge (cp_{jl}.status = Configured) \wedge\right.$
    $(cp_{ik}.cd.status = Valid) \wedge (cp_{jl}.cd.status = Valid) \wedge (c_x.isHardConstraint = True) \wedge (c_y.isHardConstraint =$
    $True) \wedge \left((c_x \wedge c_y) = True\right) \wedge \left(vp_i \in VAR(c_x)\right) \wedge \left(vp_j \in VAR(c_y)\right)))\}$
8.  IF: $C$ *is a set of VariabilityDependencyConstraints*
9.  THEN: $S = \{\bigcup_{x=1,y=2}^{nc} \left(Pair(c_x, c_y)\right| (\nexists\ cp_{ik}, cp_{jl} \left|((c_x.type = c_y.type) \wedge (c_x.relation \neq c_y.relation) \wedge (VAR(c_x) =\right.$
    $VAR(c_y)) \wedge (vp_i \in VAR(c_x)) \wedge \left(vp_j \in VAR(c_y)\right))) \vee (\nexists\ cp_{ik}, cp_{jl} \left|((c_x.type = VP - VP) \wedge (c_y.type = VP - VP) \wedge\right.$
    $(c_x.relation = c_y.relation) \wedge ((c_i \wedge c_j) = True) \wedge (vp_i \in VAR(c_x)) \wedge \left(vp_j \in VAR(c_y)\right))) \vee (\nexists\ cp_{ik}, cp_{jl}, v_{js} \left|((c_x.type =\right.$
    $VP - VA) \wedge (c_y.type = VP - VA) \wedge (c_x.relation = c_y.relation) \wedge ((c_i \wedge c_j) = True) \wedge (vp_i \in VAR(c_x)) \wedge (vp_j \in$
    $VAR(c_y)) \wedge \left(v_{js} \in VAL(c_y)\right))) \vee (\nexists\ cp_{ik}, cp_{jl}, v_{is}, v_{js} \left|((c_x.type = VA - VA) \wedge (c_y.type = VA - VA) \wedge (c_x.relation =\right.$
    $c_y.relation) \wedge ((c_i \wedge c_j) = True) \wedge (vp_i \in VAR(c_x)) \wedge \left(vp_j \in VAR(c_y)\right) \wedge (v_{is} \in VAL(c_x)) \wedge \left(v_{js} \in VAL(c_y)\right))))\}$
```

Listing 25: Formal definition of ConflictDetection

1.11 ConstraintSelection

Inputs: A set of *VariationPoint*s VP, a set of *ConfigurableParameters* CP, and a set of constraints $C = \{c_1, c_2, .., c_{nc}\}$ of a particular type (e.g., *DecisionInferenceConstraint*s) corresponding to which a subset of constraints needs to be selected.
Outputs: A subset of selected constraints.

$$\mathbf{CS}\ (\boldsymbol{VP}, \mathbf{CP}, \mathbf{C}) \overset{\text{def}}{=} \forall\ \boldsymbol{cp_{ij}} \in \boldsymbol{CP}, (\{\cup_{k=1}^{nc} \boldsymbol{c_k} \mid ((\boldsymbol{c_k} \in \boldsymbol{C}) \wedge (\boldsymbol{vp_i} \in \mathbf{VP}) \wedge ((\ \boldsymbol{vp_i} \in \boldsymbol{VAR}(\boldsymbol{c_k})) \vee (\boldsymbol{cp_{ij}}.\mathbf{selectedVariant} \in \boldsymbol{VAL}(\boldsymbol{c_k}))))\})$$

Listing 26: Formal definition of ConstraintSelection

1.12 ConfigurationOptimization

Inputs: A set of *ConfigurableParameter*s CP_C and a set of *OptimizationConstraint*s C_{OP}.
Outputs: Updated set of *ConfigurableParameter*s CP_C with optimal *Variants* assigned.

$$\begin{aligned} \boldsymbol{CO}(\boldsymbol{CP_C}, \boldsymbol{C_{OP}}) \overset{\text{def}}{=} \forall \boldsymbol{cp_{ij}} \in \boldsymbol{CP_C}, (\boldsymbol{cp_{ij}}.\boldsymbol{selectedVariant} := \boldsymbol{v_{is}} \mid (\ \boldsymbol{v_{is}} \in \boldsymbol{VA_i}\ ) \wedge (\forall\ \boldsymbol{v_{il}} \\ \in \boldsymbol{VA_i}, |(\bigcup_{k=1}^{nc} ((\boldsymbol{c_k}.\boldsymbol{evulationResult} = \boldsymbol{True}) \wedge (\boldsymbol{cp_{ij}}.\boldsymbol{selectedVariant} = \boldsymbol{v_{is}}) \wedge (\boldsymbol{vp_i} \in \boldsymbol{VAR}(\boldsymbol{c_k}))))| \\ \geq |(\bigcup_{k=1}^{nc} ((\boldsymbol{c_k}.\boldsymbol{evulationResult} = \boldsymbol{True}) \wedge (\boldsymbol{cp_{ij}}.\boldsymbol{selectedVariant} = \boldsymbol{v_{il}}) \wedge (\boldsymbol{vp_i} \in \boldsymbol{VAR}(\boldsymbol{c_k}))))|)) \end{aligned}$$

Listing 27: Formal definition of ConfigurationOptimization

1.13 RedundancyDetection

Inputs: Two *ConfigurationFile*s F_1 and F_2 (or only one) representing a *ResolutionModel* corresponding to a *MemberProduct* P containing a set of n *ConfigurableParameter*s $CP = (CP_1 \cup CP_2)$, where $CP_1 = \{cp_1, cp_2, .., cp_m\}$ and $CP_2 = \{cp_{m+1}, cp_{m+2}, .., cp_n\}$ are *ConfigurableParameter*s corresponding to F_1 and F_2 respectively. $CD_1 = \{cp_1.cd, cp_2.cd, .., cp_m.cd\}$ *and* $CD_2 = \{cp_{m+1}.cd, cp_{m+2}.cd, .., cp_n.cd\}$ are two sets of *ConfigurationData* corresponding to F_1 and F_2 respectively.
Outputs: A set of duplicate *ConfigurationData*.

```
1.  $\boldsymbol{IF}: (\boldsymbol{scope} = \boldsymbol{IntraConfigurationFile})$
2.  $\boldsymbol{THEN}: \boldsymbol{RD}\ (\boldsymbol{F_1}) \overset{\text{def}}{=} \cup_{i=1,j=2}^{n} (\boldsymbol{cp_i}.\boldsymbol{cd}, \boldsymbol{cp_j}.\boldsymbol{cd} \mid ((\boldsymbol{cp_i}.\boldsymbol{cd}, \boldsymbol{cp_j}.\boldsymbol{cd} \in \boldsymbol{CD_1}) \wedge (\boldsymbol{cp_i}.\boldsymbol{cd}.\boldsymbol{parameterID} = \boldsymbol{cp_j}.\boldsymbol{cd}.\boldsymbol{parameterID})))$
3.  $\boldsymbol{IF}: (\boldsymbol{scope} = \boldsymbol{InterConfigurationFiles})$
4.  $\boldsymbol{THEN}: \boldsymbol{RD}\ (\boldsymbol{F_1}, \boldsymbol{F_2}) \overset{\text{def}}{=} \cup_{i=1,j=1}^{i=n,j=m} (\boldsymbol{cp_i}.\boldsymbol{cd}, \boldsymbol{cp_j}.\boldsymbol{cd} \mid ((\boldsymbol{cp_i}.\boldsymbol{cd} \in \boldsymbol{CD_1}) \wedge (\boldsymbol{cp_j}.\boldsymbol{cd} \in \boldsymbol{CD_2}) \wedge (\boldsymbol{cp_i}.\boldsymbol{cd}.\boldsymbol{parameterID} =$
    $\boldsymbol{cp_j}.\boldsymbol{cd}.\boldsymbol{parameterID})))$
```

Listing 28: Formal definition of RedundancyDetection

Note that if a *ResolutionModel* is represented in more than two *ConfigurationFile*s, then *RedundancyDetection* can be applied multiple times on each pair of *ConfigurationFile*s.

1.14 IncompletenessDetection

Inputs: A *ConfigurationFile* F representing a *ResolutionModel* corresponding to a *MemberProduct* P containing a set of n *ConfigurableParameter*s $CP = \{cp_1, cp_2, .., cp_n\}$ and *ConfigurationData* $CD = \{cp_1.cd, cp_2.cd, .., cp_n.cd\}$ *representing the ConfigurationDecision*s made to configure n *ConfigurableParameter*s.
Outputs: A set of un-configured *ConfigurableParameters*.

$$\boldsymbol{ID}\ (\boldsymbol{F}) \overset{\text{def}}{=} \bigcup_{i=1}^{n} \boldsymbol{cp_i} \mid ((\boldsymbol{cp_i} \in \boldsymbol{CP}) \wedge (\boldsymbol{cp_i}.\boldsymbol{cd} \in \boldsymbol{CD}) \wedge ((\boldsymbol{cp_i}.\boldsymbol{cd}.\boldsymbol{value} = \boldsymbol{null}) \vee (\boldsymbol{cp_i}.\boldsymbol{selectedVariant} = \boldsymbol{null})))$$

Listing 29: Formal definition of IncompletenessDetection

Paper C

Mining Cross Product Line Rules with Multi-Objective Search and Machine Learning

C

Safdar Aqeel Safdar, Hong Lu, Tao Yue, Shaukat Ali

Published in the Proceedings of The Genetic and Evolutionary Computation Conference (GECCO), 2017.

Abstract

Product Line Engineering (PLE) is a well-acknowledged paradigm to improve the productivity of developing products with higher quality and at a lower cost. By benefiting from PLE, more and more systems are developed by integrating products, which belong to different product lines, and communicate and interact with each other through information networks [1, 2]. Examples of such systems include video conferencing systems (VCSs) [3] and material handling

Nowadays, an increasing number of systems are being developed by integrating products (belonging to different product lines) that communicate with each other through information networks. Cost-effectively supporting Product Line Engineering (PLE) and in particular enabling automation of configuration in PLE is a challenge. Capturing rules is the key for enabling automation of configuration. Product configuration has a direct impact on runtime interactions of communicating products. Such products might be within or across product lines and there usually don't exist explicitly specified rules constraining configurable parameter values of such products. Manually specifying such rules is tedious, time-consuming, and requires expert's knowledge of the domain and the product lines. To address this challenge, we propose an approach named as SBRM that combines multi-objective search with machine learning to mine rules. To evaluate the proposed approach, we performed a real case study of two communicating Video Conferencing Systems belonging to two different product lines. Results show that SBRM performed significantly better than Random Search in terms of fitness values, Hyper-Volume, and machine learning quality measurements. When comparing with rules mined with real data, SBRM performed significantly better in terms of *Failed Precision* (18%), *Failed Recall* (72%), and *Failed F-measure* (59%).

Keywords: Product Line; Configuration; Rule Mining; Multi-Objective Search; Machine Learning; Interacting Products

1. Introduction

Product Line Engineering (PLE) is a well-acknowledged paradigm to improve the productivity of developing products with higher quality and at a lower cost. By benefiting from PLE, more and more systems are developed by integrating products, which belong to different product lines, and communicate and interact with each other through information networks [1, 2]. Examples of such systems include video conferencing systems (VCSs) [3] and material handling systems [4]. Such systems are highly configurable by presenting the users with configuration options. Consequently, at runtime, several products belonging to multiple product lines communicate (e.g., via information networks) with each other [1, 2] under various configurations. Thus, the runtime behavior of such systems not only depends on the configuration of these communicating products but is also influenced by the communication medium. Note that the configuration in our context indicates numerous configurable parameters exposed to users after the system is deployed.

Cost-effective PLE is challenging mainly because of the lack support of automation of the configuration process [5, 6]. Capturing rules is the key to enabling automation of various configuration functionalities (e.g., consistency checking, decision propagation, and decision ordering) [7-11]. In our context, such rules describe how configurations of communicating products belonging to different product lines influence their runtime interactions via information networks.

We name rules constraining configurations (values assigned to configurable parameters) of products belonging to different product lines as Cross Product Lines (CPL) rules. CPL rules are of significant importance for mainly two reasons. First, CPL rules can be used to identify invalid configurations where products may fail to interact with a confidence level due to, e.g., dependencies on external libraries and/or platforms. Identified invalid configurations can help developers to maintain current products or evolve future products. Second, CPL rules can provide support to enable (automated or semi-automated) configuration of products of future deployments. However, the literature does not provide sufficient support to mine such rules, as current practice mainly focuses on mining rules constraining product configurations within a single product line [6, 12].

CPL rules need to be captured by running the system due to the information only known at runtime, e.g., dependencies on external libraries and/or platforms. As mentioned in [13], rules that ensure correct runtime behaviors can be identified from either domain knowledge or testing of the system. Manually specifying such rules based on domain knowledge is tedious and time-consuming, and heavily relies on expert's knowledge of the domain and the product lines. Identifying CPL rules via testing has its own challenges, as the configuration space is typically very large and testing candidate configurations is often infeasible. Besides, in practice testers often use valid configurations to test the system [13]. Therefore, identifying CPL rules requires a dedicated approach that automatically obtains rules without exploring all possible configurations of the communicating products belonging to different product lines.

In [12], a rule mining approach is proposed that mines rules for a product line where product configurations belonging to one product line are generated randomly and labeled as faulty and non-faulty. Labeled product configurations are inputted to the classification algorithm of j48 [14] to mine rules. However, randomly generating configurations to mine rules is a brute-force way and time-consuming. In this work, we advance one step further by employing search to generate product configurations intelligently using three heuristics (Section 3.2), instead of randomly generating product configurations.

We propose an approach, named as Search-based Rule Mining (SBRM), which combines multi-objective search with machine-learning techniques, to mine CPL rules in an incremental and iterative way. SBRM obtains CPL rules with different degrees of confidence (i.e., the probability of being correct) with an emphasis on mining rules that can reveal invalid configurations [15]. Instead of collecting a large amount of data required for machine learning all in once, we obtain the input data incrementally with multiple iterations. During each iteration, we use the rules mined from the previous iteration to guide the search for generating configuration data for the current iteration. The generated configuration data are combined together with those from all the previous iterations in order to incrementally refine the aforementioned rules. SBRM is validated using a real world case study of VCSs, where two products belonging to different product lines communicate (i.e., call) with each other.

We summarize the key contributions of the paper below:

- SBRM to mine CPL rules constraining configurations of communicating products across product lines.

- Three objectives to guide the search for generating configuration data in order to refine CPL rules.
- Evaluating SBRM by performing a real-world case study of two communicating VCSs belonging to different product lines. With the case study, we compared the performance of NSGA-II with Random Search (RS) using fitness values, Hyper-Volume (HV), and machine learning quality measurements. Additionally, we compared the rules mined using SBRM with the rules mined with real data extracted from test case execution logs.

Evaluation results show that SBRM is effective to produce high-quality rules as compared to RS based rule mining approach (i.e., called RBRM). Results also indicate that SBRM produces better rules as compared to the rules mined based on real data extracted from test case execution logs.

The rest of the paper is organized as follows: In Section 2, we give an overview of SBRM followed by the search-based approach for generating configuration data in Section 3. In Section 4, we present the experiment design, execution, and results. Section 5 summarizes the literature review and finally, in Section 6, we conclude the work.

2. Overview

Figure C-1 presents an overview of our proposed approach (SBRM), which relies on machine learning and multi-objective search to mine CPL rules. At the first step, an initial set of configuration data is generated randomly for the selected products belonging to different product lines. At the second step, selected products are configured with the randomly generated configuration data, and certain functionalities of the products are executed such that the selected products interact with each other via information networks (e.g., the Internet), and the states of the system are captured to know if they interact via communication network successfully. An interaction, in our context, can be defined as an action in which two or more objects (e.g., system, product, or component) are collaborating, communicating, or influencing each other. There does not exist a generic way of enabling interactions among various products of a system via communication networks as well as capturing the system states as it depends on the application domain of the system under study and its involved functionalities.

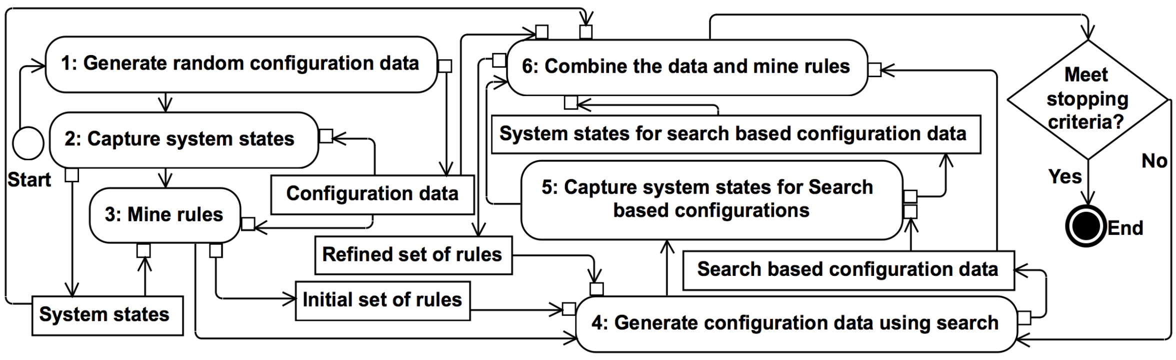

1: Generate random configuration data
2: Capture system states
3: Mine rules
Start
System states
Configuration data
Refined set of rules
Initial set of rules
6: Combine the data and mine rules
System states for search based configuration data
5: Capture system states for Search based configurations
Search based configuration data
4: Generate configuration data using search
Meet stopping criteria?
Yes
No
End

Figure C-1. Overview of the proposed approach (SBRM)

In step 3, we feed the set of generated configuration data (as Attributes) and their corresponding system states (as Classes) to Weka [14] as the initial input and apply the Pruning Rule-Based Classification algorithm (PART) [15] to mine the initial set of rules, which are consequently fed to NSGA-II for generating configuration data for the next iteration in step 4. Though C4.5 and RIPER are the two well-known algorithms, which generate rules based on decision trees [14, 15], C4.5 is expensive in terms of computation time since the process of generating/pruning rules is complex and requires global optimization. In the case of RIPPER, it suffers from over-pruning (hasty generalization) problem [16]. PART [15] combines these two paradigms while avoiding their shortcomings by generating partially pruned decision trees and inducing one rule corresponding the longest branch of each partial tree. In step 5, we repeat step 2 but take the configuration data generated from the search instead of the random one. In step 6, we combine all the configuration data generated from steps 1 and 4 and collected system states captured from steps 2 and 5, and feed all the data to Weka to mine a refined set of rules. This rule set is then used in the next iteration (starting from step 4) to generate more configuration data and further refine the rules.

In each iteration, newly generated configuration data with collected system states are added to the dataset from the previous iteration to mine a new set of rules. We repeat the process until we meet the stopping criteria, e.g., a fixed number of iterations and/or when the rules generated from two consecutive iterations are similar. Fixed number of iterations is useful when we have limited available resources for mining rules. Getting similar rules from consecutive iterations indicates that it is very unlikely to refine the rules further. We consider step 4, i.e., using search to generate configuration data, as the innovative part of the whole approach, i.e., SBRM. This is because using Weka to mine rules is a simple application of the PART algorithm and applying search requires carefully designing a fitness function. Therefore, in Section 3, we present how search is used for generating configuration data (step 4) and the evaluation of SBRM is presented in Section 4.

3. Search-Based Approach

Sections 3.1 presents definitions required to define the configuration data generation problem. Section 3.2 presents the objectives and measures, followed by the fitness function defined in Section 3.3.

3.1 Definition and Problem Representation

$\mathrm{CP} = \{\mathrm{cp}_1, \mathrm{cp}_2, .., \mathrm{cp}_{\mathrm{ncp}}\}$ represents a set of configuration parameters with the total number being ncp. For each cp_i cp_i in CPV_i CPV_i represents a set of possible values: $ncpv$ is the total number of unique values (i.e., configuration space) for all the configuration parameters, which can be calculated as: $\mathrm{ncpv} = \left|\left(\bigcup_{\mathrm{i}=1}^{\mathrm{ncp}} \mathrm{CPV}_\mathrm{i}\right)\right|$. Figure C-2 shows four sanitized configuration parameters (cp1-cp4) from our case study. For example, cp1 represents the protocol (e.g., related to video conference over IP networks) of product P1, which can be configured with four different values (e.g., Pro-1).

cp1: P1_Protocol {Pro-1, Pro-2, Pro-3, Pro-4}
cp2: P1_Encryption {Enc-1, Enc-2, Enc-3}
cp3: P2_Encryption {Enc-1, Enc-2, Enc-3}
cp4: P2_ListenPort {LP-1, LP-2}

r1: P1_Protocol = Pro-2 AND P2_ListenPort = LP-2: Failed
r2: P1_Protocol = Pro-3 AND P2_Encryption = Enc-3: Connected
r3: P1_Encryption = Enc-1 AND P2_Encryption = Enc-2: Failed

Figure C-2. Examples of sanitized configuration parameters and CPL rules

$R_N = \{r_{n1}, r_{n2}, r_{n3}, \dots, r_{nnr}\}$ represents nnr rules associated with normal states of the system, where the selected products interact as intended. $R_A = \{r_{a1}, r_{a2}, r_{a3}, \dots, r_{nar}\}$ represents nar rules related with abnormal states of the system where interactions between the selected products interact unexpectedly (Category-III). $Cf(r_i)$ represents the confidence of r_i, which is between 0 and 1. Confidence for a rule can be calculated as $Cf(r_i) = \frac{SP_i - V_i}{SP_i + V_i}$, where SP_i represents the number of instances for which r_i holds true (i.e., support) and V_i represents the number of instances that violate r_i (i.e., violation). An instance represents a set of values for configurable parameters of the selected products and corresponding system states. Based on confidence, support, and violation we further classify the R_N into two categories using two thresholds: High confidence rules (Category-I) where $Cf(r_i) > TH1$ and $(SP_i + V_i) > TH2$ and Low confidence rules (Category-II) where $Cf(r_i) \le TH1$ or $(SP_i + V_i) \le TH2$. Note that we used 0.9 (TH1) and 10 (TH2) for our experiment to classify CPL rules. Analyzing the effect of these thresholds on the performance of SBRM requires further investigation. In Figure C-2, we present three sanitized CPL rules (r1-r3) mined for the case study. For example, r3 describes that if the encryptions of products P1 and P2 are set to Enc-1 and Enc-2 respectively, the call will fail. $S = \{s_1, s_2, \dots, s_{ns}\}$ represents potential configuration solutions, where $ns = \prod_1^{ncp} (CPV_i)$, which is approximately 1.03^{e33} for our case study. Each solution s_j has a set of configuration values for ncp configuration parameters such that $s_j = \{cpvs_{j1}, \dots, cpvs_{jncp}\}$. $E_j = \{e_1, e_2, \dots, e_{ne}\}$ is a set of effectiveness measures for evaluating solution s_j.

We can then formulate the configuration generation problem as searching a non-dominant solution set S_R from ns solutions to obtain the highest effectiveness.

$$\forall_{s_r \in S_R} \forall_{i=1 \text{ to } ns} \forall_{j=1 \text{ to } ne} \exists \text{ Effect}(s_r, e_j) > \text{Effect}(s_i, e_j) \wedge s_i \notin S_R \quad (1)$$

$\text{Effect}(s_i, e_j)$ refers to the jth effectiveness measure of solution s_i.

3.2 Objectives and Effectiveness Measures

The objectives are defined based on the three categories of rules (Section 3.1). Before presenting the objectives and effectiveness measures, we first define the distance function that is used to assess the effectiveness measures. The distance function indicates to what extent a configuration solution conforms to a rule.

$$D(r_i, s) = \frac{\sum_{i=0}^{CL} d(cl_i, cpv_i)}{MCL} \quad (2)$$

where $D(r_i, s)$ calculates the distance between rule r_i and solution s. In equation (2), $d(cl_i, cpv_i)$ calculates the branch distance between a clause cl_i from rule r_i and corresponding configuration value cpv_i from solution s. MCL is the maximum number of clauses in all the rules.

To calculate the distance between $\mathrm{cl_i}$ and $\mathrm{cpv_i}$ as a branch distance, we use the distance calculation formula provided in [17, 18].

3.2.1 Avoid configuration data satisfying or close to satisfying high confidence rules with normal states

This objective is to avoid generating configuration data that completely or close to satisfy the rules in Category-I. The effectiveness measure (AHNS) corresponding to this objective can be calculated as:

$$\mathrm{AHNS}(\mathrm{R_N}, \mathrm{s}) = \sum_{\mathrm{i}=1}^{\mathrm{nnr}} \mathrm{Cf}(\mathrm{r_i}) * \mathrm{D}(\mathrm{r_i}, \mathrm{s}) \mid \mathrm{Cf}(\mathrm{r_i}) > \mathrm{TH1}\ \&\&\ (\mathrm{SP_i} + \mathrm{V_i}) > \mathrm{TH2} \quad (3)$$

where $\mathrm{AHNS}(\mathrm{R_N}, \mathrm{s})$ takes $\mathrm{R_N}$ (the set of rules related to the normal states) and one solution s as input and gives the effectiveness measure as output. To determine AHNS, we calculate the sum of weighted distances for all rules in Category-I, where the confidence of each rule is greater than threshold TH1 (i.e., 90%) and the sum number of support and violation instances for each rule is more than TH2 (i.e., 10). Weighted distance of $\mathrm{r_i}$ is calculated by multiplying $\mathrm{Cf}(\mathrm{r_i})$ with $\mathrm{D}(\mathrm{r_i}, \mathrm{s})$.

3.2.2 Generate configuration data satisfying or close to satisfying low confidence rules with normal states

This objective is to generate configuration data within the configuration space that satisfy Category-II as well as its nearby space. The nearby space contains configuration data for which the distance to the rules in Category-II is close to 0 but not exactly 0. These configuration data might help to either improve the confidence of correct rules by increasing their support or filter out incorrect ones by increasing their violation and hence reducing their confidence. The effectiveness measure (NLNS) related to the second objective can be calculated as:

$$\mathrm{NLNS}(\mathrm{R_N}, \mathrm{s}) = \sum_{\mathrm{i}=1}^{\mathrm{nnr}} \mathrm{Cf}(\mathrm{r_i}) * (1 - \mathrm{D}(\mathrm{r_i}, \mathrm{s})) \mid \mathrm{Cf}(\mathrm{r_i}) \leq \mathrm{TH1}\ ||\ (\mathrm{SP_i} + \mathrm{V_i}) \leq \mathrm{TH2} \quad (4)$$

where $\mathrm{NLNS}(\mathrm{R_N}, \mathrm{s})$ takes $\mathrm{R_N}$ (the set of rules associated with the normal states) and solution s as input and outputs NLNS. Since we want to explore the configuration space near the configuration data satisfying the rules in Category-II, configuration data with a smaller distance to the rules in Category-II is preferred. Therefore, we use $(1 - \mathrm{D}(\mathrm{r_i}, \mathrm{s}))$ in the $\mathrm{NLNS}(\mathrm{R_N}, \mathrm{s})$. To calculate NLNS, we calculate the sum of the weighted distance (i.e., calculated by multiplying $\mathrm{Cf}(\mathrm{r_i})$ with $(1 - \mathrm{D}(\mathrm{r_i}, \mathrm{s}))$) of a solution to all the rules in Category-II, where the confidence of each rule is less than or equals to TH1 (i.e., 90%) or the sum number of support and violation instances for each rule is less or equal to TH2 (i.e., 10).

3.2.3 Generate configuration data satisfying or close to satisfying rules with abnormal states

This objective is to generate configuration data within the configuration space that satisfy Category-III and its nearby space. The rules in Category-III are of high interest in our context because they indicate situations where interactions of the selected products fail. The effectiveness measure (NAS) for this objective can be calculated as:

$$\mathrm{NAS}(\mathrm{R_A}, \mathrm{s}) = \sum_{\mathrm{i}=1}^{\mathrm{nar}} \mathrm{Cf}(\mathrm{r_i}) * \left(1 - \mathrm{D}(\mathrm{r_i}, \mathrm{s})\right) \quad (5)$$

where $\mathrm{NAS}(\mathrm{R_A}, \mathrm{s})$ takes rule set $\mathrm{R_A}$ (related to the abnormal states) and solution s as input. To calculate NAS, we calculate the sum of weighted distances for all the rules in $\mathrm{R_A}$ (Category-III).

Table C-1. Overview of the experiment design*

RQ	Tasks	Description	Evaluation metrics	Algorithm's Parameters	Statistical tests
RQ1	T_1	Comparing fitness values and HV	– Individual objectives and Overall Fitness – HV	– Population size = 200 – maxEvaluations = 20K – Crossover rate = 0.9 – Mutation rate =1/(Total number of configuration parameters) – Total runs = 10	Man-Whitney U-test and Vargha and Delaney $\hat{A}_{12}$
RQ2	T_2	Comparing rule sets based machine-learning quality measurements	– Accuracy (%) – F/C Precision (%) – F/C Recall (%) – F/C F-Measurement		
RQ3	T_3				

* F= Failed (Abnormal state), C=Connected (Normal state)

3.3 Fitness Function

We first normalize the three effectiveness measures with $\mathrm{nor}(\mathrm{F}(\mathrm{x})) = \left(\frac{\mathrm{F}(\mathrm{x})-\mathrm{F}_{\mathrm{min}}}{\mathrm{F}_{\mathrm{max}}-\mathrm{F}_{\mathrm{min}}}\right)$, where $\mathrm{F}(\mathrm{x})$ is an effectiveness measure function, $\mathrm{F}_{\mathrm{max}}$ and $\mathrm{F}_{\mathrm{min}}$ are the maximum and minimum values of the effectiveness measure. For AHNS, $\mathrm{F}_{\mathrm{min}}$ is 0 when the distance between all the rules in Category-I and solution s is 0. $\mathrm{F}_{\mathrm{max}}$ can be calculated as $\sum_{\mathrm{i}=1}^{\mathrm{nnr}} \mathrm{Cf}(\mathrm{r}_{\mathrm{i}})$ where the distance between all the rules in Category-I and solution s is 1. For NLNS and NAS, $\mathrm{F}_{\mathrm{min}}$ is 0 when the distance between all the rules in the corresponding category and solution s is 1. Corresponding to NLNS and NAS, $\mathrm{F}_{\mathrm{max}}$ can be calculated as $\sum_{\mathrm{i}=1}^{\mathrm{nnr}} \mathrm{Cf}(\mathrm{r}_{\mathrm{i}})$ and $\sum_{\mathrm{i}=1}^{\mathrm{nar}} \mathrm{Cf}(\mathrm{r}_{\mathrm{i}})$ respectively, where the distance between all the rules and solution s is 0.

With the three effectiveness measures, we define the fitness function based on the three objectives as follow:

$$\mathrm{F}(\mathrm{O}_1) = 1 - \mathrm{Nor}\ (\mathrm{AHNS}(\mathrm{R}_{\mathrm{N}}, \mathrm{s})) \qquad (6)$$

$$\mathrm{F}(\mathrm{O}_2) = 1 - \mathrm{Nor}\ (\mathrm{NLNS}(\mathrm{R}_{\mathrm{N}}, \mathrm{s})) \qquad (7)$$

$$\mathrm{F}(\mathrm{O}_3) = 1 - \mathrm{Nor}\ (\mathrm{NAS}(\mathrm{R}_{\mathrm{A}}, \mathrm{s}) \qquad (8)$$

Note that, in the above equations, we define our search problem as a minimization problem by subtracting each normalized effectiveness measure from 1 to ensure that a solution with a value closer to 0 is better.

The fitness function with the three objectives is combined with NSGA-II to address the optimization problem. We implemented our problem in jMetal by encoding all the configuration parameters in the solution s as integer variables, where a variable cp_{i} holds a value $\mathrm{cpv}_{\mathrm{ij}}$ such that $\mathrm{cpv}_{\mathrm{ij}} \in \mathrm{CPV}_{\mathrm{i}}$. Initially, all variables in s are initialized with random values. During the search, SBRM generates optimized solutions guided by the fitness function.

4. Evaluation

We present experiment setup in Section 4.1, execution in Section 4.2, and results in Section 4.3. In Section 4.4, we present overall discussion and Section 4.5 presents threats to validity.

4.1 Experimental Setup

First, we present the experiment design including research questions (Section 4.1.1) followed by the case study (Section 4.1.2) and evaluation metrics (Section 4.1.3). Lastly, we present evaluation tasks, parameter settings, and statistical tests used for analysis (Section 4.1.4).

4.1.1 Research Questions

In SBRM, we apply commonly used NSGA-II [19-21] for generating configuration data as NSGA-II has proven to be effective for solving various software engineering problems such as test case prioritization and cost estimation [20, 22].

The goal of the evaluation is to assess if combining machine learning with NSGA-II in the rule mining process can improve the quality of rules. As RS is typically used as the comparison baseline [22, 23]; therefore, we investigate if NSGA-II is effective to solve the configuration generation problem and then compare the quality of rules produced from SBRM (with NSGA-II) with rules mined by RS based approach (i.e., called RBRM). To further assess the effectiveness of SBRM, we also compare rules mined from SBRM with rules mined from real data extracted from test case execution logs (i.e., called RDBRM). Thus, the evaluation is designed to answer the following three research questions:

RQ1. Is NSGA-II effective to solve the configuration generation problem as compared to RS?

RQ2. Does SBRM produce better quality rules than RBRM in terms of machine learning measurements?

RQ3. Does SBRM produce better quality rules than RDBRM in terms of machine learning measurements?

4.1.2 Case Study

Cisco Systems, Norway provides a variety of VCSs to facilitate high-quality virtual meetings [23]. Cisco has developed several product lines for VCS including C-Series, MX-Series, and SX-Series [3]. Each product from these different product lines has a large number of configuration parameters (e.g., Protocol and Encryption), which need to be configured before making calls. For each VCS we have a set of state variables representing the state of VCS (e.g., call status, camera connection status) that varies according to different hardware and software configurations. For our experiment, we used two real products C60 and MX300 developed by Cisco, which belong to C-series and MX-series, respectively. Simula Research Laboratory has a long-term collaboration with Cisco, Norway under Certus-SFI[14]. As part of our collaboration, we have access to several VCSs at our lab and thus we used these systems for our experiments. Therefore, our case study is real, but the experiment wasn't performed in the real industrial setting of Cisco.

For comparing the quality of rules produced using SBRM with ones mined by RDBRM, we obtained 9,989 test case execution logs from Cisco. Each test log contains a test case script and configurations and statuses representing the system states for all the products involved in the test case. The configurations and their corresponding system states (i.e., statuses) contained in the execution logs can be used to mine the rules. To extract the data, first, we obtained 3963 relevant (i.e., invoking the Dial command) logs from 9,989 test execution logs automatically, where the testing scenario is about making a call from one product to another. Second, corresponding to all relevant execution logs, we extracted configurations and statuses for the products involved in the test cases corresponding to execution logs. Finally, we use the extracted configurations and corresponding statuses to mine the rules.

4.1.3 Evaluation Metrics

To answer RQ1, we compared NSGA-II with RS in terms of the three objectives, and the overall fitness. Additionally, we also compared NSGA-II with RS in terms of HV, which is commonly

[14] www.certus-sfi.no

used to measure the overall performance of multi-objective search algorithms (e.g., NSGA-II) [24]. HV is for obtaining the volume in the objective space covered by members of Pareto fronts for measuring both convergence and diversity [25].

To answer RQ2 and RQ3, we compared SBRM with RBRM and SBRM with RDBRM respectively, based on four machine-learning quality measurements (MLQMs): Accuracy of the classifier, *Precision*, *Recall*, and *F-measure* for each class (i.e., call status in our case), which are calculated with 10 times 10-fold cross-validation [26]. Accuracy indicates the overall performance of PART by specifying the percentage of instances that conforms to the mined rules [27], where one instance contains one specific set of configurations and its corresponding system states.

Precision represents the percentage of instances that are correctly classified divided by the total number of instances covered by rules associated with a specific system state (e.g., connected or failed in our case) [27]. For example, 98% *Precision* for the failed state means that, according to the mined rules, there are 2% of instances whose configurations are identified as invalid ones, which led to the failed state. But actually, they lead to the connected state. The *Recall* represents the percentage of instances that are correctly classified divided by the total number of instances corresponding to a particular system state [27]. For example, 90% *Recall* for the failed state means that configurations of 10% instances are not associated with the failed state according to the mined rules, but these instances actually lead to the failed state. *F-measure* is the harmonic mean of *Precision* and *Recall* [27].

4.1.4 Experimental Tasks, Parameter Settings, and Statistical Analysis

As shown in Table C-1, we designed three tasks (T1-T3) for addressing RQ1-RQ3. T1 is to compare NSGA-II with RS in terms of HV, the three individual objectives, and the overall fitness. T2 and T3 are for comparing the quality of rules produced from SBRM with RBRM and RDBRM respectively, evaluated based on machine-learning quality measurements.

As shown in Table C-1(column 5), we used the default settings for NSGA-II as implemented in jMetal [28], which are typically recommended [29]. The single point crossover and bit-flip mutation, implemented in jMetal, were applied as crossover and mutation operators, respectively. The total number of configuration parameters is 17 for our case study. We used a population size of 200 where we select all the Pareto Non-dominated solutions for mining the rules. Since selecting the best set of parameters is application dependent [12], we used the default settings provided by Weka [14] for SBRM, RBRM, and RDBRM, which have been used in various contexts for applying the machine learning techniques [12, 30].

To compare SBRM (with NSGA-II) with RBRM and RDBRM, we use the non-parametric Mann-Whitney U-test as recommended in [31] using $\alpha = 0.05$ and the Vargha and Delaney's $\widehat{A}_{12}$ statistics as an effect size measure [32]. For all MLQMs and HV, if $\widehat{A}_{12}$ is less than 0.5, SBRM is better than RBRM/RDBRM, and a value greater than 0.5 means vice versa. Similarly, in the case of fitness values, if $\widehat{A}_{12}$ is greater than 0.5, SBRM is better than RBRM otherwise RBRM is better than SBRM.

4.2 Experimental Execution

We selected the call status as the system state to classify the configurations. A failed call status represents the abnormal state and a connected call status represents a normal state. We selected the call functionality and its associated call status as it is the main functionality of a VCS and other functionalities depend on it.

To mine the initial set of the rules we randomly generate a set of 500 configurations corresponding to two selected products (i.e., C60 and MX300). To get the system state, we configure the two products with the generated configurations and make a call from product A to B for 20 seconds. We made the call for 20 seconds in order to give sufficient time for establishing the call connection. After waiting for 20 seconds, we capture the call status and disconnect the call. We input these 500 configurations along with their corresponding system states to Weka [14] and apply PART [15] to mine the initial set of rules. To refine the rules, we use the initial set of rules to guide the search to generate 200 more configurations. To mine the refined set of rules we repeat the same process (i.e., configuring the products and making the call) to get the call status and mine a new set of rules based on 700 configurations (combining all the configurations generated so far) and corresponding system states. We repeat this incremental and iterative process for three iterations and mine the final set of rules based on a dataset containing 1100 configurations and their call statuses. We used three iterations as a stopping criterion. We also got more than 90% identical rules in the second and third iteration.

4.3 Results and Analysis

In this section, we present the results of our evaluation and answer the research questions.

4.3.1 Effectiveness of search (RQ1)

To answer RQ1, from the results of the Man-Whitney U-test, we notice that p-values corresponding to all fitness values and HV are less than 0.05 showing a significant difference between NSGA-II and RS. $\hat{A}_{12}$ values corresponding to the three objectives are all greater than 0.5 and are less than 0.5 in the case of HV, which suggests that NSGA-II is significantly better than RS.

4.3.2 Comparing SBRM with RBRM (RQ2)

To answer RQ2, we compared SBRM and RBRM in terms of MLQMs based on rules from each iteration as well as overall (i.e., combined the results for all the three iterations) based on MLQMs (Section 4.1.3).

As shown in Table C-2, for the first iteration, although all the $\hat{A}_{12}$ values indicate that SBRM has better performance for all the MLQMs, the *p* values show that the superiority of SBRM is not significant for all the MLQMs except for *Failed Recall*. In iteration-2, SBRM performed significantly better than RBRM with respect to *Accuracy*, *Failed Precision*, *Failed Recall*, and *Failed F-measure*. The results corresponding to iteration-3 and overall (Table C-2) show that SBRM has performed significantly better than RBRM in terms of all the MLQMs. So, as moving from iteration-1 to iteration-3, SBRM starts to perform better than RBRM, which leads to the conclusion that SBRM produces better rules as compared to RBRM with respect to the MLQMs.

4.3.3 Comparing SBRM with RDBRM (RQ3)

To answer RQ3, we compared SBRM with RDBRM iteration-wise as well as overall (i.e., combined the values for all the three iterations) based on MLQMs (Section 4.1.3).

Table C-2. Comparing the quality of rules produced with SBRM and RBRM – $\hat{A}_{12}$ and p-values for (RBRM VS SBRM)

Evaluation metric	Iteration-1		Iteration-2		Iteration-3		Overall		Overall Average	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	RBRM	SBRM
Accuracy	0.104	0.28	**0.010**	**0.16**	**0.002**	**0.10**	**<0.001**	**0.19**	95.7%	97.2%
Connected Precision	0.161	0.31	0.054	0.24	**0.026**	**0.20**	**0.002**	**0.27**	0.945	0.957

Connected Recall	0.173	0.32	0.150	0.31	**0.041**	**0.23**	**0.002**	**0.27**	0.955	0.971
Connected F-Measure	0.186	0.32	0.088	0.27	**0.025**	**0.20**	**0.001**	**0.25**	0.950	0.964
Failed Precision	0.063	0.25	**0.012**	**0.17**	**0.001**	**0.07**	**<0.001**	**0.19**	0.966	0.982
Failed Recall	**0.041**	**0.23**	**0.003**	**0.11**	**0.001**	**0.07**	**<0.001**	**0.16**	0.965	0.978
Failed F-Measure	0.104	0.28	**0.005**	**0.13**	**0.001**	**0.04**	**<0.001**	**0.18**	0.966	0.980

As shown in Table C-3, the results related to all the MLQMs except for *Connected Recall* and *Connected F-measure* for all the iterations as well as overall show that SBRM performed significantly better than RDBRM. Results for *Connected Recall* corresponding to all the iterations as well as overall indicate that RDBRM performed significantly better than SBRM. In iteration-1, iteration-2, and overall there is no significant difference between SBRM and RDBRM in terms of *Connected F-measure* whereas in iteration-3 SBRM outperformed RDBRM. Since for five out of the seven MLQMs, SBRM has performed significantly better than RDBRM whereas RDBRM outperformed SBRM in terms of *Connected Recall* only, it can be concluded that SBRM produces better rules than RDBRM.

Table C-3. Comparing the quality of rules produced with SBRM and RDBRM – $\hat{A}_{12}$ and p-values for (RDBRM VS SBRM)

Evaluation metric	Iteration-1		Iteration-2		Iteration-3		Overall		Actual values RDBRM
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	
Accuracy	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	92.96%
Connected Precision	**0.001**	**0.10**	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.03**	0.934
Connected Recall	**<0.001**	**1.00**	**<0.001**	**1.00**	**<0.001**	**1.00**	**<0.001**	**1.00**	0.994
Connected F-Measure	0.418	0.40	0.418	0.60	**0.012**	**0.200**	0.135	0.400	0.963
Failed Precision	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	0.796
Failed Recall	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	0.260
Failed F-Measure	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	**<0.001**	**0.00**	0.392

4.4 Overall Discussion

For RQ1, we noticed that NSGA-II has outperformed RS in terms of HV, the three objectives as well as the combined. This suggests that our problem is not trivial and requires the search.

For RQ2 and RQ3, we observed that SBRM performed significantly better than RBRM and RSBRM for most of the MLQMs. This is because we guide the search using previously mined rules and generate specific configuration data that tend to either increase or decrease the confidence of a rule. In this way, SBRM converges more rapidly than RBRM to obtain high confidence rules. To further investigate the performance differences of SBRM with RBRM and RDBRM, we calculated the relative improvement (RI) due to SBRM for all MLQMs, across iterations. We calculated the RI with respect to RBRM as $\mathrm{RI}\big(\mathrm{S}(\mathrm{x_{ij}}), \mathrm{R}(\mathrm{x_{ij}})\big) = \big(\mathrm{S}(\mathrm{x_{ij}}) - \mathrm{R}(\mathrm{x_{ij}})\big)$, where $\mathrm{S}(\mathrm{x_{ij}})$ and $\mathrm{R}(\mathrm{x_{ij}})$ give the average values corresponding to the ith MLQM and jth iterations for SBRM and RBRM, respectively. Similarly, to calculate RI with respect to RDBRM, we applied a similar formula as: $\mathrm{RI}\big(\mathrm{S}(\mathrm{x_{ij}}), \mathrm{RD}(\mathrm{x_i})\big) = \Big(\mathrm{S}(\mathrm{x_{ij}}) - \mathrm{RD}(\mathrm{x_i})\Big)$, where $\mathrm{RD}(\mathrm{x_i})$ gives the value of the ith MLQM for RDBRM. Figure C-3 and Figure C-4 show the relative improvement in MLQMs due to SBRM in comparison to RBRM and RDBRM, respectively.

From Figure C-3, one can observe that compared with RDBRM, the relative improvements of SBRM in terms of *Failed Precision*, *Failed Recall*, and *Failed F-measure* are much larger than the relative improvements of the other MLQMs, whereas it is negative in terms of *Connected Recall*. This can be justified by the fact that in SBRM we generate configurations that maximally conform to the rules with the abnormal state (i.e., the failed state). Also, we avoid generating configurations that conform to the high confidence rules with the normal state (i.e., the connected state), which justifies the negative RI value for *Connected Recall*.

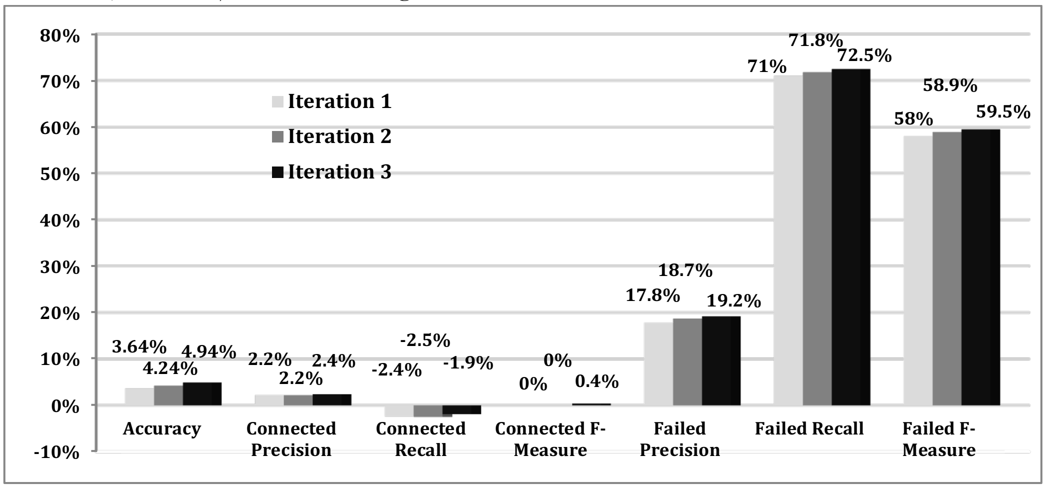


Figure C-3. Relative improvement by SBRM in comparison to RDBRM

Figure C-4 shows that the relative improvement in MLQMs for SBRM as compared to RBRM is not large as it is in comparison to RDBRM, which is probably because the sample size used for mining the rules in SBRM and RBRM is small (i.e., 700, 900, and 1100 for iteration-1, iteration-2, and iteration-3, respectively). Moreover, in these small datasets, 500 initial configurations were the same across the datasets used for SBRM and RBRM, and only maximum 600 (i.e., in iteration-3) configurations were different. On the other hand, the relative improvement for SBRM with respect to RDBRM is large because the datasets used for RDBRM and SBRM were different. Also, the size of the dataset used for RDBRM was large (i.e., 3963). However, from Figure C-4, we can observe an increasing trend of the relative improvement across the three iterations, suggesting that increasing the sample size can increase the relative improvement.

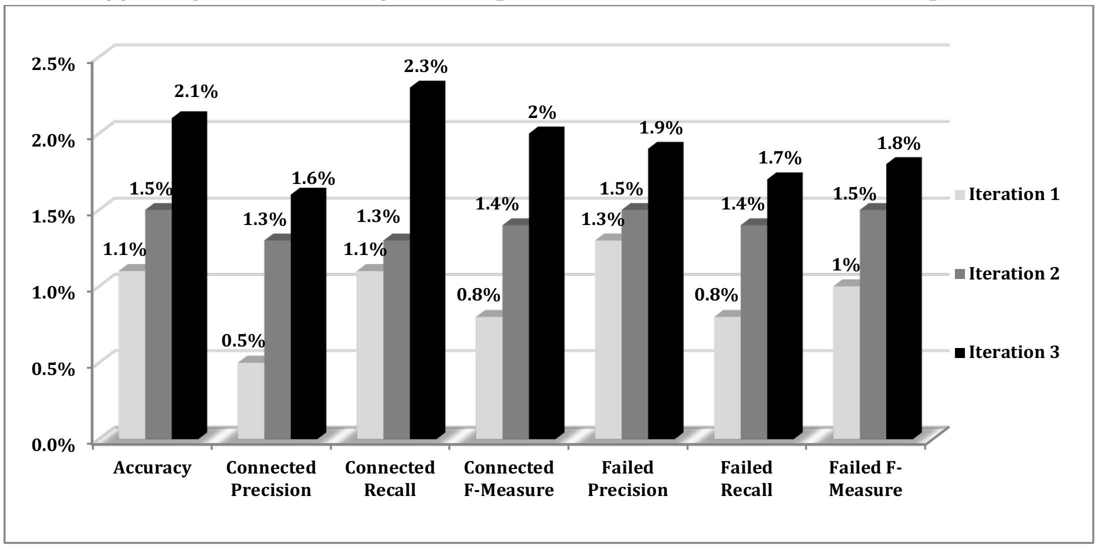


Figure C-4. Relative improvement by SBRM in comparison to RBRM

4.5 Threats to Validity

The threat to internal validity of our study is the selection of parameter settings for the selected search algorithm, which may affect the performance of the algorithm. To mitigate this threat, we used default parameter settings, which have exhibited promising results [33]. Similarly, for the machine-learning algorithm, we also used default parameters settings, as selecting parameter settings is application dependent [12]. The threat to construct validity is the use of termination criteria for the search. We used the same stopping criterion (i.e., the number of fitness evaluations) for both NSGA-II and RS to find the optimal solutions. Another threat can be a selection of stopping criteria for the number of iterations and sample size used for mining the rules. We used three iterations and during each iteration added 200 more configurations to the dataset from the previous iteration due to practical challenges (i.e., the overall cost of the whole process was high particularly on executing configurations and getting corresponding call statuses, which was 50 seconds per configuration). To assess the effect of the sample size, the number of iterations, and different values for the thresholds used to classify the CPL rules, we plan to conduct dedicated empirical studies in the future.

The threat to conclusion validity is due to the random variation inherited in search algorithms. To minimize this threat, we repeated the experiment 10 times to reduce the effect caused by randomness, as recommended in [24, 29]. Moreover, we also applied the Mann-Whitney test to determine the statistical significance of the results and the Vargha and Delaney $\hat{A}_{12}$ statistics as the effect size measure, which are recommended for randomized algorithms [29]. The first threat to external validity is the selection of search algorithm for our study. To reduce this, we selected the most widely used NSGA-II algorithm that has shown promising results in different contexts [20, 22]. The second threat to external validity is the selection of algorithms for rule mining. To tackle this threat, we selected PART, which has proven to be more effective as compared to other well-known algorithms [15, 34]. The third threat to external validity is that we evaluated our approach using only one case study. To mitigate this, we used a real case study, the Cisco Video Conferencing Systems, which contains typical communicating products across multiple product lines. However, a generalization of the results requires additional experiments. In future, we plan to conduct an empirical study using several case studies to evaluate different search algorithms and machine-learning algorithms.

5. Related Work

Search algorithms have been used to solve many problems in the context of PLE [35-37]. Since we are focusing on rule mining; therefore, we only discuss existing studies related to rule mining using machine-learning techniques in the context of PLE. In Section 5.1, we discuss dedicated approaches that focus on mining rules from different artifacts (e.g., source code, configuration file, feature model). In Section 5.2, we discuss approaches such as feature extraction, feature construction and feature recommendation, which mine crosstree constraints.

5.1 Dedicated Rule Mining Approaches

The work in [12] applies Binary Decision Tree-J48 (machine learning algorithm) to infer the constraints from a set of randomly generated product configurations. To classify the configurations as faulty and non-faulty, a computer vision algorithm was used as an oracle. To validate the approach, it was applied to an industrial video generator product line. Rules were evaluated based on expert's opinion and machine-learning measurements such as *Precision* and *Recall*. Results show that on average 86% *Precision* and 80% *Recall* rate can be achieved using the proposed approach.

In [38], Yi et al. proposed an approach to mine the crosstree binary constraints (i.e., requires, excludes) corresponding to a feature model. The approach takes a feature model as input containing the features, their descriptions, and some known crosstree binary constraints. First, it trains LIBSVM classifier (an extension of support vector machine) with existing crosstree binary constraints where the parameters of the classifier are optimized using the genetic algorithm to minimize the error rate of the classifier. Second, it extracts all the feature pairs, and finally, the optimized classifier finds the candidate features of binary constraints. The approach was validated using two feature models collected from SPLOT repository. Results show that rules with high *Recall* (i.e., close to 100%) and the variable low *Precision* (on average 42%) can be achieved using proposed approach.

In [39], another approach is presented for mining the crosstree constraints. It constructs configuration matrix (i.e., product-features matrix) from configuration files and extracts crosstree constraints using an association rule mining technique (i.e., Apriori algorithm). Rules are pruned using minimum support and minimum confidence thresholds. The approach was evaluated using a large-scale industrial software product line for embedded systems. The evaluation shows that a large number of rules with variable support (i.e., 80% to 99%) and confidence (i.e., 90% to 100%) can be identified. The majority of the rules were identified with support ranging from 80% to 85%.

In [40], an approach is presented to extract configuration constraints from existing C codebases using static analysis. It uses build time errors (e.g., pre-processor, parser, type, and link errors) as the oracle to classify the low-level system configurations (i.e., build and code files) and mine the constraints. To assess the accuracy of extracted rules, they were compared with the existing constraints specified in developer's created variability models. The approach was validated using four open source case studies (uClibc, BusyBox, eCos, and the Linux kernel). Results show that up to 19% of the total constraints can be recovered automatically from the source code, which assures successful build with the accuracy of 93%. In [13], an extension of [40] is presented in which the authors improved the static analysis and increased the recoverability rate by 9%. Additionally, an empirical study is also presented that identifies the sources of constraints.

5.2 Non-Dedicated Rule Ming Approaches

In [41], Czarnecki et al. proposed an extension of feature model called probabilistic feature model. To extract crosstree constraints from existing formally defined products, a rule mining process is presented that uses association-mining techniques to mine the constraints. The proposed mining process was applied on a small case study of Java Applets. Rules were evaluated based on machine-learning measurements (i.e., support and confidence).

In [42], an approach is proposed to model and recommend product features for any particular domain based on the product description provided by the domain expert. To mine association rules between product features, association rule mining techniques are applied to configuration matrix (i.e., product-features matrix). The proposed approach was validated with 20 different product categories using product descriptions available at SoftPedia [43]. Hariri et al. [44] extend the work presented in [42]. In [44], different clustering algorithms used to cluster the features and construct products by feature matrix were compared. The evaluation was also improved by applying the approach on diverse domains as well as a large project of a software suite for remote collaboration. Results show that rules with different *Precision* and *Recall* rates can be mined according to the threshold set for the confidence.

The work in [6] presents an approach to synthesize attributed feature models (AFM) from a set of product descriptions in the form of tables (i.e., configuration matrix). An algorithm is proposed that uses implication graph and mutex graph constructed from configuration matrix to extract the crosstree constraints. For extracting the relational constraints defined on values of attributes, the algorithm uses domain knowledge or selects the boundary values of attributes randomly when domain knowledge is not provided. The approach was validated using random configuration matrices as well as a real-world case study. Results show that the proposed algorithm can be used to mine a large number of rules for large-scale case studies.

Davril et al. [45] proposed an approach to construct a feature model automatically from informal product descriptions available over the Internet. To mine the implication rules of features, CFP-growth algorithm and Apriori algorithm are applied on configuration matrix (i.e., product-features matrix). The proposed approach was applied to a case study of antivirus software using the product descriptions available at SoftPedia [43].

5.3 Summary

All the approaches discussed above focus on mining binary crosstree constraints (requires and excludes) between different features of a product line or constraints on features' attributes. In our study, we focus on mining rules between configuration parameters and system behaviors of interacting products across product lines. Additionally, we defined three objectives (Section 3.2) for generating configuration data, which are fed to the machine-learning tool in order to refine rules. To evaluate the quality of rules, all the approaches discussed above have used machine-learning quality measurements (e.g., *Precision*). We also evaluated the quality of constraints based on machine-learning quality measurements. Additionally, we also compared the rules produced using SBRM with the rules mined based on real data.

6. Conclusion

We presented an incremental and iterative approach (named as SBRM) for mining rules for configurations of communicating products belonging to different product lines. To mine rules, we combine multi-objective search with machine learning techniques. To use the search in the rule mining process, we defined three objectives and integrated them with the widely used multi-objective optimization algorithm—NSGA-II. We compared SBRM with RS based approach (RBRM) in terms of the three objectives, HV, and machine learning quality measurements. The results of the statistical tests show that SBRM performed significantly better than RBRM for all the three objectives, HV, and machine learning quality measurements. In comparison with the rules mined based on real data (RDBRM), SBRM has performed significantly better particularly for *Failed Precision*, *Failed Recall*, and *Failed F-measure* where SBRM improved them by 18%, 72%, and 59% respectively, when compared with RDBRM.

Acknowledgement

This work was supported by the Zen-Configurator project funded by the Research Council of Norway (grant no. 240024/F20) under the category of Young Research Talents of the FRIPO funding scheme. Tao Yue and Shaukat Ali are also supported by RCN funded MBT4CPS project, RFF Hovedstaden funded MBE-CR project, EU Horizon 2020 funded U-Test project, RCN funded Certus SFI, and the EU COST action MPM4CPS.

References

1. Holl, G., P. Grünbacher, and R. Rabiser, A systematic review and an expert survey on capabilities supporting multi product lines. Information and Software Technology (IST), 2012. **54**(8): p. 828-852.
2. Rosenmüller, M. and N. Siegmund. Automating the Configuration of Multi Software Product Lines. in Proceeding of International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). 2010. Elsevier.
3. Video Conferencing Systems Available from: http://www.cisco.com/.
4. ULMA Handling Systems. Available from: http://www.ulmahandling.com.
5. Yue, T., S. Ali, and B. Selic. Cyber-Physical System Product Line Engineering: Comprehensive Domain Analysis and Experience Report. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2015. ACM.
6. Bécan, G., et al. Synthesis of attributed feature models from product descriptions. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2015. ACM.
7. Nie, K., et al. Constraints: the core of supporting automated product configuration of cyber-physical systems. in Proceeding of International Conference on Model-Driven Engineering Languages and Systems (MODELS). 2013. Springer.
8. Safdar, S.A., et al. Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering. in Proceeding of International Conference on System Analysis and Modeling (SAM). 2016. Springer.
9. Lu, H., et al., Model-based Incremental Conformance Checking to Enable Interactive Product Configuration. Information and Software Technology (IST), 2015. **72**: p. 68-89.
10. Lu, H., et al., Nonconformity Resolving Recommendations for Product Line Configuration, in International Conference on Software Testing. 2016, IEEE. p. 57-68.
11. Lu, H., et al., Zen-CC: An Automated and Incremental Conformance Checking Solution to Support Interactive Product Configuration, in 25th International Symposium on Software Reliability Engineering. 2014, IEEE. p. 13-22.
12. Temple, P., et al. Using Machine Learning to Infer Constraints for Product Lines. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2016. ACM.
13. Nadi, S., et al., Where do configuration constraints stem from? an extraction approach and an empirical study. IEEE Transactions on Software Engineering (TSE), 2015. **41**(8): p. 820-841.
14. Witten, I.H., E. Frank, and M.A. Hall, Data Mining: Practical machine learning tools and techniques. Third ed. 2011: Morgan Kaufmann.
15. Frank, E. and I.H. Witten. Generating accurate rule sets without global optimization. in Proceeding of International Conference on Machine Learning (ICML). 1998. University of Waikato, Department of Computer Science.
16. Satti, A., N. Cercone, and V. Keselj, Experiments in Web Page Classification for Semantic Web, in Workshop on Web-based Support Systems. 2004. p. 137-141.
17. McMinn, P., Search-based software test data generation: a survey. Software Testing Verification and Reliability (STVR), 2004. **14**(2): p. 105-156.
18. Ali, S., et al., Generating test data from OCL constraints with search techniques. IEEE Transactions on Software Engineering (TSE), 2013. **39**(10): p. 1376-1402.
19. Deb, K., et al., A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions on, 2002. **6**(2): p. 182-197.
20. Sarro, F., A. Petrozziello, and M. Harman. Multi-objective software effort estimation. in Proceeding of International Conference on Software Engineering (ICSE). 2016. ACM.
21. Pradhan, D., et al., Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study, in Genetic and Evolutionary Computation Conference. 2016, ACM. p. 1085-1092.
22. Pradhan, D., et al. STIPI: Using Search to Prioritize Test Cases Based on Multi-objectives Derived from Industrial Practice. in Proceeding of International Conference on Testing Software and Systems (ICTSS). 2016. Springer.
23. Wang, S., et al. Multi-objective test prioritization in software product line testing: an industrial case study. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2014. ACM.
24. Wang, S., et al., A Practical Guide to Select Quality Indicators for Assessing Pareto-based Search Algorithms in Search-Based Software Engineering, in International Conference on Software Engineering (ICSE). 2016.

25. Nebro, A.J., et al., AbYSS: Adapting scatter search to multiobjective optimization. Evolutionary Computation, IEEE Transactions on, 2008. **12**(4): p. 439-457.
26. Sokolova, M. and G. Lapalme, A systematic analysis of performance measures for classification tasks. Information Processing & Management (IPM), 2009. **45**(4): p. 427-437.
27. Han, J., J. Pei, and M. Kamber, Data mining: concepts and techniques. 2011: Elsevier.
28. Durillo, J.J. and A.J. Nebro, jMetal: A Java framework for multi-objective optimization. Advances in Engineering Software, 2011. **42**(10): p. 760-771.
29. Arcuri, A. and L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in 33rd International Conference on Software Engineering. 2011, IEEE. p. 1-10.
30. Ali, S. and K.A. Smith, On learning algorithm selection for classification. Applied Soft Computing, 2006. **6**(2): p. 119-138.
31. Mann, H.B. and D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other. The Annals of Mathematical Statistics, 1947. **18**(1): p. 50-60.
32. Vargha, A. and H.D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics (JEBS), 2000. **25**(2): p. 101-132.
33. Arcuri, A. and G. Fraser. On parameter tuning in search based software engineering. in Proceeding of International Symposium On Search Based Software Engineering (SSBSE). 2011. Springer.
34. Holmes, G., M. Hall, and E. Prank. Generating rule sets from model trees. in Proceeding of Australasian Joint Conference on Artificial Intelligence (AI). 1999. Springer.
35. Lopez-Herrejon, R.E., L. Linsbauer, and A. Egyed, A systematic mapping study of search-based software engineering for software product lines. Information and Software Technology (IST), 2015. **61**: p. 33-51.
36. Harman, M., et al. Search based software engineering for software product line engineering: a survey and directions for future work. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2014. ACM.
37. Wang, S., S. Ali, and A. Gotlieb, Cost-effective test suite minimization in product lines using search techniques. Journal of Systems and Software (JSS), 2014. **103**: p. 370-391.
38. Yi, L., et al. Mining binary constraints in the construction of feature models. in Proceeding of International Requirements Engineering Conference (RE). 2012. IEEE.
39. Zhang, B. and M. Becker. Mining complex feature correlations from software product line configurations. in Proceeding of International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). 2013. ACM.
40. Nadi, S., et al. Mining configuration constraints: Static analyses and empirical results. in Proceeding of International Conference on Software Engineering (ICSE). 2014. ACM.
41. Czarnecki, K., S. She, and A. Wasowski. Sample spaces and feature models: There and back again. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2008. IEEE.
42. Dumitru, H., et al. On-demand feature recommendations derived from mining public product descriptions. in Proceeding of International Conference on Software Engineering (ICSE). 2011. IEEE.
43. Softpedia. Available from: http://www.softpedia.com.
44. Hariri, N., et al., Supporting domain analysis through mining and recommending features from online product listings. IEEE Transactions on Software Engineering (TSE), 2013. **39**(12): p. 1736-1752.
45. Davril, J.-M., et al. Feature model extraction from large collections of informal product descriptions. in Proceeding of Joint Meeting on Foundations of Software Engineering (FSE). 2013. ACM.

Paper D

Using Multi-Objective Search and Machine Learning to Infer Rules Constraining Product Configurations

Safdar Aqeel Safdar, Tao Yue, Shaukat Ali, Hong Lu

Published in the Journal of Automated Software Engineering (ASE), 2019.

Abstract

Modern systems are being developed by integrating multiple products within/across product lines that communicate with each other through information networks. Runtime behaviors of such systems are related to product configurations and information networks. Cost-effectively supporting Product Line Engineering (PLE) of such systems is challenging mainly because of lacking the support of automation of the configuration process. Capturing rules is the key for automating the configuration process in PLE. However, there does not exist explicitly-specified rules constraining configurable parameter values of such products and product lines. Manually specifying such rules is tedious and time-consuming. To address this challenge, in this paper, we present an improved version (named as SBRM$^+$) of our previously proposed Search-based Rule Mining (SBRM) approach. SBRM$^+$ incorporates two machine learning algorithms (i.e., C4.5 and PART) and two multi-objective search algorithms (i.e., NSGA-II and NSGA-III), employs a clustering algorithm (i.e., k-means) for classifying rules as high or low confidence rules, which are used for defining three objectives to guide the search. To evaluate SBRM$^+$ (i.e., SBRM$^+_{\text{NSGA-II}}$-C45, SBRM$^+_{\text{NSGA-III}}$-C45, SBRM$^+_{\text{NSGA-II}}$-PART, and SBRM1$^+_{\text{NSGA-III}}$-PART), we performed two case studies (Cisco and Jitsi) and conducted three types of analyses of results: difference analysis, correlation analysis, and trend analysis. Results of the analyses show that all the SBRM$^+$ approaches performed significantly better than two Random Search-based approaches (RBRM$^+$-C45 and RBRM$^+$-PART) in terms of fitness values, six quality indicators, and 17 machine learning quality measurements (MLQMs). As compared to RBRM$^+$ approaches, SBRM$^+$ approaches have improved the quality of rules based on MLQMs up to 27% for the Cisco case study and 28% for the Jitsi case study.

Keywords: Product Line; Configuration; Rule Mining; Multi-Objective Search; Machine Learning; Interacting Products

1 Introduction

Product Line Engineering (PLE) is a well-acknowledged paradigm to improve the productivity of developing products with higher quality and at a lower cost. By benefiting from PLE, more and more systems are developed by integrating different products, which often belong to different product lines, and communicate and interact with each other through information networks [27, 28]. An example of such systems is video conferencing systems [71] (VCSs). These systems are highly configurable as each product has a large number of configurable parameters (e.g., a VCS product developed by Cisco [15] can have more than 120 configurable parameters) offering different configuration options to users (Figure D-1). For example, in case of VCSs, users can select different protocols for making a call. Each product has a set of operations that enable it to communicate/interact with other products (Figure D-1). Each product has state variables for defining system states. At runtime, configured products belonging to multiple product lines communicate (e.g., via information networks) with each other [27, 28] (Figure D-1). Thus, runtime behaviors of such systems not only depend on the configuration of these communicating

[15] www.cisco.com/c/en/us/products/collaboration-endpoints/index.html

products but are also influenced by information networks (also named as communication medium). Note that configuration in our context is about assigning a set of values to configurable parameters (i.e., including communication medium specific) of communicating products.

Cost-effective PLE is challenging mainly because of the lack of the support for automation of the configuration process [31, 32]. Capturing rules is the key to enabling automation of various configuration functionalities (e.g., consistency checking, decision propagation, and decision ordering) [3, 34, 72, 73]. In our context, such rules describe how configurations of communicating products within/across product lines impact their runtime interactions via information networks. We name such rules as Cross Product Lines (CPL) rules. CPL rules are important for two reasons. First, they can be used to identify invalid configurations where products may fail to interact due to, for example, violated dependencies among features of interacting products [38]. Identified invalid configurations can help developers to maintain current product lines or develop future product lines. Second, CPL rules can provide support for enabling (automated or semi-automated) configurations of products of future deployments. However, the literature does not provide sufficient support to mine such rules, as it mainly focuses on mining rules constraining product configurations within a single product line [32, 37].

As mentioned in [38], rules (i.e., configuration constraints) can be identified from either domain knowledge or testing of the system. Manually specifying such rules based on domain knowledge is tedious and time-consuming, and heavily relies on experts' knowledge of the domain. Moreover, certain information is only known at runtime (e.g., network related information such as bandwidth, traffic congestion , and maximum transmission unit (MTU) size) [38], which makes CPL rules only possible to be captured at runtime. Identifying CPL rules via testing has its own challenges, as the configuration space is typically very large and testing all possible configurations is infeasible. Besides, in practice, testers often use valid configurations to test a system [38]. Therefore, identifying CPL rules requires an automated approach without exhaustively exploring all possible configurations of communicating products within/across product lines.

In [37], a rule mining approach was proposed to mine rules for a product line where product configurations are generated randomly and labeled as faulty or non-faulty. Labeled product configurations are the input to the classification algorithm of J48 [74] to mine rules. However, randomly generating configurations to mine rules is inefficient, as rules with all classes are not equally important (i.e., rules with faulty classes are more important than non-faulty ones). We advanced one step further by employing search to generate product configurations with three heuristics and our initial investigation was presented in our previous work [75], where we proposed an approach, named as Search-based Rule Mining (*SBRM*), combining multi-objective search with machine-learning to mine CPL rules. The three heuristics aim to generate configurations maximally violating high confidence rules with non-faulty classes and satisfying low confidence rules with non-faulty classes and rules with faulty classes). *SBRM* has three major components (Figure D-1): 1) *Initial Configuration Generation*: randomly generating an initial set of configurations for communicating products; 2) *Rule Mining*: taking the generated configurations as input along with corresponding system states and applying the machine learning algorithm to mine CPL rules; and 3) *Search-based Configuration Generation*: taking the mined CPL rules as input and generating an another set of configurations using multi-objective search algorithm, which are combined with the previously generated configurations to mine a refined set of CPL rules. *SBRM* obtains CPL rules with different degrees of confidence (i.e., the probability of being correct) with an emphasis on mining rules that can reveal invalid configurations by specifying the configurations that may lead to abnormal (i.e., unwanted) system states [40]. Instead of collecting a large amount of data required for machine learning all at once, we obtain input data incrementally with multiple iterations. During each iteration, we use rules mined from the previous iteration to guide the search for generating configurations. Newly generated configurations are combined with those from all the previous iterations to incrementally refine the aforementioned rules.

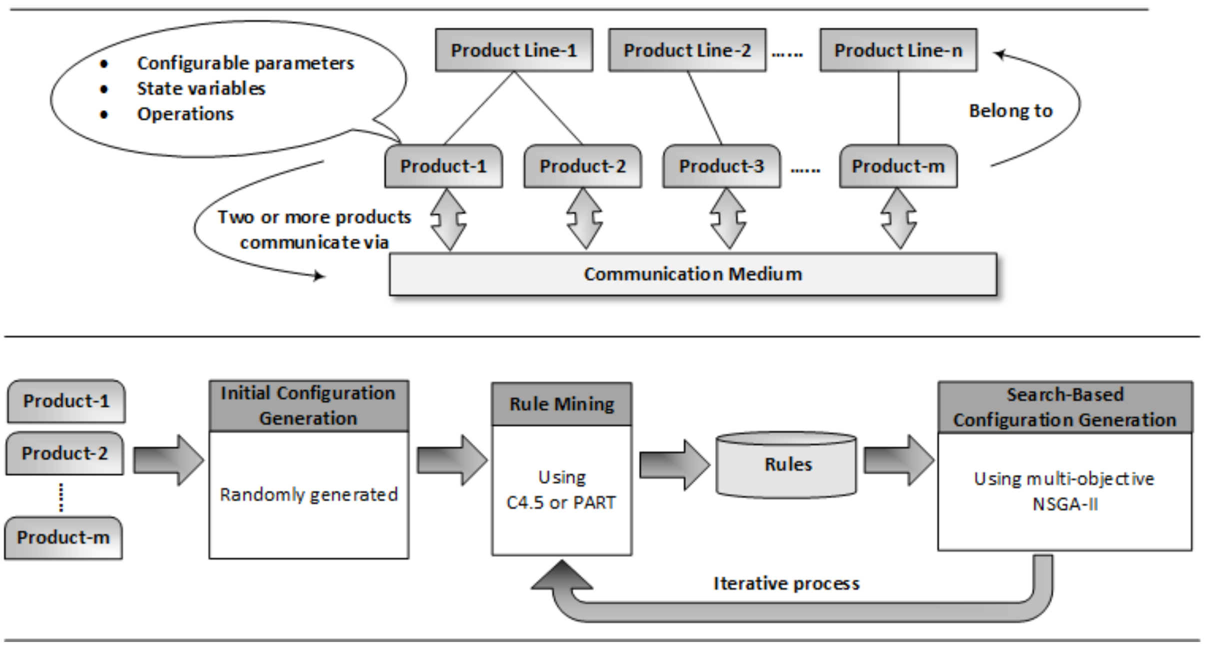


Figure D-1. The overall context and scope of SBRM and SBRM$^+$

In our previous investigation [75], we applied the PART algorithm as the learning algorithm and NSGA-II as the search algorithm in *SBRM*. We validated the approach using a relatively small-sized real-world case study of two communicating VCS products belonging to two different product lines with 17 configurable parameters. In this paper, we extend the prior work by making several additional contributions:

- A significantly improved version of *SBRM* (called *SBRM*$^+$) is proposed for mining CPL rules constraining configurations of communicating products across/within product lines.
    - A clustering algorithm (i.e., *k*-means) is employed in *SBRM*$^+$ (as compared to using thresholds in *SBRM*) for classifying rules as high and low confidence rules, which are used for defining the three heuristics/objectives.
    - Two multi-objective search algorithms NSGA-II and NSGA-III are integrated into *SBRM*$^+$, whereas in *SBRM*, we used only NSGA-II.
    - Two decision tree based rule mining algorithms PART and C4.5 are integrated into *SBRM*$^+$ (referred as *SBRM*$^+_{NSGA\text{-}II}$*-C45*, *SBRM*$^+_{NSGA\text{-}III}$*-C45*, *SBRM*$^+_{NSGA\text{-}II}$*-PART*, and *SBRM*$^+_{NSGA\text{-}III}$*-PART* in the rest of the paper), whereas in *SBRM*, we used only PART.
- The *SBRM*$^+$ approaches were evaluated by performing a real-world case study of three communicating VCS products belonging to three different product lines (Cisco) with 27 configurable parameters and a real-world open source case study of three products of

Audio/Video Internet Phone and Instant Messenger, belonging to the same product line (Jitsi) with 39 configurable parameters. Note that evaluation results presented in this paper are based on new experiments conducted using two relatively larger case studies (with 27 and 39 configurable parameters) of three communicating products and no results were taken from our previous work [75].

- We conducted three types of analyses of results for both case studies: *difference analysis*, *correlation analysis*, and *trend analysis*.
    - We conducted *difference analysis* to compare the performance of NSGA-II and NSGA-III combined with PART and C4.5 with Random Search (RS) combined with PART and C4.5 in terms of fitness values, six commonly used quality indicators (i.e., *Hyper Volume* (*HV*), Inverted Generational Distance (*IGD*), Epsilon (ϵ), Euclidean Distance from the Ideal Solution (*ED*), Generational Distance (*GD*), and Generalized Spread (*GS*)), and 17 machine learning quality measurements (MLQMs), in comparison with the prior work where we evaluated NSGA-II and RS using fitness values, *HV*, and six MLQMs. Furthermore, we have also compared the performance of the four *SBRM*$^+$ approaches in terms of the 17 MLQMs to identify the best-suited approach for mining CPL rules.
    - We conducted *correlation analysis* to study the correlation between the quality of rules in terms of MLQMs and average fitness values and quality indicators, which was not performed in our prior work.
    - We conducted *trend analysis* to see the trend in the quality of rules based on MLQMs across different iterations of *SBRM*$^+$ (also not performed in the prior work).
- We also significantly extended the related work.

Evaluation results show that *SBRM*$^+$ is effective to produce high-quality rules as compared to RS based rule mining approach (i.e., *RBRM*$^+$), as in 7 out of 8 comparisons for the two case studies, the *SBRM*$^+$ approaches significantly outperformed the *RBRM*$^+$ approaches in terms of the majority of MLQMs. In eighth comparison, neither one of the two approaches dominate other. Among the four *SBRM*$^+$ approaches, *SBRM*$^+_{NSGA\text{-}II}$*-C45* produced the highest quality rules based on MLQMs for the Cisco case study and *SBRM*$^+_{NSGA\text{-}II}$*-PART* for the Jitsi case study. *Correlation analysis* suggests that in most of the cases lower average fitness values, lower values of quality indicators (except for *HV*) and higher *HV* values mean overall higher quality rules in terms of MLQMs. Moreover, *trend analysis* shows an increasing trend of the quality of rules in terms of MLQMs for all the four *SBRM*$^+$ approaches across the five iterations.

The rest of the paper is organized as follows: Section 2 provides the background knowledge. In Section 3, we give an overview of *SBRM*$^+$ followed by the search-based approach for generating configurations in Section 4. In Section 5, we present experiment design and execution. In Section 4.3, we present results and analyses, followed by the overall discussion and threats to validity. Section 5 summarizes the literature review, and finally, in Section 8, we conclude the paper.

2 Background

In this section, we briefly introduce relevant knowledge on multi-objective search (Section 2.1), machine learning techniques for rule mining and clustering (Section 2.2), and branch distance calculation heuristic (Section 2.3).

2.1 Multi-objective Search

Multi-objective search has been widely applied to address different software engineering optimization problems such as test case prioritization, cost estimation, and configuration generation [60, 61, 76, 77]. Multi-objective search algorithms are designed to solve problems where various objectives are competing with each other, and no single optimal solution exists. They aim to find a set of non-dominated solutions for trading off different objectives.

To address our problem, we selected the most commonly used Non-dominated Sorting Genetic Algorithm (NSGA-II) [45, 78], which has proven to be effective for solving various software engineering problems such as test case prioritization and cost estimation [45, 47]. NSGA-II relies on the Pareto dominance theory, which yields a set of non-dominated solutions for multiple objectives [78]. At first, candidate solutions (i.e., the population) are sorted into various non-dominated fronts using a ranking algorithm. Then, individual solutions are selected from non-dominated fronts based on the crowd distance, which measures the distance between the individual solutions and the rest of the solutions in the population [79]. If two solutions belong to the same non-dominated front, then the solution with a higher crowd distance will be selected to increase the diversity of solutions.

We also selected a relatively new multi-objective algorithm NSGA-III [80, 81], which has shown to perform better than NSGA-II in several contexts [82]. The basic working procedure of NSGA-III is quite similar to the NSGA-II but with significant changes in its selection operator. Unlike NSGA-II, NSGA-III's selection process utilizes well-spread reference points to apply the selection pressure to maintain the diversity among population members. We use Random Search (RS) as the comparison baseline.

2.2 Machine Learning

Machine learning is typically used for classifying, clustering, and identifying/predicting patterns in data [83]. It has also been used for inferring rules [37, 70]. Machine learning techniques can be classified into supervised learning (i.e., for labeled data) and unsupervised learning (i.e., for unlabeled data). Supervised learning aims to find relations between input data and its outcome whereas unsupervised learning is for identifying hidden patterns inside input data without labeled responses. We adopted supervised learning in our approach, as we aim to find rules between product configurations (i.e., input) and system states indicating the states of products' interaction (i.e., outcome).

There are two major paradigms of rule generation: 1) creating rules from decision trees, converting the trees into rules and pruning them as opted by C4.5 [84]; 2) employing the separate-and-conquer rule learning technique used by Repeated Incremental Pruning to Produce Error Reduction [85]. Creating rules from decision trees is computationally expensive in the presence of noisy data, and the separate-and-conquer rule learning technique has the over pruning (hasty generalization) problem [59]. The Pruning Rule-Based Classification algorithm (PART) combines the two paradigms mentioned above of rule generation while avoiding their shortcomings. PART generates partial decision trees, and corresponding to each partial tree, a single rule is extracted for the branch that covers maximum nodes [59]. Therefore, in our previous study [75], we opted for PART. In this study, we also included C4.5, as it is the most popular algorithm in the research community as well as industry [86].

We used Lloyd's algorithm [87] for clustering rules, a commonly used *k*-means algorithm, which minimizes the average squared distance between points within the same cluster. Initially, it selects *k* data points randomly as centers of *k* clusters. Furthermore, it uses the Euclidean distance function [88] to calculate the distances between each data point and centers of *k* clusters, and assign each data point to its nearest cluster. After assigning all the data points to *k* clusters, it updates the centers of *k* clusters by calculating the mean of all the data points within each cluster. Once centers are updated, it recalculates the Euclidean distance for all the data points and reassigns them to *k* clusters. This process continues until the centers of *k* cluster do not change in two consecutive iterations.

2.3 Branch Distance Calculation Heuristic

Branch distance is a heuristic used in search-based software engineering, which indicates to what extent the given data satisfy the predicate (aka condition or clause) in the rule/constraint. For measuring the branch distance between a configuration of a configurable parameter and a predicate in the rule, we opted the branch distance calculation approach provided in [42, 43]. In Table D-1, we summarize the distance calculation formula corresponding to different operations for numerical and enumerated data.

Table D-1. Branch distance functions [42] *

Predicate type	Operation	Distance function
Predicates with relational operators	a=b	0
	a!=b	a!=b → 0 **else** nor(|a−b| +1) *k
Predicate with a Boolean condition		True → 0 **else** k
Logical connective of two predicates	$Pr_1 \wedge Pr_2$	$Pr_1 + Pr_2$ (sum of branch distances for both predicates)

* *k* is a positive constant greater than zero, we used *k*=1; *nor* gives a normalized value between zero and one.

3 Overview

Figure D-2 presents an overview of *SBRM*$^+$, which relies on machine learning and multi-objective search to mine CPL rules in an iterative and incremental process. As shown in Figure D-2, the whole process consists of seven steps, which are organized into four types of activities: *Generation*, *Execution*, *Mining*, and *Clustering*. *Generation* related steps (i.e., Steps 1 and 5) about generating configurations for the selected products within/across product lines, using a search algorithm (e.g., NSGA-II) or RS. *Execution*-related steps (i.e., Steps 2 and 6) configure the selected products with generated configurations and obtaining their consequent system states to label the configurations. *Mining*-related steps (i.e., Steps 3 and 7) combine all generated configurations with system states and apply machine learning algorithms (e.g., PART) to mine rules. *Clustering* Step 4 clusters and classifies the mined rules into categories with a clustering algorithm (e.g., *k*-means).

Based on the categories, we defined three search objectives for guiding the search for generating configurations (Step 5).

As shown in Figure D-2, in the first step, an initial set of configurations is randomly generated for configurable parameters of the selected products within/across product lines. The second step obtains the system states indicating the states of interaction among selected products. During the second step, we configure the selected products with randomly generated configurations, execute certain functionalities to enable the communication among the selected products, and capture the system states to know if the products communicated successfully (as intended). In our context, an *interaction* can be defined as a communication between two or more products communicating via a communication medium. An interaction can be enabled by executing certain functionalities (i.e., a sequence of operations) of the communicating products to make them communicate with each other.

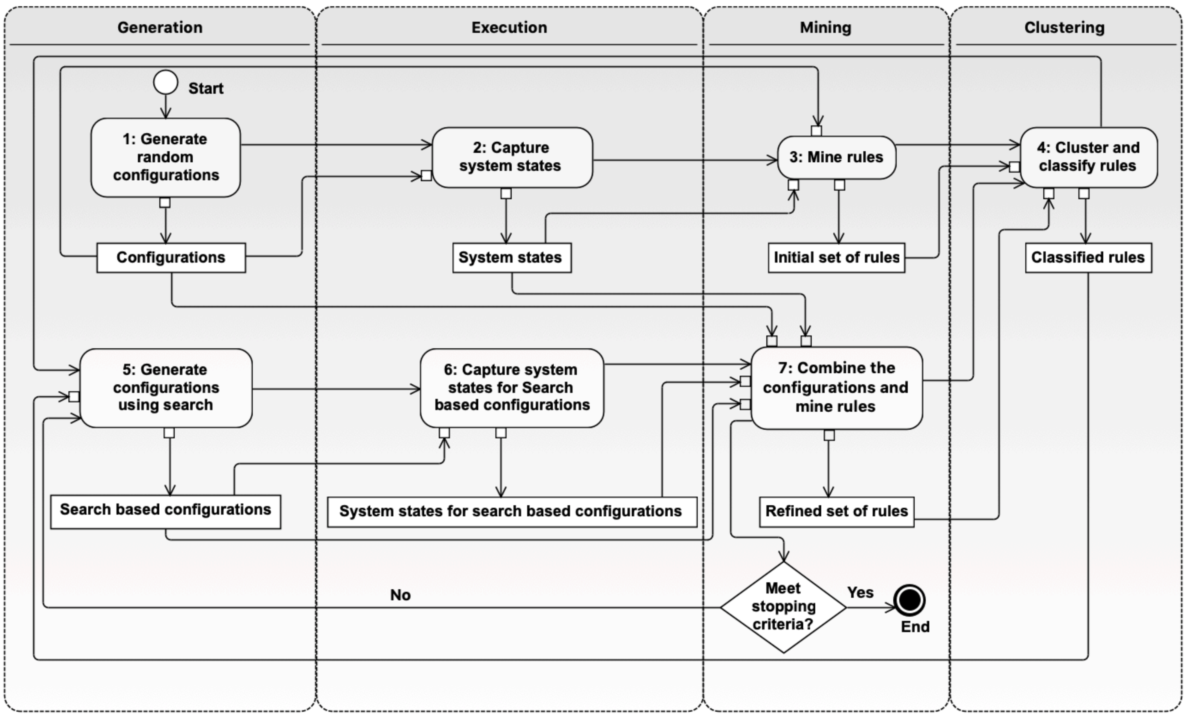

Generation
Execution
Mining
Clustering
Start
1: Generate random configurations
2: Capture system states
3: Mine rules
4: Cluster and classify rules
Configurations
System states
Initial set of rules
Classified rules
5: Generate configurations using search
6: Capture system states for Search based configurations
7: Combine the configurations and mine rules
Search based configurations
System states for search based configurations
Refined set of rules
No
Meet stopping criteria?
Yes
End

Figure D-2. Overview of the proposed approach (SBRM$^+$)

In step 3, we feed the set of generated configurations (as Attributes) and their corresponding system states (as Classes) to Weka [74] as the initial input and apply a rule mining algorithm (e.g., PART or C4.5) to mine the initial set of rules. Normally, a classifier (e.g., C4.5 and PART) trains a model using a training dataset and then validates the model using a test dataset. In our case, the input configurations are used as the training dataset. For validation, we used 10 times 10-fold stratified cross-validation as it presents all classes (approximately) equally across each test fold [51, 86]. This means both PART and C4.5 use 10% of the training data (i.e., generated configurations with corresponding system states provided as input) in each test fold to validate the model. Note, PART gives a set of rules as the outcome. However, C4.5 gives a decision tree, where a non-leaf node in a branch represents a predicate specifying the configuration value for a particular configurable parameter and the leaf node represents the predicted Class (e.g., the call status *ConnectedConnected* in our context). From each branch of a generated tree, we extract a rule by joining all non-leaf nodes in the branch with the AND operator to form the antecedent of the rule and using the leaf node as its consequent. We provide the code for extracting the rules from the tree in the Bitbucket repository[16].

In step 4, the mined rules are clustered using the *k*-means clustering algorithm (Section 2.2) and classified into different categories. The classified rules are fed to NSGA-II or NSGA-III for generating configurations for the next iteration in step 5. In step 6, we repeat step 2 but take the configurations generated from the search instead of the random one. In step 7, we combine all the configurations generated from steps 1 and 5 and collected system states captured from steps 2 and 6, and feed all the data to Weka to mine a refined set of rules. This rule set is then used in the next iteration (starting from step 4) to generate more configurations and further refine the rules.

In each iteration, newly generated configurations with collected system states are added to the dataset from the previous iteration to mine a new set of rules. We repeat the process (step 4 to step 7) until we meet the stopping criteria, e.g., a fixed number of iterations and/or when the rules mined from two consecutive iterations are similar. We used a fixed number of iterations in our experiments, as we have limited available resources for mining rules. Getting similar rules from consecutive iterations indicates that it is very unlikely to refine the rules further. All the iterations (e.g., five iterations in our experiment) used for refining the rules and repeated before meeting the stopping criteria make a complete cycle. We consider step 4 (i.e., classification of rules) and step 5 (i.e., using search to generate configurations), as the innovative part of the whole approach, i.e., *SBRM*$^+$. This is because using Weka to mine rules is an application of the rule mining algorithm (e.g., PART or C4.5), but applying search requires carefully designing a fitness function. Similarly, classification of rules requires applying the *k*-means clustering algorithm based on certain attributes, ranking the clusters using specific formula and classify the rules into different categories, which are consequently fed to the search algorithm (e.g., NSGA-III) to generate configurations. Both steps 4 and 5 are discussed in detail in the following section.

Pseudocode 1 is the pseudocode of SBRM$^+$, where in L1, we encode the configuration generation problem by representing all the configurable parameters as numerical variables (Integer or Real) and restricting their domains by defining their upper and lower limits. In L2-L5 (i.e., Zero-Iteration), we generate the initial set of configurations randomly, decode them, and mine the initial set of rules. Similarly, in L6-L19, we cluster and classify the rules (L7), generate configurations (L8) using the search (e.g., NSGA-II), decode the configurations (L9), and mine the refined set of rules (L10-L11). Note, encoding and decoding are discussed in detail in Section 4.3, whereas the mining and clustering are introduced in Section 4.2.

```
Input: A set of $n$ configurable parameters $CP = \{cp_1, cp_2, .., cp_n\}$ with their sets of possible values $CPV = \{CPV_1, CPV_2, .., CPV_n\}$, Number of intial randomly generated configurations $NC_{RG}$, Number of iterations $NI$, and Number of configurations to be generated per iteration $NC_{PI}$, a set of parameters for search algorithm $P_{SA}$
Output: A set of rules $RRS$
Begin
            # …………………. Encoding Configuration Generation Problem …………..…….
      L1. $ECP$, $UpperLimits$, $LowerLimits$ ← Encode_Configurations_Generation_Problem ($CP$, $CPV$)

        # ……….. Generating Initial Set of Rules based on Randomly Generated Configurations…….
      L2. $I := 0$              # Zero-Iteration where we use configurations generated randomly
```

[16] https://bitbucket.org/safdaraqeel/ase-ruleextraction

```
L3. $EC_{RG}$ ← Generate_Configurations_Randomly ($ECP$, $UpperLimits$, $LowerLimits$, $NC_{RG}$)
L4. $DC_{RG}$ ← Decode_Configurations ($EC_{RG}$, $UpperLimits$, $LowerLimits$)
L5. $RS_{Initial}$ ← Mine_Initial_RuleSet ($DC_{RG}$)

# … Generating Refined Set of Rules Based on Configurations Generated Using Search Algorithm…
L6. $I := I + 1$          # First iteration where we use configurations generated by search algorithms
L7. $CRS_{Initial}$ ← Cluster_And_Catergorize_Rules($RS_{Initial}$)
L8. $EC_{SBG}$ ← Generate_Configurations_Using_Search ($ECP$, $UpperLimits$, $LowerLimits$, $NC_{PI}$, $CRS_{Initial}$,
    $P_{SA}$)
L9. $DC_{SBG}$ ← Decode_Configurations ($EC_{SBG}$, $UpperLimits$, $LowerLimits$)
L10. $DC_{RG+SBG}$ ← Combine_Configurations ($DC_{RG}$, $DC_{SBG}$)
L11. $RS_{Refined}$ ← Mine_Refine_RuleSet ($DC_{SBG+RG}$)

L12. while ($I \leq NI$) do
L13.     $I := I + 1$
L14.     $CRS_{Refined}$ ← Cluster_And_Classify_Rules($RS_{Refined}$)
L15.     $EC_{SBG}$ ← Generate_Configurations_Using_Search ( $ECP$ , $UpperLimits$ , $LowerLimits$ , $NC_{PI}$ ,
    $CRS_{Refined}$, $P_{SA}$)
L16.     $DC_{SBG}$ ← Decode_Configurations ($EC_{SBG}$, $UpperLimits$, $LowerLimits$)
L17.     $DC_{RG+SBG}$ ← Combine_Configurations ($DC_{RG+SBG}$, $DC_{SBG}$)
L18.     $RS_{Refined}$ ← Mine_Refine_RuleSet ($DC_{RG+SBG}$)

L19. return $RS_{Refined}$
```

Pseudocode 1: Search Based Rule Mining (SBRM^{+})

4 Search-Based Configuration Generation Approach

Sections 3.1 presents formal definitions required to define the configuration generation problem. In Section 3.2, we present details about the classification of CPL rules. Section 4.3 presents the objectives and effectiveness measures, followed by the fitness function in Section 3.3.

4.1 Formalization of Configuration Generation Problem

We formalize relevant concepts and exemplify them with an example of three communicating VCS products belonging to two different product lines (Figure D-4). The definitions, formal representations, and examples of the concepts related to the product lines and rule mining are presented in Table D-2. Moreover, we also constructed a class diagram shown in Figure D-3 to conceptually describe how the defined concepts are related to each other.

As shown in Figure D-3, each product line has two or more products, which are communicating via a communication medium (e.g., Wired Internet, Wireless Internet, Bluetooth). A product has one or more configurable parameters, state variables, and operations. Each configurable parameter has two or more configurable parameter values. Similarly, each state variable has two or more state values. An operation can take zero or more operation parameters as input, where each operation parameter has two or more operation parameter values. The operation parameter values assigned to the operation parameters of the operation may affect the behavior of the operation. Different products can communicate with each other by enabling a particular interaction. Enabling a particular interaction requires executing a sequence of operations belonging to one or more communicating products. State rules defined on state variables of the communicating products can be used to define the system states, which indicate whether products interact/communicate successfully (as intended). The configurable parameter values assigned to the configurable parameters of the communicating products determine the success of the interaction. Moreover, the communication medium may also influence the interaction. An interaction has at least one source product and one or more target products. An interaction is homogeneous if the communicating products belong to the same product line otherwise heterogeneous. The communication between products enabled by an interaction can be unidirectional or bidirectional.

In Figure D-4**,** *VCS-PL1* and *VCS-PL2* are two VCS product lines. *VCS1*, *VCS2*, and *VCS3* are three products communicating through *WiredInternet*, where *VCS1* and *VCS2* belong to *VCS-PL1*, and *VCS3* belongs to *VCS-PL2*. *VCS1* has three configurable parameters (e.g., *VCS1.defaultProtocol*), three state variables (e.g., *VCS1.callStatus*), and five operations (e.g., *VCS1.dial()*). Similarly, *VCS2* has three configurable parameters, one state variable, and three operations whereas *VCS3* has four configurable parameters, one state variables, and three operations. *dial()* operation of all three VCS products has three operation parameters including *protocol*, *callRate*, and *callType*.

Table D-2. Formalization of concepts*

Def#	Concept	Definition and formal representation with examples
1	Product line	A product line can be defined as a set of products sharing explicitly defined and managed common and variable features and relying on the same domain architecture. A set of npl product lines for a particular application domain can be presented as: $PL = \{pl_1, pl_2, .., pl_{npl}\}$, where pl_i represents the i^{th} product line. For example, {*VCS-PL1*, *VCS-PL2*} is a set of two product lines.
2	Product	A product can be defined as a triplet (CP, SV, OP), where CP, SV, and OP are sets of configurable parameters, state variables, and operations. A set of inp products for a product line pl_i can be presented as: $P_i = \{p_{i1}, p_{i2}, .., p_{inp}\}$, where p_{ij}represents the j^{th} product of pl_i. For example, {*VCS1, VCS2*} represent a set of products for *VCS-PL1*.
3	Configurable parameter	A configurable parameter is a numerical (e.g., integer, real) or non-numerical (e.g., binary, ordinal, nominal) type variable, which can take different values [33]. The possible values of a numerical type configurable parameter can be specified by defining the constraints on its upper and lower limits whereas, for a non-numerical type configurable parameter, they can be specified as a set of predefined values. A set of $incp$ configurable parameters for a product p_i can be presented as: $CP_i = \{cp_{i1}, cp_{i2}, .., cp_{incp}\}$, where cp_{ij} represents the j^{th} configurable parameter of p_i. For example, {*defaultProtocol*, *encryption*, *defaultCallRate*} is a set of configurable parameters of product *VCS1*, where *defaultProtocol* and *encryption* are non-numerical and *defaultCallRate* a numerical type configurable parameters.
4	Configurable parameter value	A configurable parameter value is a value that can be assigned to a configurable parameter. A set of $incpv$ configurable parameter values for each cp_i can be presented as: $CPV_i = \{cpv_{i1}, cpv_{i2}, .., cpv_{incpv}\}$, where cpv_{ij} represents the j^{th} value of cp_i. For a real type configurable parameter, $incpv$ will be infinity because such configurable parameter can take infinite values between its upper and lower limits. For example, *defaultProtocol* can be configured with *SIP*, *H323*, and *AIM* whereas *defaultCallRate* can take a value between 64 to 6000.
5	State variable	State variables are variables used to describe the state of the system. A set of $insv$ state variables for a product p_i can be presented as $SV_i = \{sv_{i1}, sv_{i2}, .., sv_{insv}\}$, where sv_{ij} represents the j^{th} state variable for the product p_i. For example, {*callStatus*, *numberOfActiveCalls*, *isPresentationShared*} is a set of state variables of *VCS1*.
6	State value	A state value is a possible value of a state variable, specific to one product. A set of inv possible state values for a state variable sv_i, can be presented as $V_i = \{v_{i1}, v_{i2}, .., v_{inv}\}$, where v_{ij} represents the j^{th} state value of sv_i. For example, {*Connected*, *Failed*} represents a set of state values for *callStatus* of *VCS1*.
7	Operation	An operation is a function implemented in a product. A set of $inop$ operations for a product p_i can be presented as: $OP_i = \{op_{i1}, op_{i2}, .., op_{inop}\}$, where op_{ij} represents

		the j^th^ operation of p_i. For example, {*dial()*, *accept()*, *disconnect()*, *startPresentation()*, *stopPresentation()*} is a set of operations for *VCS1*.
8	Operation parameter	An operation parameter is a variable of numerical (e.g., integer, real) or non-numerical (e.g., binary, ordinal, nominal) type, provided as input to an operation. A set of $inpm$ parameters for an operation op_i can be presented as $PM_i = \{pm_{i1}, pm_{i2}, .., pm_{inpm}\}$, where pm_{ij} represents the j^th^ operation parameter for the operation op_i. For example, {*protocol*, *callRate*, *callType*} represents a set of operation parameters for *dial()* operation of *VCS1*.
9	Operation parameter value	An operation parameter value is a value that can be assigned to an operation parameter. A set of $inpv$ parameter values for an operation parameter pm_i can be presented as $PV_i = \{pv_{i1}, pv_{i2}, .., pv_{inpv}\}$, where pv_{ij} represents the j^th^ operation parameter value of pm_i. For example, {*Audio*, *Video*} represents a set of operation parameter values for *callType* operation parameter corresponding to the *dial()* operation of *VCS1*.
10	Interaction	An interaction is communication between at least one source product and one or more target products communicating via a communication medium, enabled by a sequence of operations belonging to the source and target products. A set of $inin$ interactions supported by a product p_i to communicate with other products can be presented as: $IN_i = \{in_{i1}, in_{i2}, .., in_{inin}\}$, where in_{ij} represents the j^th^ interaction supported by product p_i. For example, {*making-call*, *sharing-presentation*} represents a set of interactions supported by *VCS1*.
11	Selected products	A set of nsp communicating products under study can be presented as: $SP = \{p_1, p_2, .., p_{nsp}\}$, where p_i represents the i^th^ product in the set of communicating products. Such products may belong to different product lines or same product line. For example, SP = {*VCS1*, *VCS2*, *VCS3*} represented a set of selected products where *VCS1* and *VCS2* belong to *VCS-PL1*, and *VCS3* belongs to *VCS-PL2*.
12	Selected interactions	A set of $nsin$ selected interactions for the selected products can be defined as: $SIN_{SP} = \{in_1, in_2, .., in_{nsin}\}$, where in_j represents the j^th^ selected interaction. For example, SIN_{SP} = {*making-call*} represents the set of selected interactions.
13	Selected operations	For each selected interaction in_i, a sequence of operations required to enable interaction in_i can be defined as: $OPS_i = (op_{i1}, op_{i2}, .., op_{insop})$, where op_{ij} represents the j^th^ operation (in order) required to enable interaction in_i. For example, (*VCS1.dial()*, *VCS2.accept()*, *VCS3.accept()*, *VCS1.disconnect()*) represents the sequence of operations required to enable *making-call* interaction.
14	Selected configurable parameters	A set of $nscp$ selected configurable parameters for all the selected products can be defined as: $SCP_{SP} = \{cp_1, cp_2, .., cp_{nscp}\}$, where cp_i represents the i^th^ configurable parameter of the selected product. For example, SCP_{SP}={*VCS1.defaultProtocol*, *VCS1.defaultCallRate*, *VCS1.encryption*, *VCS2.encryption*, *VCS3.encryption*} represents the set of selected configurable parameters for the selected products.
15	Selected state variables	A set of $nssv$ selected state variables for all the selected products related to the selected interaction can be defined as $SSV_{SP} = \{sv_1, sv_2, .., sv_{nssv}\}$, where vs_i represents the i^th^ state variable. The selected state variables may belong to different products. For example, {*VCS2.callStatus*, *VCS3.callStatus*} represents the set of selected state variables.
16	System states	System states are the combinatorial states for all the products involved in the selected interactions, which indicate whether products communicated successfully (as intended). Such states are described by defining the state rules on the selected state variables. A set of nis possible system states corresponding to the selected interactions can be defined as: $SS = \{ss_1, ss_2, .., ss_{nis}\}$, where ss_i represents the j^th^ system state. For example, {*ConnectedConnected*, *ConnectedFailed*, *FailedConnected*, *FailedFailed*} represents a set of system states, which are specified by concatenating (state rule) the states values of the selected state variables.
17	Predicate	A predicate is a conditional statement in a rule with one configurable parameter and its value joined by one of the relational operators (i.e., $=, \neq, <, \leq, >, \geq$). For example, "*VCS1.encryption* = *On*" and "*VCS1. defaultCallRate* > 1000" are two predicates.
18	Rule	In the context of rule mining, a rule with npr predicates can be represented as: $r_i = pr_1\ AND\ pr_2\ AND\ ...\ AND\ pr_{npr} : ss_k$, where pr_j represents the j^th^ predicate of rule r_i and ss_k represents k^th^ system state. For example, r_1: "*VCS1.encryption* = *On* AND *VCS2.encryption* = *Off* AND *VCS3.encryption* = BestEffort: *ConnectedFailed*" and r_2:

		"*VCS1.encryption = On* AND *VCS2.encryption = BestEffort* AND *VCS3.encryption = On: ConnectedConnected*" are two rules.
19	Confidence of a rule	For a rule r_i, $Cf(r_i)$ represents the *confidence* of r_i, which is between 0 and 1. *Confidence* for a rule r_i can be calculated as: $Cf(r_i) = \frac{(SP_i - V_i)}{(SP_i + V_i)}$, where SP_i represents the number of instances for which r_i holds true (i.e., *support*) and V_i represents the number of instances that violate r_i (i.e., *violation*). An instance represents a set of configurable parameter values for the selected configurable parameters of the communicating products and corresponding system state.
20	Configuration solution	A configuration solution $\{s_j\}$ is a set of configurable parameter values assigned to all the selected configurable parameters, which mathematically can be represented as: $s_j = \{cpv_{j1}, .., cpv_{jnscp}\}$, where cpv_{ji} represents the configurable parameter value assigned to the *i*th configurable parameter (i.e., cp_i) in $\{s_j\}$. For example, {*SIP*, 5000, *On*, *BestEffort*, *Off*} is a configuration solution.
21	Configuration space	A set of ns potential configuration solutions (i.e., configuration space) can be defined as: $S = \{\{s_1\}, \{s_2\}, .., \{s_{ns}\}\}$, where $\{s_i\}$ represents the *i*th configuration solution. ns can be calculated as the cardinality of the Cartesian product of configurable parameter values' sets for all the selected configurable parameters, which can be represented mathematically as: $\lvert CPV_1 \times .. \times CPV_{nscp} \rvert$. The configuration space for the Cisco's case study contains approximately 1.03e33 configuration solutions and 6.54e60 for the Jitsi case study
22	Effectiveness measures	A set of ne effectiveness measures can be defined as: $E = \{e_1, e_2, .., e_{ne}\}$, where e_i represents the *i*th effectiveness measure. For example, a set of three effectiveness measures (i.e., AHNS, NLNS, and NAS) defined in Section 4.3
23	Explored solutions	A set of nes configuration solutions explored during the search is a proper subset of S, which mathematically can be represented as: $S_{Ex} = \{\{s_1\}, \{s_2\}, .., \{s_{nes}\}\}$, where $nes < ns$.

* Note: All the examples provided are based on the running example. Also, by selected elements (e.g., products, configurable parameters), we mean elements under study for learning CPL rules.

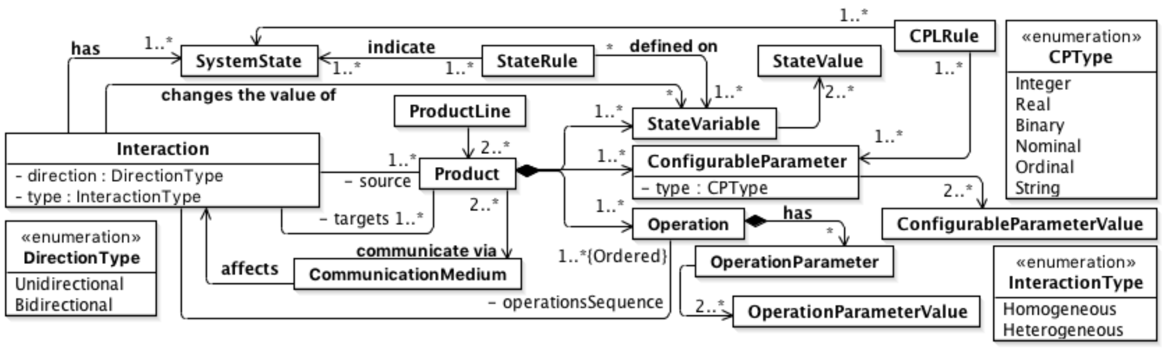


Figure D-3. A conceptual model for interacting products

Based on the concepts presented in Table D-2, our configuration generation problem can be formulated as searching a solution set S_R from a set of explored solutions (i.e., S_{EX}) such that $S_R \subset S_{Ex}$, and all the solutions in S_R have highest effectiveness in terms of effectiveness measures E than all the other explored solutions in $\{S_R \backslash S_{Ex}\}$.

$$\forall_{s_r \in S_R} \forall_{s_i \in S_{Ex}} \forall_{e_j \in \mathrm{E}}\ s_i \notin S_R \wedge Effect(s_r, e_j) \geq Effect\ (s_i, e_j)$$

$$\wedge \exists_{e_k \in \mathrm{E}}\ Effect(s_r, e_k) > Effect\ (s_i, e_k) \qquad (1)$$

where $Effect\ (s_i, e_j)$ gives the value of the *j*th effectiveness measure (Section 4.4) for configuration solution s_i.

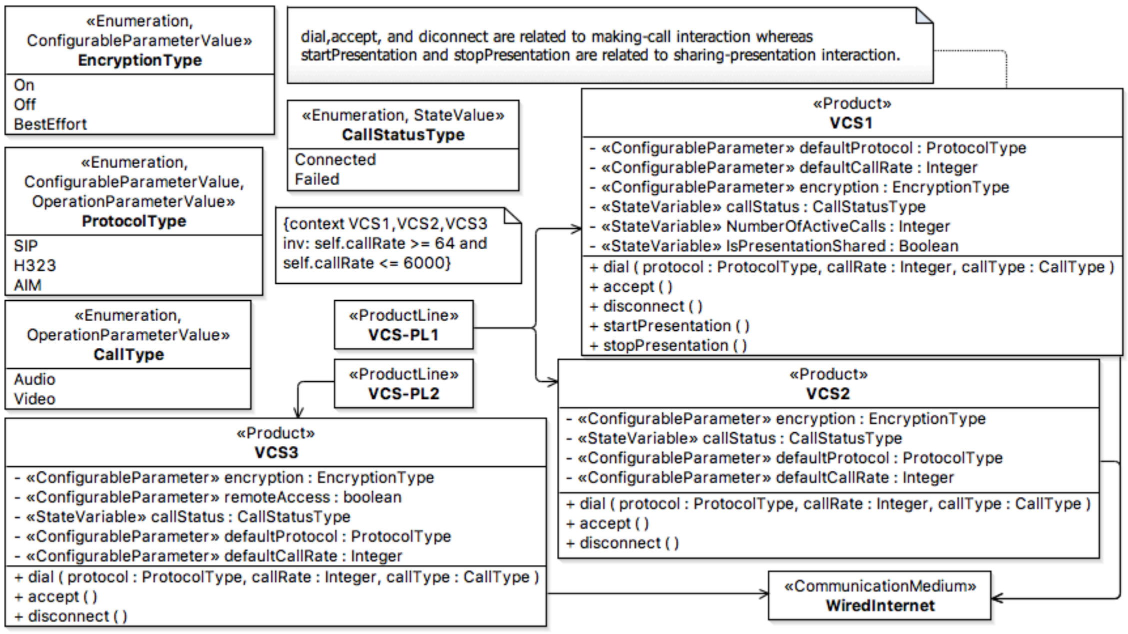

«Enumeration, ConfigurableParameterValue»
EncryptionType
On
Off
BestEffort
dial,accept, and diconnect are related to making-call interaction whereas startPresentation and stopPresentation are related to sharing-presentation interaction.
«Enumeration, StateValue»
CallStatusType
Connected
Failed
«Product»
VCS1
- «ConfigurableParameter» defaultProtocol : ProtocolType
- «ConfigurableParameter» defaultCallRate : Integer
- «ConfigurableParameter» encryption : EncryptionType
- «StateVariable» callStatus : CallStatusType
- «StateVariable» NumberOfActiveCalls : Integer
- «StateVariable» IsPresentationShared : Boolean
+ dial (protocol : ProtocolType, callRate : Integer, callType : CallType)
+ accept ( )
+ disconnect ( )
+ startPresentation ( )
+ stopPresentation ( )
«Enumeration, ConfigurableParameterValue, OperationParameterValue»
ProtocolType
SIP
H323
AIM
{context VCS1,VCS2,VCS3 inv: self.callRate >= 64 and self.callRate <= 6000}
«Enumeration, OperationParameterValue»
CallType
Audio
Video
«ProductLine»
VCS-PL1
«ProductLine»
VCS-PL2
«Product»
VCS2
- «ConfigurableParameter» encryption : EncryptionType
- «StateVariable» callStatus : CallStatusType
- «ConfigurableParameter» defaultProtocol : ProtocolType
- «ConfigurableParameter» defaultCallRate : Integer
+ dial (protocol : ProtocolType, callRate : Integer, callType : CallType)
+ accept ( )
+ disconnect ( )
«Product»
VCS3
- «ConfigurableParameter» encryption : EncryptionType
- «ConfigurableParameter» remoteAccess : boolean
- «StateVariable» callStatus : CallStatusType
- «ConfigurableParameter» defaultProtocol : ProtocolType
- «ConfigurableParameter» defaultCallRate : Integer
+ dial (protocol : ProtocolType, callRate : Integer, callType : CallType)
+ accept ( )
+ disconnect ( )
«CommunicationMedium»
WiredInternet

Figure D-4. Exemplifying concepts related to the product interaction

4.2 Clustering and Classification of CPL Rules

Generally, from the user perspective, the system states (Table D-2) can be categorized as normal states and abnormal states. Normal states indicate that interaction was enabled successfully and selected products interacted/communicated successfully as intended whereas abnormal states show that interaction failed and selected products did not interact/communicate successfully. Consequently, CPL rules can be classified into two categories: $R_A = \{r_{a1}, r_{a2}, r_{a3}, \dots, r_{nar}\}$ for abnormal states (Category-I), where r_{ai} represents the *i*th rule with abnormal state and nar represents the total number of rules with abnormal states; $R_N = \{r_{n1}, r_{n2}, r_{n3}, \dots, r_{nnr}\}$ is for normal states, where r_{ni} represents the *i*th rule with normal state and nnr represents the total number of rules with normal states.

We apply *k*-means (Section 2.2) to cluster R_N into three clusters based on three attributes of rules: *confidence*, *support*, and *violation*. *Support* and *violation* have a different scale than *confidence*, and generally, clustering algorithm does not work with attributes of different scales [89]. Thus, we divided *support* and *violation* by the sum of maximum support and maximum violation in order to normalize *support* and *violation*. After clustering the rules, we calculate the rank for each cluster as:

$$Rank\ (c_i) = (Support(c_i) + Confidence(c_i) - Violation(c_i)) \qquad (2)$$

Where $Support(c_i)$, $Violation(c_i)$, and $Confidence(c_i)$ are mean values of normalized *support*, *violation*, and *confidence* for all the rules belonging to cluster c_i. Based on the calculated ranks of the three clusters, all the rules are classified into two categories: R_{N1} represents high-confidence rules belonging to a cluster with the highest rank (Category-II) whereas R_{N2} represents low-confidence rules belonging to other two clusters with the lowest and medium ranks (Category-III). In Table D-2 (Def# 18), we present two CPL rules r_1 and r_2 where r_1 is a rule with an abnormal state *ConnectedFailed* and r_2 is a rule with a normal state *ConnectedConnected*. For example, r_1 describes that if the encryption of *VCS1* (i.e., *Caller*) is set to be "On", encryption of *VCS2* (i.e., *Callee1*) is set to be "Off", and encryption of *VCS3* (i.e., *Callee2*) is set to be "BestEffort", the conference call will connect to the *VCS2* but will fail to connect with the *VCS3*. Rule r_1 is an abnormal state rule, as the consequent of r_1 (i.e., *ConnectedFailed*) is an abnormal system state. Similarly, r_2 is a normal state rule because its consequent (i.e., *ConnectedConnected*) is a normal system state.

4.3 Solution Encoding and Decoding

As mentioned in Table D-2 (Def#3), a configurable parameter can be a numerical (e.g., Integer) or non-numerical (e.g., Boolean, Nominal) type variable. Thus, to apply search algorithms for the configuration generation problem, we encode all the configurable parameters as a vector of integer variables to represent the configuration solutions. Considering three configurable parameters *encryption* (i.e., Nominal), *remoteAccess* (i.e., Boolean), and *callRate* (i.e., Integer) of three communicating products *VCS1*, *VCS2*, and *VCS3* in Figure D-5, *encryption* can take one of the three values (*On*, *Off*, and *BestEffort*) and *remoteAccess* can take *True* or *False* whereas *callRate* can take a value from 64 to 6000.

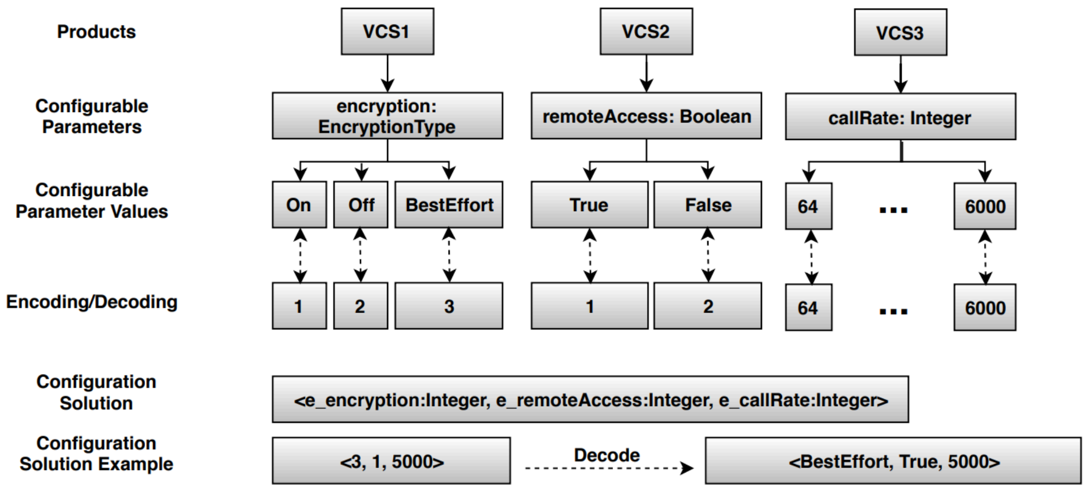


Figure D-5. Exemplifying the encoding and decoding mechanism employed in SBRM⁺

To encode the non-numerical configurable parameters, we map all the configurable parameter values to a sequence of numbers (Figure D-5). For example, we mapped *On*, *Off*, and *BestEffort* to *1*, *2*, and *3* respectively in order to encode *encryption*. The configuration solution is represented as a vector of integer variables (i.e., *e_encryption*, *e_remoteAccess*, and *e_callRate*) where each variable represents a particular configurable parameter. For example, *e_encryption* represents *encryption* in Figure D-5. To decode a particular configuration solution, we replace the integer values in the vector with the configurable parameter values of corresponding configurable parameters. For example, in Figure D-5, we replace values *3* and *1* with *BestEffort* and *True* to get the final decoded configuration solution: <*BestEffort*, *True*, *5000*>.

4.4 Objectives and Effectiveness Measures

CPL rules could reveal invalid configurations that lead to unwanted states of the system (i.e., abnormal states) are more important, therefore, the invalid configurations are of more interest. This encouraged us to use the search to generate configurations in a smart way. To be more specific, by applying search heuristics, we embrace configurations under which communicating products may fail to interact/communicate with each other and avoid configurations that lead to successful interactions among products. To achieve this goal, we define three objectives based on the distances between a configuration solution and the three categories of rules (Category-I, Category-II, Category-III). Before presenting the objectives and effectiveness measures, we first define the distance function that is used to assess the effectiveness measures. The distance function indicates to what extent a configuration solution conforms to a rule.

$$D(r_i, s_r) = \frac{\sum_{j=1}^{npr} d(pr_j, cpv_r)}{mnp} \qquad (3)$$

where $D(r_i, s_r)$ calculates the distance between rule r_i and configuration solution s_r. In equation (3), $d(pr_j,\ cpv_r)$ calculates the branch distance between j^{th} predicate pr_j from rule r_i and corresponding configurable parameter value cpv_r of the configurable parameter involved in predicate pr_j from configuration solution s_r. npr represents the total number of predicates in rule r_i whereas mnp represents the number of predicates in a rule with the maximum number of predicates. To calculate the distance between pr_j and cpv_r as a branch distance, we use the distance calculation formula provided in [42] (Section 2.3).

Objective-1: This objective is to avoid generating configurations that completely or close to satisfy rules in Category-II. The effectiveness measure $AHNS$ corresponding to this objective can be calculated as:

$$AHNS(R_N, s_r) = \sum_{i=1}^{nnr} Cf(r_i) * D(r_i, s_r) \mid r_i \in R_{N1} \qquad (4)$$

where $AHNS(R_N, s_r)$ takes R_N (the set of rules related to the normal states) and one configuration solution s_r as input and gives the effectiveness measure as output. To determine $AHNS$, we calculate the sum of weighted distances for all the rules in Category-II (i.e., R_{N1}), where each rule belongs to the cluster with the highest rank. The weighted distance of r_i is calculated by multiplying $Cf(r_i)$ with $D(r_i, s_r)$.

Objective-2: This objective is to generate configurations within the configuration space that satisfy Category-III (i.e., R_{N2}) as well as its nearby space. The nearby space contains configurations for which the distance to the rules in Category-III is close to 0 but not exactly 0. These configurations might help to either improve the confidence of correct rules by increasing their support or filter out incorrect ones by increasing their violation and hence reducing their confidence. The effectiveness measure $NLNS$ related to the second objective can be calculated as:

$$NLNS(R_N, s_r) = \sum_{i=1}^{nnr} Cf(r_i) * (1 - D(r_i, s_r)) \mid r_i \in R_{N2} \qquad (5)$$

where $NLNS(R_N, s_r)$ takes R_N (the set of rules associated with the normal states) and configuration solution s_r as input and outputs $NLNS$. Since we want to explore the configuration space near the configurations satisfying the rules in Category-III, configurations with a smaller distance to the rules in Category-III are preferred. Therefore, we use $(1 - D(r_i, s_r))$ in the $NLNS(R_N, s_r)$. To calculate $NLNS$, we calculate the sum of the weighted distance (i.e., calculated by multiplying $Cf(r_i)$ with $(1 - D(r_i, s_r))$*)* of a configuration solution to all the rules in Category-III (i.e., R_{N2}), where each rule belongs to a cluster with middle rank or lowest rank.

Objective-3: This objective is to generate configurations within the configuration space that satisfy Category-I and its nearby space. The rules in Category-I are of high interest in our context because they indicate situations where interactions of the selected products fail. The effectiveness measure NAS for this objective can be calculated as:

$$NAS(R_A, s_r) = \sum_{i=1}^{nar} Cf(r_i) * \left(1 - D(r_i, s_r)\right) \quad (6)$$

where $NAS(R_A, s_r)$ takes rule set R_A (related to the abnormal states) and configuration solution s_r as input. To calculate NAS, we calculate the sum of weighted distances for all the rules in R_A (Category-I).

4.5 Fitness Function

We first normalize the three effectiveness measures using the simple yet robust unity-based normalization function $nor\big(F(x)\big) = \left(\frac{F(x)-F_{min}}{F_{max}-F_{min}}\right)$ [90, 91], where $F(x)$ is an effectiveness measure function, F_{max} and F_{min} are the maximum and minimum values of the effectiveness measure. For $AHNS$, F_{min} is 0 when the distance between all the rules in Category-II and configuration solution s_r is 0. F_{max} can be calculated as $\sum_{i=1}^{nnr} Cf(r_i)$ where the distance between all the rules in Category-II and configuration solution s_r is 1. For $NLNS$ and NAS, F_{min} is 0 when the distance between all the rules in the corresponding category and configuration solution s_r is 1. Corresponding to $NLNS$ and NAS, F_{max} can be calculated as $\sum_{i=1}^{nnr} Cf(r_i)$ and $\sum_{i=1}^{nar} Cf(r_i)$ respectively, where the distance between all the rules and configuration solution s_r is 0.

With the three effectiveness measures, we define the fitness function based on the three objectives as follow:

$$F(O_1) = 1 - Nor\ (AHNS(R_N, s_r)) \quad (7)$$
$$F(O_2) = 1 - Nor\ (NLNS(R_N, s_r)) \quad (8)$$
$$F(O_3) = 1 - Nor\ (NAS(R_A, s_r) \quad (9)$$

Note that, in the above equations, we define our search problem as a minimization problem by subtracting each normalized effectiveness measure from 1 to ensure that a configuration solution with a value closer to 0 is better.

The fitness function with the three objectives is combined with NSGA-II and NSGA-III to address the configuration generation optimization problem. We implemented our problem in jMetal by encoding all the configurable parameters in the configuration solution s_r as integer variables (Section 4.3). Besides the possible values for all the variables that are specified by constraining their upper and lower limits, there are no additional constraints. Initially, all the variables in s_r are initialized with random values between their upper and lower limits. During the search, *SBRM*$^{+}$ generates optimized solutions guided by the fitness function. The jMetal based implementation of our configuration generation problems for both of the case studies are provided in the Bitbucket repositories[17].

[17] https://bitbucket.org/safdaraqeel/sbrm-jitsi/, https://bitbucket.org/safdaraqeel/sbrm-cisco

5 Evaluation

The overall objective of the evaluation is to assess the effectiveness of combining two different machine learning algorithms (i.e., PART and C4.5) with NSGA-II and NSGA-III to mine CPL rules. In Section 4.1, we present experiment design, followed by the experiment execution (Section 4.2).

5.1 Experiment Design

We present research questions in Section 4.1.1, the two case studies in Section 4.1.2, evaluation metrics in Section 4.1.3, evaluation tasks and parameter settings in Section 4.1.4, and statistical tests used for analysis in Section 5.1.5. In Table D-3, we provide a summary of the experiment design.

5.1.1 Research Questions

The overall objective of the evaluation is to investigate if NSGA-II and NSGA-III are effective, as compared to RS, in terms of solving the configuration generation problem, and assess the quality of rules mined using two machine learning algorithms (PART and C4.5) when combined with NSGA-II and NSGA-III. The overall objective can be achieved by answering the following research questions:

RQ1. Are NSGA-II and NSGA-III effective to generate configurations for the purpose of mining rules as compared to RS?

RQ2. Does SBRM$^+$ produce better quality rules (in terms of machine learning measurements) than RBRM$^+$?

RQ3. To what extent the quality of rules improved using SBRM$^+$ in comparison to RBRM$^+$ (after the final iteration)?

RQ4. Which one of NSGA-II and NSGA-III is more effective to generate configurations for mining rules?

RQ5. Which one of PART and C4.5, when combined with NSGA-II and NSGA-III, produces better quality rules?

RQ6. How is the quality of rules correlated with average fitness values and quality indicators?

RQ7. What is the trend of the quality of rules produced by SBRM$^+$ across the iterations?

RQ8. Is it feasible to apply SBRM$^+$ in practice in terms of time required for employing search to generate configurations?

5.1.2 Case Studies

Cisco Systems[18], Norway provides a variety of VCSs to facilitate high-quality virtual meetings [48]. Cisco has developed several product lines for VCS including C-Series, MX-Series, and SX-Series. Each product from these different product lines has several configurable parameters (e.g., *defaultProtocol* and *encryption*), which need to be configured before making calls. For each VCS, we have a set of state variables representing the states of VCS (e.g., *callStatus*, *numberOfActiveCalls*, *cameraConnected*) that vary according to different hardware and software configurations. Each

[18] www.cisco.com/c/en/us/products/collaboration-endpoints/index.html

Table D-3. Overall design of the experiment*

RQs	Tasks	Evaluation metrics	Comparison/Treatment	Statistical tests and plot types
1	T_1-comparing fitness values and six quality indicators for $SBRM^+_{NSGA-II}$-PART and $SBRM^+_{NSGA-III}$-PART with $RBRM^+$-PART, and $SBRM^+_{NSGA-II}$-C45 and $SBRM^+_{NSGA-III}$-C45 with $RBRM^+$-C45	– FV-O1 – FV-O2 – FV-O3 – OFV – Hyper Volume (HV) – Inverted Generational Distance (IGD) – Epsilon (ϵ) – Euclidean Distance (ED) – Generational Distance (GD) – Generated Spread (GS)	– $SBRM^+_{NSGA-II}$-PART vs. $RBRM^+$-PART – $SBRM^+_{NSGA-III}$-PART vs. $RBRM^+$-PART – $SBRM^+_{NSGA-II}$-C45 vs. $RBRM^+$-C45$SBRM^+_{NSGA-III}$-C45 vs. $RBRM^+$-C45	– Mann-Whitney U-test – Vargha and Delaney's $\widehat{A}_{12}$ statistics
2	T_2-comparing the quality of rules for $SBRM^+_{NSGA-II}$-PART and $SBRM^+_{NSGA-III}$-PART with $RBRM^+$-PART, and $SBRM^+_{NSGA-II}$-C45 and $SBRM^+_{NSGA-III}$-C45 with $RBRM^+$-C45	– Accuracy – MAE – RMSE – RAE – RRSE – CC/FF/CF/FC-Precision – CC/FF/CF/FC-Recall – CC/FF/CF/FC-FMeasure		
3	T_3-quantifying the average relative improvements in the quality of rules based on MLQMs at the end of a cycle for $SBRM^+_{NSGA-II}$-PART, $SBRM^+_{NSGA-III}$-PART, $SBRM^+_{NSGA-II}$-C45, and $SBRM^+_{NSGA-III}$-C45	– ARI in Accuracy – ARI in MAE – ARI in RMSE – ARI in RAE – ARI in RRSE – ARI in CC/FF/CF/FC-Precision – ARI in CC/FF/CF/FC-Recall – ARI in CC/FF/CF/FC-FMeasure		– Column plot using average values
4	T_4-comparing fitness values and six quality indicators for $SBRM^+_{NSGA-II}$-C45 and $SBRM^+_{NSGA-II}$-PART with $SBRM^+_{NSGA-III}$-C45 and $SBRM^+_{NSGA-III}$-PART	– FV-O1 – FV-O2 – FV-O3 – OFV – Hyper Volume (HV)	– $SBRM^+_{NSGA-II}$-C45 vs. $SBRM^+_{NSGA-III}$-C45 – $SBRM^+_{NSGA-II}$-PART vs. $SBRM^+_{NSGA-III}$-PART	– Mann-Whitney U-test – Vargha and Delaney's $\widehat{A}_{12}$ statistics

		– Inverted Generational Distance (IGD) – Epsilon (ϵ) – Euclidean Distance (ED) – Generational Distance (GD) – Generated Spread (GS)		
5	T_5-comparing the quality of rules for SBRM$^+_{\text{NSGA-II}}$-PART, SBRM$^+_{\text{NSGA-III}}$-PART, SBRM$^+_{\text{NSGA-II}}$-C45, and SBRM$^+_{\text{NSGA-III}}$-C45	– Accuracy – MAE – RMSE – RAE – RRSE – CC/FF/CF/FC-Precision – CC/FF/CF/FC-Recall – CC/FF/CF/FC-FMeasure	– SBRM$^+_{\text{NSGA-II}}$-C45 vs. SBRM$^+_{\text{NSGA-II}}$-PART – SBRM$^+_{\text{NSGA-III}}$-C45 vs. SBRM$^+_{\text{NSGA-III}}$-PART – Winner (SBRM$^+_{\text{NSGA-II}}$-C45 vs. SBRM$^+_{\text{NSGA-II}}$-PART) vs. Winner (SBRM$^+_{\text{NSGA-III}}$-C45 vs. SBRM$^+_{\text{NSGA-III}}$-PART)	– Mann-Whitney U-test – Vargha and Delaney's $\hat{A}_{12}$ statistics
6	T_6-assessing the correlation of average fitness values and quality indicators with MLQMs for SBRM$^+_{\text{NSGA-II}}$-PART, SBRM$^+_{\text{NSGA-III}}$-PART, SBRM$^+_{\text{NSGA-II}}$-C45, and SBRM$^+_{\text{NSGA-III}}$-C45	– All the MQLMS vs. AFV-O1 – All the MQLMS vs. AFV-O2 – All the MQLMS vs. AFV-O3 – All the MQLMS vs. OAFV – All the MQLMS vs. HV – All the MQLMS vs. IGD – All the MQLMS vs. ϵ – All the MQLMS vs. ED – All the MQLMS vs. GD – All the MQLMS vs. GS	– SBRM$^+_{\text{NSGA-II}}$-C45 – SBRM$^+_{\text{NSGA-III}}$-C45 – SBRM$^+_{\text{NSGA-II}}$-PART – SBRM$^+_{\text{NSGA-III}}$-PART	– Spearman's correlation
7	T_7-assessing the trend of the quality of rules based on MLQMs across the iterations for SBRM$^+_{\text{NSGA-II}}$-PART, SBRM$^+_{\text{NSGA-III}}$-PART, SBRM$^+_{\text{NSGA-II}}$-C45, and SBRM$^+_{\text{NSGA-III}}$-C45	– Accuracy – MAE – RMSE – RAE – RRSE – CC/FF/CF/FC-Precision – CC/FF/CF/FC-Recall – CC/FF/CF/FC-FMeasure		– Scatter plot – Linear Regression
8	T_8-assessing the feasibility of applying search based on the average time required to generate configurations	– ATPI – ATPC	– SBRM$^+_{\text{NSGA-II}}$-C45 – SBRM$^+_{\text{NSGA-III}}$-C45 – RBRM$^+$-C45	– Average values

			– SBRM$^{+}_{\text{NSGA-II}}$-PART – SBRM$^{+}_{\text{NSGA-III}}$-PART – RBRM^{+}-PART	

* FV-O1= Fitness values for the first objective, FV-O2= Fitness values for the second objective, FV-O3= Fitness values for the third objective, OFV = Overall fitness values, MAE= Mean Absolute Error, RMSE= Root Mean Squared Error, RAE= Relative Absolute Error, RRSE= Root Relative Squared Error, CC=ConnectedConnected, FF= FailedFailed, FC= FailedConnected, CF= ConnectedFailed, ARI= Average Relative Improvement, AFV-O1= Average fitness values for the first objective, AFV-O2= Average fitness values for the second objective, AFV-O3= Average fitness values for the third objective, OAFV= Overall average fitness values, ATPI= Average time (minutes) required to generate configurations per iteration, ATPC= Average time (minutes) required to generate configurations per cycle.

product has several operations (e.g., *dial()*, *disconnect()*, *hold()*, *accept()*, *transfer()*) to support different interactions (e.g., making a call, sharing presentation) supported by the product. An operation can also take several parameters as input (e.g., *callType*, *callRate*, and *Protocol* for *dial()* operation). For our experiment, we used three real products C60, SX20, and MX300 developed by Cisco, which belong to three different product lines C-series, SX-series, and MX-series. We selected 27 configurable parameters (i.e., including network specific ones) for the Cisco case study, which were related to the call functionality. Simula Research Laboratory has a long-term collaboration with Cisco, Norway under Certus-SFI [92]. As part of our collaboration, we have access to several VCSs at our lab, and thus we used these systems for our experiments. Therefore, our case study is real, but the experiment was not performed in the real industrial setting of Cisco.

Jitsi [93] is a real-world open source Audio/Video Internet Phone, and Instant Messenger developed in Java, which supports several known protocols including *SIP*, *AIM*, and *ICQ*. Jitsi was developed based on the OSGI architecture using Apache-Felix implementation. Jitsi provides a large number of features such as encrypted audio/video conference calls, messaging, desktop sharing, call hold, transfer, and call recording. Jitsi has several configurable parameters (e.g., *sIPZtpr*, *defaultProtocol*, *audioCodec*) and state variables (e.g., *callStatus*, *numberOfConferenceParticipants*). Just like the Cisco case study, Jitsi also has several operations such as *dial()*, *accept()*, and *hold()*. We extended the case study by adding a new OSGI bundle to introduce several new configurable parameters (e.g., *defaultCallRate*, *MTU*) and implemented several rules constraining the configurable parameter values. These implemented rules determine the success of a call connection based on configurable parameter values assigned to the configurable parameters of the caller and two callees, as we used three instances (products) of Jitsi in our experiment as for the Cisco case study. The total number of the configurable parameters selected for the Jitsi case study is 39.

For both Cisco and Jitsi case studies, we selected making a call as the interaction because making a call is the main functionality of a VCS/VoIP and other functionalities depend on it. The call statuses of both callees were therefore selected as the state variables. Based on the two-state variables (i.e., call statuses for both callees) system states were defined by concatenating their state values, which were used to classify the configurations. For both case studies (i.e., Cisco and Jitsi), we have one normal system state *ConnectedConnected* (CC) and three abnormal system states *FailedFailed* (FF), *FailedConnected* (FC), and *ConnectedFailed* (CF), constituting four classes in our rule-mining problem, which is in nature a classification problem in machine learning. The *ConnectedConnected* shows that caller is connected to both of the callees successfully and *FailedFailed* indicates that the caller is failed to establish connections with the two callees. *FailedConnected* shows that the caller is connected to the second callee and failed to connect with the first callee whereas *ConnectedFailed* shows that caller is successfully connected with the first callee but failed to connect with the second callee. For both case study, to enable the making a call interaction, we used two operations *dial()* and *disconnect()* of the caller, one operation *accept()* for both callees.

5.1.3 Evaluation Metrics

To answer RQ1 (Table D-3), we compared NSGA-II and NSGA-III with RS in terms of FV-O1, FV-O2, FV-O3, and OFV. FV-O1, FV-O2, and FV-O3 are fitness values of Objective-1, Objective-2, and Objective-3 respectively (Section 3.2) whereas OFV is the overall fitness. OFV is calculated by taking the average of FV-O1, FV-O2, and FV-O3, as common practice [94]. Additionally, we compared NSGA-II and NSGA-III with RS in terms of six quality indicators: *Hypervolume (HV)*, *Inverted Generational Distance (IGD)*, *Epsilon (ϵ)*, *Euclidean Distance from the Ideal Solution (ED)*, *Generational Distance (GD)*, and *Generated Spread (GS)*. These quality indicators have been used in the existing literature [95-99] to measure the quality of solutions produced by the search algorithms in terms of convergence and diversity. Lower values of all the quality indicators except *HV* show better performance of the algorithm. Since the optimal Pareto font PF_0 is not known for our problem like most of the real-world problems, thus, we used reference Pareto front to compute the values of indicators. To compute the reference Pareto front, we combined the Pareto fronts produced by all the search algorithms. Note, we computed two separate reference Pareto fronts for the approaches using C45 and PART as rule mining algorithms.

To answer RQ2 (Table D-3), we compared *SBRM*$^{+}_{\text{NSGA-II}}$*-C45* and *SBRM*$^{+}_{\text{NSGA-III}}$*-C45* with *RBRM*$^{+}$*-C45*, and *SBRM*$^{+}_{\text{NSGA-II}}$*-PART* and *SBRM*$^{+}_{\text{NSGA-III}}$*-PART* with *RBRM*$^{+}$*-PART* based on 17 (i.e., five related to the classifier and 12 related to the four classes) machine-learning quality measurements (MLQMs): *Accuracy*, *Mean Absolute Error* (*MAE*), Root *Mean Squared Error* (*RMSE*), *Relative Absolute Error* (*RAE*), and *Root Relative Squared Error* (*RRSE*) of a classifier and *Precision*, *Recall*, and *FMeasure* for the four classes [86]. To differentiate *Precision*, *Recall*, and *FMeasure* corresponding to four classes, we used abbreviations of the classes with *Precision*, *Recall*, and *FMeasure*. For example, *Precision* for *ConnectedConnected* is represented as *CC-Precision*.

- *Accuracy* indicates the overall performance of rule mining algorithms (e.g., C4.5, PART) by specifying the percentage of instances that conform to mined rules [83], where one instance contains one specific configuration (i.e., a set of configurable parameter values for the selected configurable parameters of the communicating products) and corresponding system state.
- *Precision* represents the percentage of instances that are correctly classified divided by the total number of instances covered by rules associated with a specific system state (i.e., defined based on the call statuses of both callees in our case). For example, 98% *FF-Precision* means that, according to the mined rules, there are 2% of instances whose configurations are identified as invalid ones, which led to the *FailedFailed* state. But actually, they lead to the other states (e.g., *ConnectedConnected*, *FailedConnected*, *ConnectedFailed*).
- *Recall* represents the percentage of instances that are correctly classified divided by the total number of instances corresponding to a particular system state. For example, 90% *FF-Recall* means that configurations of 10% instances are not associated with the *FailedFailed* state according to the mined rules, but these instances actually lead to the *FailedFailed* state.
- *FMeasure* is the harmonic mean of *Precision* and *Recall* [83].
- *Mean Absolute Error (MAE)* represents an average of individual errors (i.e., differences between values predicted by the classifier and the actual observed values) without considering the sign of the error.
- *Root Mean Squared Error (RMSE)* is the square root of the mean of the absolute squared error (i.e., square of *MAE*). *Root Mean Squared Error* amplify the effect of outliers (i.e., individuals with large errors) by squaring their errors.

- *Relative Absolute Error (RAE)* is calculated as *MAE* divided by the error of the default predictor (i.e., ZeroR classifier, which simply selects the most frequent value from training dataset (if nominal) or the average value (if numerical).
- *Root Relative Squared Error (RRSE)* is the square root of the relative mean squared error (i.e., square of *RAE*) [86].

For calculating the values for the MLQMs mentioned above, we used 10 times 10-fold stratified cross-validation [51, 86], as stratified cross-validation ensures that each class is (approximately) equally represented across each test fold [86] (Section 3).

For answering RQ3 (Table D-3), we calculate the average relative improvements (ARIs) in terms of 17 MLQMs mentioned above achieved at the end of each cycle (i.e., after *iteration-5*) using *SBRM*$^+_{\text{NSGA-II}}$-*C45* and *SBRM*$^+_{\text{NSGA-III}}$-*C45* in comparison to *RBRM*$^+$-*C45*, and *SBRM*$^+_{\text{NSGA-II}}$-*PART* and *SBRM*$^+_{\text{NSGA-III}}$-*PART* in comparison to *RBRM*$^+$-*PART*. We calculated the ARIs for *Accuracy* of classifier and *Precision*, *Recall*, and *FMeasure* for all the classes with respect to *SBRM*$^+_{\text{NSGA-II}}$-*C45*, *SBRM*$^+_{\text{NSGA-III}}$-*C45*, *SBRM*$^+_{\text{NSGA-II}}$-*PART*, and *SBRM*$^+_{\text{NSGA-III}}$-*PART* as:

$$ARI = \frac{\sum_{c=1}^{10}(S(x_{ic}) - R(x_{ic}))}{10} \quad (9)$$

where $S(x_{ic})$ and $R(x_{ic})$ give the values of ith MLQM in *iteration-5* for cth cycle corresponding to *SBRM*$^+_{\text{NSGA-II}}$-*C45* or *SBRM*$^+_{\text{NSGA-III}}$-*C45* (*SBRM*$^+_{\text{NSGA-II}}$-*PART* or *SBRM*$^+_{\text{NSGA-III}}$-*PART*) and *RBRM*$^+$-*C45*(*RBRM*$^+$-*PART*), respectively. To calculate the ARIs for *MAE*, *RAE*, *RMSE*, and *RRSE* with respect to *SBRM*$^+_{\text{NSGA-II}}$-*C45*, *SBRM*$^+_{\text{NSGA-III}}$-*C45*, *SBRM*$^+_{\text{NSGA-II}}$-*PART*, and *SBRM*$^+_{\text{NSGA-III}}$-*PART* we used the following formula:

$$ARI = \frac{\sum_{c=1}^{10}(R(x_{ic}) - S(x_{ic}))}{10} \quad (10)$$

For RQ4 (Table D-3), we compared NSGA-II with NSGA-III in terms of FV-O1, FV-O2, FV-O3, OFV*,* and six quality indicators as we did in RQ1 for comparing NSGA-II and NSGA-III with RS. For RQ5, we compared the quality of the rules produced from *SBRM*$^+_{\textit{NSGA-II}}$-*C45*, *SBRM*$^+_{\textit{NSGA-III}}$-*C45*, *SBRM*$^+_{\textit{NSGA-II}}$-*PART*, and *SBRM*$^+_{\textit{NSGA-III}}$-*PART*, based on 17 MLQMs mentioned above to find the best-suited search algorithm combined with rule mining algorithm for mining CPL rules. To answer RQ6 (Table D-3), we computed the correlation estimates (ρ) and the *p*-values using the Spearman's test corresponding to all the 17 MLQMs in correlation to the average fitness values for the three individual objectives (i.e., *AFV-O1*, *AFV-O2*, and *AFV-O3*), overall average fitness (*OAFV*), and six quality indicators (i.e., *HV*, *IGD*, ϵ, *ED*, *GD*, and *GS*). *AFV-O1*, *AFV-O2*, *AFV-O3*, and *OAFV* are calculated based on the values of *FV-O1*, *FV-O2*, *FV-O3*, and *OFV* respectively, corresponding to each iteration of all the runs. For example, *AFV-O1* corresponding to one iteration can be calculated as: $AFV - O1 = \frac{\sum_{i=1}^{500} FV-O1_i}{500}$, where 500 is the total number of fitness values. For RQ7 (Table D-3), we assessed the trend of the quality of rules in terms of above-mentioned 17 MLQMs for *SBRM*$^+_{\textit{NSGA-II}}$-*C45*, *SBRM*$^+_{\textit{NSGA-III}}$-*C45*, *SBRM*$^+_{\textit{NSGA-II}}$-*PART*, and *SBRM*$^+_{\textit{NSGA-III}}$-*PART*, across the five iterations. To answer RQ8 (Table D-3), we calculated the average time required by NSGA-II in *SBRM*$^+_{\textit{NSGA-II}}$-*C45* and *SBRM*$^+_{\textit{NSGA-II}}$-*PART*, NSGA-III in *SBRM*$^+_{\textit{NSGA-III}}$-*C45* and *SBRM*$^+_{\textit{NSGA-III}}$-*PART*, and RS in *RBRM*$^+$-*C45* and *RBRM*$^+$-*PART* for generating configurations per iteration (*ATPI*) and per cycle (*ATPC*). *ATPI* is calculated as: $ATPI = \frac{\sum_{i=1}^{5}\sum_{c=1}^{10} T_{ic}}{50}$, where T_{ic} represents the time required by the approach in the ith iteration of the cth cycle. *ATPC* is calculated as: $ATPC = \sum_{i=1}^{5}(\frac{\sum_{c=1}^{10} T_{ic}}{10})$.

5.1.4 Experimental Tasks and Parameter Settings

As shown in Table D-3, we designed eight tasks (T_1-T_8) for addressing RQ1-RQ8. We used the default settings for NSGA-II and NSGA-III as implemented in jMetal [53, 100]. The single point crossover and bit-flip mutation, implemented in jMetal were applied as crossover and mutation operators, respectively with 0.9 crossover rate and (1/total number of configurable parameters) mutation rate. We used a population size of 500 and 50,000 fitness evaluations where we selected all the Pareto Non-dominated configuration solutions for mining the rules. NSGA-III produces 92 solutions for three objective problems regardless the larger population size [80], thus, we executed it using multiple threads to get 500 solutions in one run. We used five iterations per cycle, and in each iteration, we run the search algorithm (NSGA-II, NSGA-III, or RS) once, which means we have five runs of the search algorithm in a complete cycle. We used total 10 cycles (i.e., 50 runs of the search algorithm) for our experiment to cater the randomness inherited in the search algorithms.

Since selecting the best set of parameters is application dependent [37], we used the default settings provided by Weka [74] for both PART and C4.5. Default settings have been used in various contexts such as mining rules for video generator product line [37] and comparing the performance of different classification algorithms [55]. We used 0.25 and 2 for minConfidence (i.e., the minimum confidence for a rule) and minNumObj (i.e., the minimum number of instances for a rule) respectively.

5.1.5 Statistical Analyses

As inspired by [101], we systematically conducted three types of analyses: *difference analysis*, *correlation analysis*, and *trend analysis* to answer RQ1-RQ2 and RQ4-RQ7 (Section 4.1.1). To answer RQ3 and RQ8, we report descriptive statistics, as for RQ3 we intend to assess the magnitude of ARI achieved by $SBRM^{+}_{NSGA\text{-}II}$-*C45*, $SBRM^{+}_{NSGA\text{-}III}$-*C45*, $SBRM^{+}_{NSGA\text{-}II}$-*PART*, and $SBRM^{+}_{NSGA\text{-}III}$-*PART* whereas, for RQ8, we aim to show the total time required for generating configurations.

Difference analysis is the most commonly used analysis, which studies the distributions of a single measure between two groups. We carried out the difference analysis to compare rule mining approaches (e.g., $SBRM^{+}_{NSGA\text{-}II}$-*C45*, $RBRM^{+}$-*C45*) in terms of fitness values, quality indicators, and MLQMs (Table D-3) to answer RQ1, RQ2, RQ4, and RQ5. To conduct the *difference analysis*, we use the non-parametric Mann-Whitney U-test as recommended in [56] with a significance level of 0.05 and the Vargha and Delaney's $\hat{A}_{12}$ statistics as an effect size measure [57]. For comparing the *Accuracy, Precision*, *Recall*, *F-measure*, and *HV* for a comparison pair (A_i vs. A_j), if $\hat{A}_{12}$ is greater than 0.5, A_i is better than A_j, and a value less than 0.5 means vice versa. Similarly, in the case of fitness values, *IGD*, ϵ, *ED*, *GD*, *GS MAE*, *RMSE*, *RAE*, and *RRSE,* if $\hat{A}_{12}$ is less than 0.5, A_i is better than A_j, otherwise, A_j is better than A_i. $A_i > A_j$ shows that approach A_i performed significantly better than A_j based on the results of the Mann-Whitney U-test and Vargha and Delaney's $\hat{A}_{12}$ statistics. Similarly, $A_i < A_j$ shows that A_j significantly outperformed A_j whereas $A_i = A_j$ indicates no significant difference between the two approaches being compared.

Correlation analysis evaluates the correlation (positive/negative) between two variables (e.g., x and y) and its statistical significance. To find the correlation of MLQMs with average fitness values and six quality indicators (RQ6), we applied the nonparametric Spearman's test [102] and reported the correlation coefficients (ρ) and *p*-values. The value of ρ ranges from -1 to 1 where a value of $\rho > 0$ (or $\rho < 0$) shows a positive (or negative) correlation between x and y, whereas $\rho = 0$ indicates no correlation. The p-value lower than 0.05 shows the correlation is statistically significant. The analysis aims to test whether *Accuracy, Precision*, *Recall*, and *FMeasure* are positively correlated with *HV* and negatively correlated with average fitness values, *IGD*, ϵ, *ED*, *GD*, and *GS* and *MAE*, *RMSE*, *RAE*, and RRSE have a negative correlation with *HV* and a positive correlation with average fitness values, *IGD*, ϵ, *ED*, *GD*, and *GS* (i.e., hypothesis). Satisfying the hypothesis is regarded as good performance of the approach because we believe that smaller fitness and indicator values (except for *HV*, as the larger *HV* is better) lead to better quality of rules in terms of MLQMs.

To discover the trend of the quality of rules based on MLQMs (RQ7), we constructed 2D scatter plots and fitted linear regression lines. In 2D plots, the x-axis represents the iteration number in one cycle, and the y-axis represents different machine learning quality measurements such as *Accuracy*. This kind of analyses indicates variation in the quality of rules across the iterations.

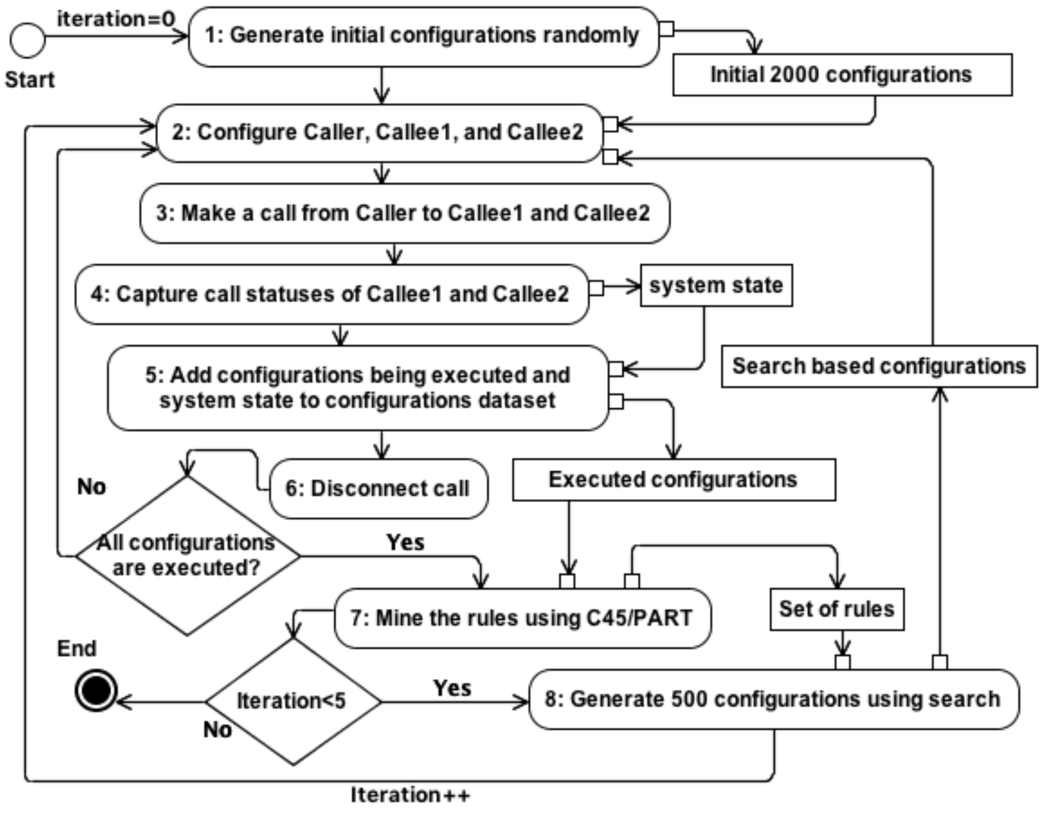
For assessing the magnitude of average relative improvements (ARIs) in the quality of rules in terms of MLQMs (RQ3), we reported mean, min, and max values. Similarly, for assessing the feasibility of applying NSGA-II in *SBRM*$^{+}_{NSGA-II}$*-C45* and *SBRM*$^{+}_{NSGA-II}$*-PART*, NSGA-III in *SBRM*$^{+}_{NSGA-III}$*-C45* and *SBRM*$^{+}_{NSGA-III}$*-PART* based on the time required for generating configurations (RQ8), we reported average values of time required to generate configurations per iteration (i.e., *ATPI*) and per cycle (*ATPC*).

5.2 Experiment Execution

Figure D-6 presents an overview of the experiment execution. As shown in Figure D-6, at the first step, we randomly generated a set of 2000 configurations corresponding to the three selected products (*Caller*, *Callee1*, and *Callee2*) for each case study. For the Cisco case study, we selected C60 (i.e., *Caller*), MX300 (i.e., *Callee1*), and SX20 (i.e., *Callee2*). For the Jitsi case study, we used three instances (products) belonging to the same product line. At the second step, we configured the *Caller*, *Callee1*, and *Callee2* using randomly generated configurations and made a call from *Caller* to *Callee1* and *Callee2* for 10 seconds (step 3). We made the call for 10 seconds to give sufficient time for establishing the call connection. To make the call, first, we execute the *dial()* operation of *Caller* and then *accept()* operation of *Callee1* and *Callee2*. In step 4, we captured call statuses of *Callee1* and *Callee2* to get the system state and added the current configuration being executed and its corresponding system state to the executed configurations (step 5) whereas, in step 6, we disconnected the call by executing the *disconnect()* operation of *Caller*. We repeated step 2 to step 6 until all the configurations (2000 configurations) are executed. In step 7, we input executed configurations containing 2000 configurations along with their corresponding system states to Weka [74] and applied PART [40] and C4.5 to mine the initial set of rules.

To refine the rules, we used the initial set of rules to guide the search algorithms (i.e., NAGA-II, NAGA-III, and RS) to generate 500 more configurations (step 8). For mining the refined set of rules, we repeated the same process starting from step 2 to step 7 (i.e., configuring the products, making the calls, adding the configurations and associated system states to the executed configurations, disconnecting the calls, and mining the rules using all the executed configurations). We repeated this incremental and iterative process for five iterations in a complete cycle and mined the final set of rules based on a dataset (i.e., represented as executed configurations in Figure D-6) containing 4500 configurations and corresponding system states. We used five iterations as a stopping criterion.

For generating configurations using NSGA-II, NSGA-III, or RS for both case studies, we ran the experiment on a laptop with Intel Core i7 2.8 GHz CPU and 16GB RAM running the macOS Sierra v10.12.5 operating system. To make calls for the Jitsi case study, we installed three instances of Jitsi (*Caller*, *Callee1*, *Callee2*) on three computers. *Caller* was installed on the laptop mentioned above (i.e., the one used for generating configurations). *Caller1* was installed on a desktop (iMac) with Intel Core i5 CPU 2.7 GHz and 8GB RAM running the macOS Sierra v10.12.4 operating system. *Caller2* was installed on a laptop with Intel Core i7 CPU 2.5 GHz and 16GB RAM running the Windows-7 (x64) operating system. For the Cisco case study, all the three products have their dedicated hardware.



iteration=0
Start
1: Generate initial configurations randomly
Initial 2000 configurations
2: Configure Caller, Callee1, and Callee2
3: Make a call from Caller to Callee1 and Callee2
4: Capture call statuses of Callee1 and Callee2
system state
5: Add configurations being executed and system state to configurations dataset
Search based configurations
No
6: Disconnect call
Executed configurations
All configurations are executed?
Yes
7: Mine the rules using C45/PART
Set of rules
End
Iteration<5
Yes
No
8: Generate 500 configurations using search
Iteration++

Figure D-6. An overview of the experiment execution

6 Results and Analysis

In this section, we present the results and analysis of the evaluation and answer the research questions for both of the case studies (i.e., Cisco and Jitsi).

6.1 Effectiveness of Search (RQ1)

To answer RQ1, we compare *SBRM*$^+_{NSGA\text{-}II}$*-C45* and *SBRM*$^+_{NSGA\text{-}III}$*-C45* with *RBRM*$^+$*-C45* and *SBRM*$^+_{NSGA\text{-}II}$*-PART* and *SBRM*$^+_{NSGA\text{-}III}$*-PART* with *RBRM*$^+$*-PART* regarding the fitness values (i.e., *FV-O1*, *FV-O2*, *FV-O3*, and *OFV*) and six quality indicators corresponding to five individual iterations as well as overall, for both of the two case studies. In Table D-4, we summarize the results for answering RQ1 whereas the detailed results can be found in the technical report corresponding to this paper [103].

The results of Man-Whitney U-test and Vargha and Delaney's $\widehat{A}_{12}$ for all the fitness values (i.e., FV-O1, FV-O2, FV-O3, and OFV) show that *SBRM*$^{+}$ (i.e., *SBRM*$^{+}_{NSGA\text{-}II}$*-C45*, *SBRM*$^{+}_{NSGA\text{-}III}$*-C45*, *SBRM*$^{+}_{NSGA\text{-}II}$*-PART* and *SBRM*$^{+}_{NSGA\text{-}III}$*-PART*) significantly outperformed *RBRM*$^{+}$ (i.e., *RBRM*$^{+}$*-C45* and *RBRM*$^{+}$*-PART*) corresponding to both the Cisco and Jitsi case studies. Similarly, from the results of the quality indicators (Table D-4), we noticed that *SBRM*$^{+}$ significantly outperformed *RBRM*$^{+}$ in terms of the majority of the comparisons (i.e., minimum 25 and maximum 32 out of 36 comparisons). Note, for each comparison pair, we have six comparisons (five individual iterations and overall) in terms of a particular quality indicator for one case study (i.e., total 36 comparisons for six indicators per case study and 48 comparisons for one quality indicator for all comparisons pairs and two case studies). We observed that for five indicators (except for *GS*), *SBRM*$^{+}$ significantly outperformed *RBRM*$^{+}$ for 221 out of 240 comparisons whereas *RBRM*$^{+}$ significantly outperformed *SBRM*$^{+}$ in terms of *GS* for 32 out of 48 comparisons for both of the case studies. Based on the results of RQ1, it can be concluded that NSGA-II and NSGA-III are more effective than RS for configuration generation problem. The detailed results of RQ1 can be found in [103].

Table D-4. Comparing SBRM^{+} with RBRM in terms of the quality indicators*

Case study	Comparison Pair											
	A_1 vs A_5			**A_2 vs A_5**			**A_3 vs A_6**			**A_4 vs A_6**		
	$A_1>A_5$	$A_1<A_5$	$A_1=A_5$	$A_2>A_5$	$A_2<A_5$	$A_2=A_5$	$A_3>A_6$	$A_3<A_6$	$A_3=A_6$	$A_4>A_6$	$A_4<A_6$	$A_4=A_6$
Cisco	25/36	6/36	5/36	32/36	0/36	4/36	28/36	6/36	2/36	30/36	3/36	3/36
Jitsi	32/36	1/36	3/36	29/36	6/36	1/36	27/36	4/36	5/36	27/36	6/36	3/36

*A_1= SBRM$^{+}_{NSGA\text{-}II}$-C45, A_2= SBRM$^{+}_{NSGA\text{-}III}$-C45, A_2= SBRM$^{+}_{NSGA\text{-}II}$-PART, A_4= SBRM$^{+}_{NSGA\text{-}III}$-PART, A_5= RBRM^{+}-C45, A_6= RBRM^{+}-PART

6.2 Comparing SBRM^{+} with RBRM^{+} (RQ2)

To answer RQ2, we compare *SBRM*$^{+}_{NSGA\text{-}II}$*-C45* and *SBRM*$^{+}_{NSGA\text{-}III}$*-C45* with *RBRM*$^{+}$*-C45*, and *SBRM*$^{+}_{NSGA\text{-}II}$*-PART* and *SBRM*$^{+}_{NSGA\text{-}III}$*-PART* with *RBRM*$^{+}$*-PART* in terms of MLQMs based on the rules mined from each iteration as well as *Overall* (i.e., combining the results of all the five iterations), for both case studies. In Table D-5, we summarize the results for answering RQ2. Detailed results are provided in [103] for reference.

Table D-5. Comparing SBRM^{+} with RBRM in terms of MLQMs*

Case study	Comparison Pair											
	A_1 vs A_5			**A_2 vs A_5**			**A_3 vs A_6**			**A_4 vs A_6**		
	$A_1>A_5$	$A_1<A_5$	$A_1=A_5$	$A_2>A_5$	$A_2<A_5$	$A_2=A_5$	$A_3>A_6$	$A_3<A_6$	$A_3=A_6$	$A_4>A_6$	$A_4<A_6$	$A_4=A_6$
Cisco	78/90	0/90	12/90	54/90	7/90	29/90	70/93	3/93	20/93	57/93	3/93	33/93
Jitsi	86/102	0/102	16/102	19/102	20/102	63/102	88/102	0/102	14/102	48/102	38/102	16/102

*A_1= SBRM$^{+}_{NSGA\text{-}II}$-C45, A_2= SBRM$^{+}_{NSGA\text{-}III}$-C45, A_2= SBRM$^{+}_{NSGA\text{-}II}$-PART, A_4= SBRM$^{+}_{NSGA\text{-}III}$-PART, A_5= RBRM^{+}-C45, A_6= RBRM^{+}-PART

As shown in Table D-5, for the Cisco case study, *SBRM*$^{+}_{NSGA\text{-}II}$*-C45* and *SBRM*$^{+}_{NSGA\text{-}III}$*-C45* significantly outperformed *RBRM*$^{+}$*-C45* in 87% (i.e., 78/90) and 60% (i.e., 54/90) of the total comparisons. Respectively. *SBRM*$^{+}_{NSGA\text{-}II}$*-PART* and *SBRM*$^{+}_{NSGA\text{-}III}$*-PART* significantly outperformed *RBRM*$^{+}$*-PART* in 75% (i.e., 70/93) and 61% (i.e., 57/93) of the total comparisons.

In 8% (i.e., 7/90) of the total comparisons, *RBRM*$^+$*-C45* significantly outperformed *SBRM*$^+$$_{\text{NSGA-III}}$*-C45* whereas for 3% (3/90) and 3%(3/93) of the total comparisons RBRM$^+$*-PART* significantly outperformed *SBRM*$^+$$_{\text{NSGA-II}}$*-PART* and *SBRM*$^+$$_{\text{NSGA-III}}$*-PART*, respectively. For the remaining comparisons, there was no significant difference between the *SBRM*$^+$ approaches and the *RBRM*$^+$ approaches.

Corresponding to the Jitsi case study, *SBRM*$^+$$_{\text{NSGA-II}}$*-C45* and *SBRM*$^+$$_{\text{NSGA-III}}$*-C45* significantly outperformed *RBRM*$^+$*-C45* in 84% (i.e., 86/102) and 19% (i.e., 19/102) of the total comparisons respectively, whereas *SBRM*$^+$$_{\text{NSGA-II}}$*-PART* and *SBRM*$^+$$_{\text{NSGA-III}}$*-PART* significantly outperformed *RBRM*$^+$*-PART* in 86% (i.e., 88/102) and 47% (i.e., 48/102) of the total comparisons. In 20% (i.e., 20/102) and 37% (i.e., 38/102) of total comparisons *RBRM*$^+$*-C45* and *RBRM*$^+$*-PART* significantly outperformed *SBRM*$^+$$_{\text{NSGA-III}}$*-C45* and *SBRM*$^+$$_{\text{NSGA-III}}$*-PART* respectively, whereas for the remaining comparisons there was no significant difference between the *SBRM*$^+$ (*SBRM*$^+$$_{\text{NSGA-II}}$*-C45*, *SBRM*$^+$$_{\text{NSGA-III}}$*-C45*, *SBRM*$^+$$_{\text{NSGA-II}}$*-PART*, and *SBRM*$^+$$_{\text{NSGA-III}}$*-PART*) and *RBRM*$^+$(*RBRM*$^+$ *-C45* and *RBRM*$^+$*-PART*).

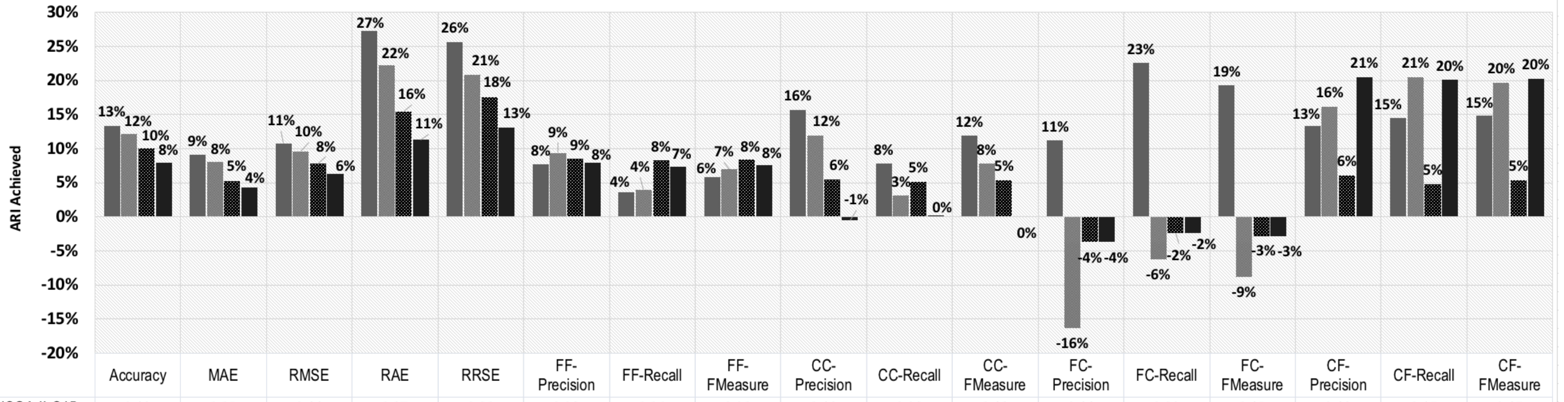
Since for both of the case studies, *SBRM*$^+$ significantly outperformed *RBRM*$^+$ in terms of the majority of MLQMs (i.e., 84% for *SBRM*$^+$$_{\text{NSGA-II}}$*-C45*, 86% for *SBRM*$^+$$_{\text{NSGA-II}}$*-PART*, and 47% for *SBRM*$^+$$_{\text{NSGA-III}}$*-PART*) except for *SBRM*$^+$$_{\text{NSGA-III}}$*-C45* corresponding to the Jitsi case study where neither one of the two approaches (i.e., *SBRM*$^+$$_{\text{NSGA-III}}$*-C45* and *RBRM*$^+$*-C45)* dominates the other. Thus, we can conclude that given the same context (i.e., the same case study, machine learning algorithm and its parameter settings) *SBRM*$^+$ tends to produce rules with higher quality as compared to *RBRM*$^+$ with respect to the MLQMs. In the worst case, *SBRM*$^+$ produces rules with the same quality as for *RBRM*$^+$.

6.3 Average Relative Improvements in the Quality of Rules (RQ3)

For RQ3, we computed the average relative improvements (ARIs) in terms of MLQMs achieved at the end of the cycle (i.e., after *iteration-5*) using *SBRM*$^+$ (*SBRM*$^+$$_{\text{NSGA-II}}$*-C45*, *SBRM*$^+$$_{\text{NSGA-III}}$*-C45*, *SBRM*$^+$$_{\text{NSGA-II}}$*-PART*, and *SBRM*$^+$$_{\text{NSGA-III}}$*-PART*) in comparison to *RBRM*$^+$(*RBRM*$^+$ *-C45* and *RBRM*$^+$*-PART*) (Section 5.1.3). In Figure D-7 and Figure D-8, we present the ARIs in terms of all the MLQMs for *SBRM*$^+$$_{\text{NSGA-II}}$*-C45*, *SBRM*$^+$$_{\text{NSGA-III}}$*-C45*, *SBRM*$^+$$_{\text{NSGA-II}}$*-PART*, and *SBRM*$^+$$_{\text{NSGA-III}}$*-PART* corresponding to the Cisco and Jitsi case studies respectively. Moreover, the detailed results are presented in [103].

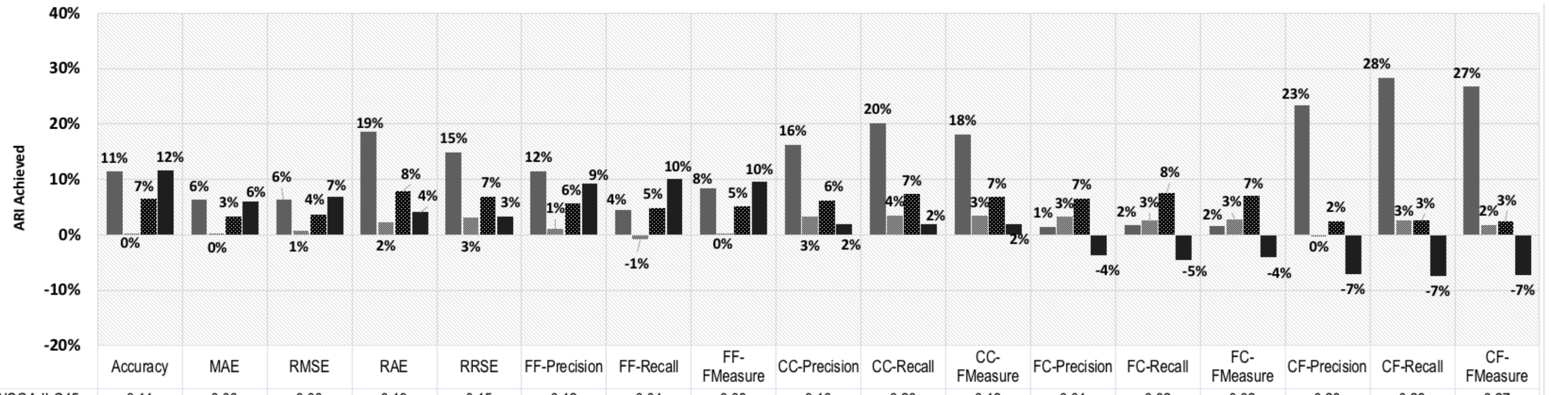
As shown in Figure D-7, for the Cisco case study, on average *SBRM*$^+$ achieved 8% to 13% higher *Accuracy* than *RBRM*$^+$ and 4% to 27% lower values for the four error-related MLQMs (i.e., *MAE*, *RMSE*, *RAE*, and *RRSE*). The ARIs in terms of *FF-Precision*, *FF-Recall*, and *FF-FMeasure* for *SBRM*$^+$ range between 4% and 9% and for *CC-Precision*, *CC-Recall*, and *CC-FMeasure*, the ARIs are up to 16%. The ARIs corresponding to *FC-Precision*, *FC-Recall*, and *FC-FMeasure* for *SBRM*$^+$$_{\textit{NSGA-II}}$*-C45* range between 11% and 23%, while *SBRM*$^+$$_{\textit{NSGA-III}}$*-C45*, *SBRM*$^+$$_{\textit{NSGA-II}}$*-PART*, and *SBRM*$^+$$_{\textit{NSGA-III}}$*-PART* have negative ARIs ranging from -16% to -2%. This is because they did not produce rules related to *FailedConnected* due to less number of configurations leading to *FailedConnected* system state. About *CF-Precision*, *CF-Recall*, and *CF-FMeasure*, the ARIs for *SBRM*$^+$ are between 5% and 21%.

As shown in Figure D-8, for the Jitsi case study, on average *SBRM*$^+$ achieved up to 12% higher *Accuracy* and 19% lower values for error-related MLQMs (i.e., *MAE*, *RMSE*, *RAE*, and *RRSE*) as compared to *RBRM*$^+$. In terms of *FF-Precision*, *FF-Recall*, and *FF-FMeasure*, the ARIs



	Accuracy	MAE	RMSE	RAE	RRSE	FF-Precision	FF-Recall	FF-FMeasure	CC-Precision	CC-Recall	CC-FMeasure	FC-Precision	FC-Recall	FC-FMeasure	CF-Precision	CF-Recall	CF-FMeasure
■ SBRM+NSGA-II-C45	0,13	0,09	0,11	0,27	0,26	0,08	0,04	0,06	0,16	0,08	0,12	0,11	0,23	0,19	0,13	0,15	0,15
■ SBRM+NSGA-III-C45	0,12	0,08	0,10	0,22	0,21	0,09	0,04	0,07	0,12	0,03	0,08	-0,16	-0,06	-0,09	0,16	0,21	0,20
▨ SBRM+NSGA-II-PART	0,10	0,05	0,08	0,16	0,18	0,09	0,08	0,08	0,06	0,05	0,05	-0,04	-0,02	-0,03	0,06	0,05	0,05
■ SBRM+NSGA-III-PART	0,08	0,04	0,06	0,11	0,13	0,08	0,07	0,08	-0,01	0,00	0,00	-0,04	-0,02	-0,03	0,21	0,20	0,20

Figure D-7. RI achieved by SBRM$^+_{\text{NSGA-II}}$-C45, SBRM$^+_{\text{NSGA-III}}$-C45, SBRM$^+_{\text{NSGA-II}}$-PART, and SBRM$^+_{\text{NSGA-III}}$-PART for the Cisco case study



	Accuracy	MAE	RMSE	RAE	RRSE	FF-Precision	FF-Recall	FF-FMeasure	CC-Precision	CC-Recall	CC-FMeasure	FC-Precision	FC-Recall	FC-FMeasure	CF-Precision	CF-Recall	CF-FMeasure
■ SBRM+NSGA-II-C45	0,11	0,06	0,06	0,19	0,15	0,12	0,04	0,08	0,16	0,20	0,18	0,01	0,02	0,02	0,23	0,28	0,27
■ SBRM+NSGA-III-C45	0,00	0,00	0,01	0,02	0,03	0,01	-0,01	0,00	0,03	0,04	0,03	0,03	0,03	0,03	-0,00	0,03	0,02
▨ SBRM+NSGA-II-PART	0,07	0,03	0,04	0,08	0,07	0,06	0,05	0,05	0,06	0,07	0,07	0,07	0,08	0,07	0,02	0,03	0,03
■ SBRM+NSGA-III-PART	0,12	0,06	0,07	0,04	0,03	0,09	0,10	0,10	0,02	0,02	0,02	-0,04	-0,05	-0,04	-0,07	-0,07	-0,07

Figure D-8. ARI achieved by SBRM$^+_{\text{NSGA-II}}$-C45, SBRM$^+_{\text{NSGA-III}}$-C45, SBRM$^+_{\text{NSGA-II}}$-PART, and SBRM$^+_{\text{NSGA-III}}$-PART for the Jitsi case study for *SBRM*$^{+}$ are between -1% and 12% whereas for *CC-Precision*, *CC-Recall*, and *CC-FMeasure*, the ARIs range between 2% and 20%. Concerning *FC-Precision*, *FC-Recall*, and *FC-FMeasure*, the ARIs for *SBRM*$^{+}$ are up to 8% whereas the ARIs in terms of *CF-Precision*, *CF-Recall*, and *CF-FMeasure* are up to 28%. However, we observed that for some MLQMs (e.g., *CF-Precision*, *CF-Recall*) for *SBRM*$^{+}_{\text{NSGA-III}}$*-PART* have negative ARIs.

From Figure D-7, one can observe that *SBRM*$^{+}$ has positive improvements for the majority of the MLQMs (i.e., 85%) with an ARI up to 27% for the Cisco case study. Similarly, for the Jitsi case study, Figure D-8 shows that *SBRM*$^{+}$ has positive values for ARIs corresponding to the majority of the MLQMs (i.e., 90%) with an ARI up to 28%. This shows that for both of the case studies, *SBRM*$^{+}$ has significantly improved the quality of rules in terms of MLQMs as compared to *RBRM*$^{+}$, as also suggested by the statistical analysis results (Section 6.2).

6.4 Comparing the Effectiveness of NSGA-II and NSGA-III (RQ4)

To answer RQ4, we compare *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* with *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* and *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART* with *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART* in terms of the fitness values (i.e., *FV-O1*, *FV-O2*, *FV-O3*, and *OFV*) and the six quality indicators (Section 4.1.3) for both of the two case studies. Table D-6 summarizes the results of RQ4 whereas detailed results are presented in [103].

Table D-6. Comparing SBRM$^{+}_{\text{NSGA-II}}$-C45 with SBRM$^{+}_{\text{NSGA-III}}$-C45 and SBRM$^{+}_{\text{NSGA-II}}$-PART with SBRM$^{+}_{\text{NSGA-III}}$-PART in terms of fitness values and quality indicators*

Case study	Fitness value based comparison						Quality indicators based comparison					
	A_1 vs. A_2			**A_3 vs. A_4**			**A_1 vs. A_2**			**A_3 vs. A_4**		
	$A_1>A_2$	$A_1<A_2$	$A_1=A_2$	$A_3>A_4$	$A_3<A_4$	$A_3=A_4$	$A_1>A_2$	$A_1<A_2$	$A_1=A_2$	$A_3>A_4$	$A_3<A_4$	$A_3=A_4$
Cisco	7/24	16/24	1/24	4/24	16/24	4/24	2/36	23/36	11/36	2/36	28/36	6/36
Jitsi	7/24	17/24	0/24	5/24	15/24	4/24	18/36	4/36	14/36	5/36	4/36	27/36

*A_1= SBRM$^{+}_{\text{NSGA-II}}$-C45, A_2= SBRM$^{+}_{\text{NSGA-III}}$-C45, A_2= SBRM$^{+}_{\text{NSGA-II}}$-PART, A_4= SBRM$^{+}_{\text{NSGA-III}}$-PART

As shown in Table D-6, for the Cisco (Jitsi) case study, *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* for 16/24 (17/24) fitness-based comparisons and *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART* significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART* for 16/24 (15/24) comparisons whereas in only 7/24 (7/24) and 4/24 (5/24) fitness-based comparisons *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* and *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART* significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* and *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART* respectively.

In terms of quality indicators, Table D-6 shows that for the Cisco case study, *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART*) significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART*) for 23/36 (28/36) indicator-based comparisons whereas for only 2/36 (2/36) indicator-based comparisons *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART*) significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART*). Similarly, for the Jitsi case study, *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART*) significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART*) in terms of the quality indicators for 18/36 (5/36) comparisons whereas for only 4/36 (4/36) *SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}III}}$*-PART*) significantly outperformed *SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-C45* (*SBRM*$^{+}_{\mathit{NSGA\text{-}II}}$*-PART*). To summarize the results of RQ4, we can notice that in most of the cases NSGA-III significantly outperformed NSGA-II in terms of fitness values and quality indicators, however, in some cases (e.g., for *GS*) we observed otherwise.

6.5 Comparing the quality of rules for SBRM+ (RQ5)

To answer RQ5, we compare the four *SBRM*⁺ approaches in terms of MLQMs based on the rules from each iteration and *Overall* (i.e., the rules of all the five iterations) for both of the case studies. To do so, first, we compare *SBRM*$^+_{NSGA\text{-}II}$*-C45* with *SBRM*$^+_{NSGA\text{-}III}$*-C45* and *SBRM*$^+_{NSGA\text{-}II}$*-PART* with *SBRM*$^+_{NSGA\text{-}III}$*-PART* and then we compare the two better performing approaches from these two comparisons to find the best. Table D-7 summarizes the results of RQ5 whereas the details results can be found in [103].

Table D-7. Comparing the quality of rules for the SBRM⁺ approaches in terms of MLQMs*

Case study	Comparison Pair								
	A_1 vs. A_2			**A_3 vs. A_4**			**W_1 vs. W_2**		
	$A_1>A_2$	$A_1<A_2$	$A_1=A_2$	$A_3>A_4$	$A_3<A_4$	$A_3=A_4$	$W_1>W_2$	$W_1<W_2$	$W_1=W_2$
Cisco	49/90	6/90	35/90	24/84	6/84	54/84	74/90	3/90	13/90
Jitsi	85/102	2/102	15/102	60/102	35/102	7/102	2/102	82/102	18/102

*A_1= SBRM$^+_{NSGA\text{-}II}$-C45, A_2= SBRM$^+_{NSGA\text{-}III}$-C45, A_2= SBRM$^+_{NSGA\text{-}II}$-PART, A_4= SBRM$^+_{NSGA\text{-}III}$-PART, W_1= Winner of A_1 vs. A_2, W_2= Winner of A_3 vs. A_4

As shown in Table D-7, for the two case studies, *SBRM*$^+_{NSGA\text{-}II}$*-C45* significantly outperformed *SBRM*$^+_{NSGA\text{-}III}$*-C45* in 54% (i.e., 49/90) and 83% (i.e., 85/102) of the total comparisons respectively whereas in only 7% (i.e., 6/90) and 2% (i.e., 2/102) of the total comparisons *SBRM*$^+_{NSGA\text{-}III}$-C45 significantly outperformed *SBRM*$^+_{NSGA\text{-}II}$*-C45*. Similarly, *SBRM*$^+_{NSGA\text{-}II}$*-PART* significantly outperformed *SBRM*$^+_{NSGA\text{-}III}$*-PART* in 29% (i.e., 24/84) and 59% (i.e., 60/102) of total comparisons for the Cisco and Jitsi case studies respectively whereas in 7% (i.e., 6/84) and 34% (i.e., 35/102) of total comparisons *SBRM*$^+_{NSGA\text{-}III}$*-PART* significantly performed better than *SBRM*$^+_{NSGA\text{-}II}$*-PART*. Since *SBRM*$^+_{NSGA\text{-}II}$*-C45* and *SBRM*$^+_{NSGA\text{-}II}$*-PART* are two winners from the first two comparisons, we use these two approaches as the third comparison pair to find the best for both case studies.

Table D-7 indicates that in 82% (i.e., 74/90) of the total comparisons, *SBRM*$^+_{NSGA\text{-}II}$*-C45* significantly outperformed *SBRM*$^+_{NSGA\text{-}II}$*-PART* whereas in only 3% (i.e., 3/90) *SBRM*$^+_{NSGA\text{-}II}$*-PART* significantly performed better than *SBRM*$^+_{NSGA\text{-}II}$*-C45* for the Cisco case study. Similarly, for Jitsi, in 80% (i.e., 82/102) of the total comparisons, *SBRM*$^+_{NSGA\text{-}II}$*-PART* significantly outperformed *SBRM*$^+_{NSGA\text{-}II}$*-C45* while in only 2% (i.e., 2/102) of the total comparisons *SBRM*$^+_{NSGA\text{-}II}$*-C45* significantly performed better than *SBRM*$^+_{NSGA\text{-}II}$*-PART*.

Since *SBRM*$^+_{NSGA\text{-}II}$*-C45* significantly outperformed other the other three *SBRM*⁺ approaches in terms of MLQMs for the Cisco case study and *SBRM*$^+_{NSGA\text{-}II}$*-PART* for the Jitsi case study, we can conclude that given the default parameter settings for both the machine learning algorithms and the search algorithms, *SBRM*$^+_{NSGA\text{-}II}$*-C45* and *SBRM*$^+_{NSGA\text{-}II}$*-PART* produce better rules with respect to MLQMs for the Cisco and Jitsi case studies, respectively.

6.6 Correlation Analysis (RQ6)

To answer RQ6, we compute the correlation coefficients (ρ) and *p*-values using Non-Parametric Spearman's test for all the MLQMs in correlation to the average fitness values (*AFV*) for the three individual objectives (i.e., *AFV-O1*, *AFV-O2* and *AFV-O3*), overall average fitness values (*OAFV*) and six quality indicators corresponding to both case studies. Through correlation analysis, we intend to test our hypothesis, i.e., *Accuracy, Precision*, *Recall*, and *FMeasure* have positive correlations with *HV* and negative correlations with the average fitness values and the other five quality indicators whereas *MAE*, *RMSE*, *RAE*, and RRSE are negatively correlated with *HV* and positively correlated with the average fitness values and the other five quality indicators (Section 5.1.5). The results of RQ6 are summarized in Table D-8 for both Cisco and Jitsi case studies whereas the detailed results can be found in [103].

Table D-8. Summary of the correlation analysis' results (RQ6) *

Case study	$SBRM^{+}_{NSGA\text{-}II}$-C45			$SBRM^{+}_{NSGA\text{-}III}$-C45			$SBRM^{+}_{NSGA\text{-}II}$-PART			$SBRM^{+}_{NSGA\text{-}III}$-PART		
	HS	HR	NS	HS	HR	NS	HS	HR	NS	HS	HR	NS
Cisco	28/170	11/170	131/170	30/140	52/140	58/140	25/140	44/140	71/140	39/140	11/140	90/140
Jitsi	91/170	11/170	68/170	60/170	16/170	94/170	74/170	16/170	80/170	29/170	22/170	119/170

*HS= Hypothesis satisfied, HR= Hypothesis rejected, NS= Not significant

As shown in Table D-8, for the Cisco case study, 23% (i.e., 39/170), 59% (i.e., 82/140), 49% (i.e., 69/140), and 36% (i.e., 50/140) of the total correlations are significant for *$SBRM^{+}_{NSGA\text{-}II}$-C45*, *$SBRM^{+}_{NSGA\text{-}III}$-C45*, *$SBRM^{+}_{NSGA\text{-}II}$-PART*, and *$SBRM^{+}_{NSGA\text{-}III}$-PART* respectively, where 72% (i.e., 28/39), 37% (i.e., 30/82), 36% (i.e., 25/69), and 78% (i.e., 39/50) of significant correlations satisfy our hypothesis (Section 5.1.5). Similarly, for the Jitsi case study, 60% (i.e., 102/170), 45% (i.e., 76/170), 53% (i.e., 90/170), and 30% (i.e., 51/170) of the total correlations are significant for *$SBRM^{+}_{NSGA\text{-}II}$-C45*, *$SBRM^{+}_{NSGA\text{-}III}$-C45*, *$SBRM^{+}_{NSGA\text{-}II}$-PART*, and *$SBRM^{+}_{NSGA\text{-}III}$-PART* respectively, where 89% (i.e., 91/102), 79% (i.e., 60/76), 82% (i.e., 74/90), and 57% (i.e., 29/51) of significant correlations satisfy our hypothesis (Section 5.1.5).

6.7 Trend Analysis of the Quality of Rules Across the Iterations (RQ7)

To answer RQ7, we study the variation in the quality of rules in terms of MLQMs across the iterations (from *iteration-1* to *iteration-5*) for the *$SBRM^{+}$* approaches for both case studies. To do so, we plotted the scatter plots and fitted Linear Regression lines for all the MLQMs. The results of the trend analysis are summarized below, and the plotted graphs are provided in [103].

Table D-9. Summary of trend analysis' results (RQ7) *

Case study	$SBRM^{+}_{NSGA\text{-}II}$-C45			$SBRM^{+}_{NSGA\text{-}III}$-C45			$SBRM^{+}_{NSGA\text{-}II}$-PART			$SBRM^{+}_{NSGA\text{-}III}$-PART		
	IT	**DT**	**ST**	**IT**	**DT**	**ST**	**IT**	**DT**	**ST**	**IT**	**DT**	**ST**
Cisco	15/17	1/17	1/17	10/17	0/17	7/17	14/17	0/17	3/17	10/17	2/17	5/17
Jitsi	14/17	0/17	3/17	16/17	1/17	0/17	17/17	0/17	0/17	14/17	0/17	3/17

*IT= Increasing trend of the quality of rules, DT= Decreasing trend of the quality of rules, ST= Straight line (no change in the quality of rules)

As shown in Table D-9, for both case studies, we observed an increasing trend of quality of rules in terms of the majority (81%, i.e., 110/136) of the MLQMs for all the four *$SBRM^{+}$* approaches across the iterations. Also, for both case studies, we witnessed a slightly decreasing trend in only 3% (i.e., 4/136) of MLQMs for all the four *$SBRM^{+}$* approaches whereas in the remaining 16% (i.e., 22/136), we observed a straight line. Note that the quality of rules in terms of the MLQMs increases if values of error related MLQMs (i.e., *MAE*, *RAE*, *RMSE*, and *RRSE*) decrease and other MLQMs increase.

6.8 Cost of Applying Search to Generate Configurations (RQ8)

To answer RQ8, we calculated the average time required by NSGA-II (in *$SBRM^{+}_{NSGA\text{-}II}$-C45* and *$SBRM^{+}_{NSGA\text{-}II}$-PART*), NSGA-III (in *$SBRM^{+}_{NSGA\text{-}III}$-C45* and *$SBRM^{+}_{NSGA\text{-}III}$-PART*), and RS (in *$RBRM^{+}$-C45* and *$RBRM^{+}$-PART*) to generate configurations per iteration (i.e., *ATPI*) as well as per cycle (i.e., *ATPC*) (Section 5.1.3). Table D-10 shows the average time required by *$SBRM^{+}_{NSGA\text{-}}$*

II-C45, *SBRM*$^+$*NSGA-III-C45*, *SBRM*$^+$*NSGA-II-PART*, *SBRM*$^+$*NSGA-III-PART*, *RBRM*$^+$*-C45*, and *RBRM*$^+$*-PART* to generate configurations per iteration and per cycle for both of the case studies.

Table D-10. Average time (minutes) required for generating configurations

Case Study	Metric	SBRM$^+$NSGA-II-C45	SBRM$^+$NSGA-III-C45	RBRM$^+$-C45	SBRM$^+$NSGA-II-PART	SBRM$^+$NSGA-III-PART	RBRM$^+$-PART
Cisco	ATPI	22	3224	18	23	8765	22
	ATPC	108	16118	90	116	43824	108
Jitsi	ATPI	32	22527	40	10	10179	21
	ATPC	159	112636	199	52	50896	103

From Table D-10, we can observe that the costs of generating configurations using *SBRM*$^+$*NSGA-II-C45*, *SBRM*$^+$*NSGA-II-PART*, *RBRM*$^+$*-C45*, and *RBRM*$^+$*-PART* are quite comparable. However, *SBRM*$^+$*NSGA-III-C45* and *SBRM*$^+$*NSGA-III-PART* took significantly more time than the others, because NSGA-III is significantly slower than NSGA-II and RS. Also, NSGA-III produces only 92 solutions for the three objective problems regardless of its population size [80], thus, we executed it multiple times to get 500 configuration solutions corresponding to each iteration. We used a fixed number of fitness evaluations instead of time budget as the termination criterion of the search because 1) different frameworks for multi-objective optimization with metaheuristics (e.g., jMetal [53]) use fitness evaluations instead of time budget; 2) A fixed number of fitness evaluations are widely applied in SBSE [104-107]; 3) We used 50,000 fitness evaluations as termination criterion, because we were able to obtain good results in our earlier studies involving industrial datasets [48, 105]; and 4) We think comparing search algorithms based on fixed time is biased towards faster algorithms, as a slower one gets less chance to evolve towards a better solution, particularly in the context where the time cost of executing an approach is not important which is the case of applying our approach.

From Table D-10, we can also notice that *SBRM*$^+$*NSGA-II-C45*, *SBRM*$^+$*NSGA-III-C45*, and *RBRM*$^+$*-C45* took more time than *SBRM*$^+$*NSGA-II- PART*, *SBRM*$^+$*NSGA-III- PART*, and *RBRM*$^+$*- PART* respectively. This can be explained as C4.5 produced lengthier rules (i.e., more predicates) than PART (Table D-11). Thus, approaches producing lengthier rules have a higher cost of calculating fitness values and consequently higher execution time. On average, C4.5 produced 1.7 and 2 more predicates per rule than PART for the Cisco and Jitsi case studies, respectively.

Table D-11. Average number of predicates for the Cisco and Jitsi case studies

Approach	Cisco		Jitsi	
	Avg. predicates per rule	Avg. predicates per run	Avg. predicates per rule	Avg. predicates per run
SBRM$^+$NSGA-II-C45	5.2	14420	4.6	100095
SBRM$^+$NSGA-II-PART	3.6	17102	2.7	22764
SBRM$^+$NSGA-III-C45	5.7	14853	5.5	134451
SBRM$^+$NSGA-III-PART	4.0	20668	2.7	20498
RBRM$^+$-C45	5.6	21763	4.0	106454
RBRM$^+$-PART	3.9	26632	2.6	25208

6.9 Discussion

For RQ1, we noticed that NSGA-II and NSGA-III significantly outperformed RS in terms of all the fitness values and majority of the quality indicators for both of the case studies (Section 6.1).

This can be simply explained as NSGA-II and NSGA-III generate and select better solutions using operators such as mutation and crossover. We also noticed that RS performed better than NSGA-II and NSGA-III in terms of *GS* (representing the diversity of obtained solutions) for 15/24 and 17/24 comparisons for the Cisco and Jitsi case studies, respectively. This is because 1) for our problem, higher convergence to the objectives (e.g., *Objective-1* avoids generating configurations satisfying high confidence rules with normal states) may reduce the search space to be explored, which consequently affects diversity negatively; and 2) RS explores the search space more uniformly as compared to other algorithms [108], thus, the solutions produced by RS has high diversity but low convergence as shown by the results of RQ1 (Section 6.1).

For RQ2, we observed that in 7 out of 8 comparisons for both of the case studies, *SBRM*$^+$ performed significantly better than the two *RBRM*$^+$ approaches in terms of the majority of MLQMs whereas in one of the 8 comparisons there was no significant difference observed between the two approaches (i.e., *SBRM*$^+_{\text{NSGA-III}}$-*C45* and *RBRM*$^+$-*C45*) (Section 6.2). *SBRM*$^+_{NSGA\text{-}II}$-*C45*, *SBRM*$^+_{NSGA\text{-}III}$-*C45*, *SBRM*$^+_{NSGA\text{-}II}$-*PART*, and *SBRM*$^+_{NSGA\text{-}III}$-*PART* have achieved an ARI up to 27%, 22%, 18%, and 21% for the Cisco case study respectively (Section 6.3). Similarly, for the Jitsi case study, *SBRM*$^+_{NSGA\text{-}II}$-*C45*, *SBRM*$^+_{NSGA\text{-}III}$-*C45*, *SBRM*$^+_{NSGA\text{-}II}$-*PART*, and *SBRM*$^+_{NSGA\text{-}III}$-*PART* have achieved an ARI up to 28%, 4%, 8%, and 12% respectively (Section 6.3). This is because the three objectives use previously mined rules for guiding the search to generate configurations that increase the support of the correct rules and filter out incorrect ones. In addition, the operators of NSGA-II and NSGA-III help *SBRM*$^+$ to converge faster than *RBRM*$^+$.

For RQ4, NSGA-III significantly outperformed NSGA-II in terms of fitness values and the quality indicators in most of the cases. For RQ5, *SBRM*$^+_{\textbf{NSGA-II}}$-*C45* significantly outperformed other three *SBRM*$^+$ approaches in producing better quality rules in terms of MLQMs for the Cisco case study and *SBRM*$^+_{\text{NSGA-II}}$-*PART* for the Jitsi case study. This deviation in the results for the two case studies could be explained as follow: 1) The number of categorical configurable parameters is different (15 for Cisco and 27 for Jitsi); 2) The maximum number of possible configurations for a categorical configuration parameter is different (4 for Cisco and 16 for Jitsi); 3) The total number of configurable parameters is different (27 for Cisco and 39 for Jitsi); and 4) the configuration spaces are different (1.03e^{33} for Cisco and 6.54e^{60} for Jitsi). The categorical parameters are of more importance because satisfying the predicates with the categorical parameters in the rules is more difficult than satisfying the predicates with numerical parameters. This is because usually in the rules, predicates with numerical parameters allow a large number of values to satisfy the predicates, whereas satisfying predicates with categorical parameters requires exact values from predefined candidate values. The different characteristics of the case studies could make different algorithms suitable for mining the rules. Based on the characteristics of the two selected case studies and their corresponding results, we can argue that PART is a preferred choice to integrate with NSGA-II (i.e., *SBRM*$^+_{NSGA\text{-}II}$-*PART*) for mining rules for a relatively larger case study whereas C4.5 (*SBRM*$^+_{NSGA\text{-}II}$-*C45*) is a better choice in the case of a smaller sized case study. Nevertheless, these results cannot be generalized based on the evaluation of merely two case studies. Besides, the selection of the machine learning algorithms and their parameter settings are usually application dependent. Thus, generalizing the results further requires a much larger scale empirical evaluation with more case studies.

From the *correlation analysis* for RQ6, we noticed that the majority of cases satisfy our hypothesis (Section 5.1.5) that the overall quality of rules in terms of MLQMs improves by reducing the average fitness values and quality indicators (except for *HV*) and increasing *HV*. However, smaller average fitness values and quality indicators (except for *HV*) and larger *HV* do not mean that all the MLQMs will always be improved, as we observed several cases (e.g., correlations of *FF-Precision* and *CF-Recall* with *AFV-O3* corresponding to $SBRM^{+}_{NSGA\text{-}II}$-*PART* for the Cisco case study, correlations of *CC-Recall* and *CC-FMeasure* with *AFV-O1* corresponding to $SBRM^{+}_{NSGA\text{-}II}$-*PART* for the Jitsi case study) that reject our hypothesis. It is quite possible that certain MLQMs are affected negatively due to several reasons, 1) *Objective-1* avoids generating the configurations satisfying high confidence rules with *ConnectedConnected* class due to which mining algorithm will give more preference to other classes (i.e., *FailedFailed*, *FailedConnected*, and *ConnectedFailed*), therefore, MLQMs such as *CC-Recall* and *CC-FMeasure* may decrease with the decrement in *AFV-O1* as it did for the Jitsi case study; 2) *Objective-2* and *Objective-3* generate configurations satisfying low confidence (i.e., higher violation and lower support) rules with normal and abnormal states, which increase the violation of low confidence rules that may affect MLQMs negatively in certain cases (e.g., when violation of rules increased but not enough to remove them from rule set) as it did for the Cisco case study. In such cases, MLQMs may decrease with the reduction in *AFV-O2* and *AFV-O3*.

For RQ7, we noticed an increasing trend of the quality of rules based on the majority of MLQMs for all the four $SBRM^{+}$ approaches for both case studies. This is because, in each new iteration, we refined the rules by generating the configurations based on the rules mined from the previous iteration and mining a new set of refined rules, which improves the quality based on MLQMs in each new iteration. Thus, the incremental, iterative process refines rules across iterations, and the number of iterations does have an impact on the results. For RQ8, the best performing $SBRM^{+}_{NSGA\text{-}II}$-*C45* took 108 minutes for the Cisco case study whereas $SBRM^{+}_{NSGA\text{-}II}$-*PART* took 52 minutes corresponding to the Jitsi case study, for generating configurations for a complete cycle, which is acceptable as it is a one-time process.

Furthermore, to know the distribution of the mined rules associated with the four system states (*ConnectedConnected*, *FailedFailed*, *ConnectedFailed*, and *FailedConnected*) in the five iterations for both case studies, we plotted stacked column plots. Note, we have also presented the distribution of the rules for the iteration zero, to be complete. Figure D-9 presents the average numbers of rules mined with the different approaches for the Cisco case study. From Figure D-9, we can see that $RBRM^{+}$-*C45* ($RBRM^{+}$-*PART*) produced more rules than $SBRM^{+}_{NSGA\text{-}II}$-*C45* and $SBRM^{+}_{NSGA\text{-}III}$-*C45* ($SBRM^{+}_{NSGA\text{-}II}$-*PART* and $SBRM^{+}_{NSGA\text{-}III}$-*PART*) in all the five iterations except that $SBRM^{+}_{NSGA\text{-}II}$-*C45* produced slightly more rules than $RBRM^{+}$-*C45* in *iteration-1* and *iteration-2*. We can also notice that no rules were produced for *FailedConnected* in the first three iterations and significantly fewer numbers of rules produced for *FailedConnected* (to compare with the other categories) in only *iteration-4* and *iteration-5*.

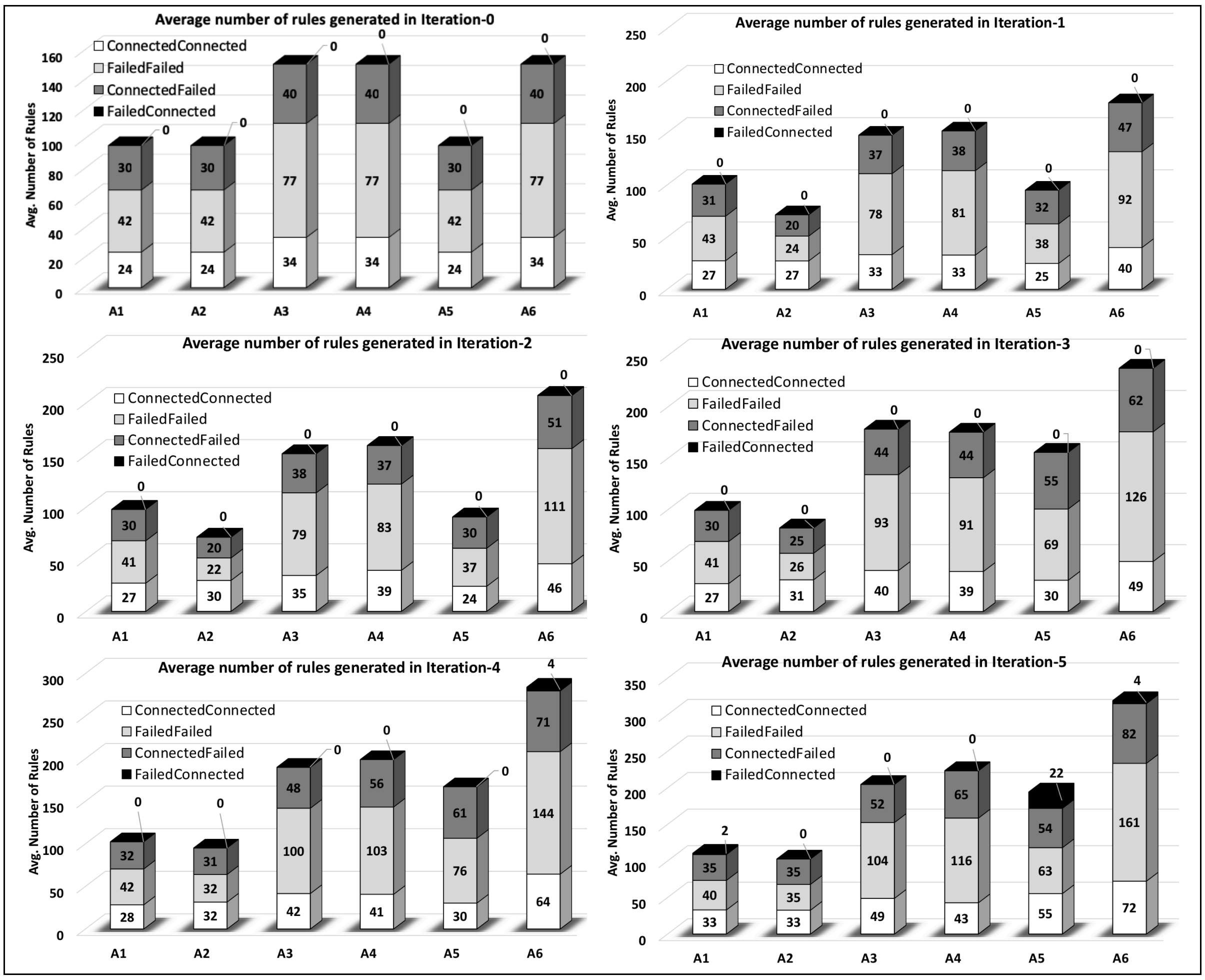
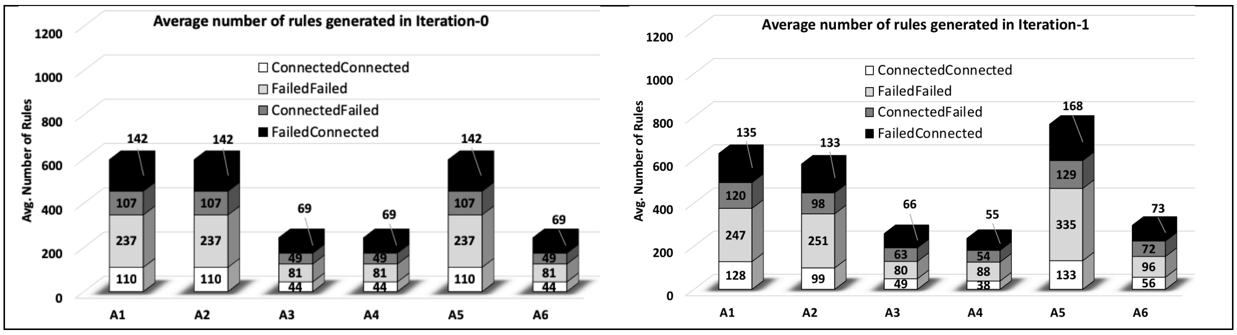


Figure D-9. Average numbers of rules mined in each iteration for the Cisco case study*

*A_1= SBRM$^+_{NSGA\text{-}II}$-C45, A_2= SBRM$^+_{NSGA\text{-}III}$-C45, A_3= SBRM$^+_{NSGA\text{-}II}$-PART, A_4= SBRM$^+_{NSGA\text{-}III}$-PART, A_5= RBRM$^+$-C45, A_6= RBRM$^+$-PART

Figure D-10 presents the average numbers of rules mined for the Jitsi case study. From the figure, we can observe that *RBRM$^+$-C45* (*RBRM$^+$-PART*) produced more rules than *SBRM$^+_{NSGA\text{-}II}$-C45* and *SBRM$^+_{NSGA\text{-}III}$-C45* (*SBRM$^+_{NSGA\text{-}II}$-PART* and *SBRM$^+_{NSGA\text{-}III}$-PART*) in all the iterations just as for the Cisco case study. For both of the case studies, we observed that the *SBRM$^+$* approaches produced less number of rules than the two *RBRM$^+$* approaches. This is because the three objectives refine the rules by removing low confidence incorrect rules and the search operators (i.e., mutation, crossover, and selection) help *SBRM$^+_{NSGA\text{-}II}$-C45* and *SBRM$^+_{NSGA\text{-}II}$-PART* to get optimal configurations in terms of three objectives.



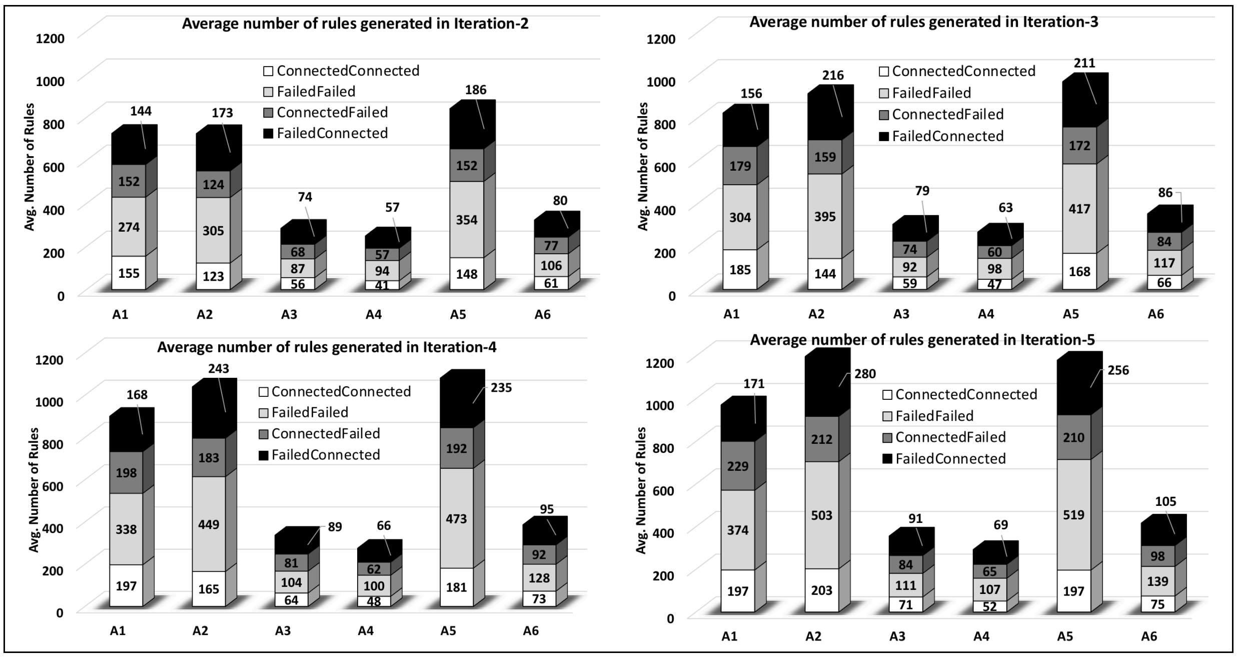


Figure D-10. Average numbers of rules mined in each iteration for the Jitsi case study*

*A$_1$= SBRM$^+_{\text{NSGA-II}}$-C45, A$_2$= SBRM$^+_{\text{NSGA-III}}$-C45, A$_3$= SBRM$^+_{\text{NSGA-II}}$-PART, A$_4$= SBRM$^+_{\text{NSGA-III}}$-PART, A$_5$= RBRM$^+$-C45, A$_6$= RBRM$^+$-PART

Figure D-11 shows the distribution of rules with respect to normal and abnormal system states, obtained using *SBRM*$^+$ for both case studies. From Figure D-11, one can see that the majority of the rules produced are rules with abnormal system state, which is expected because *SBRM*$^+$ focused on generating invalid configurations.

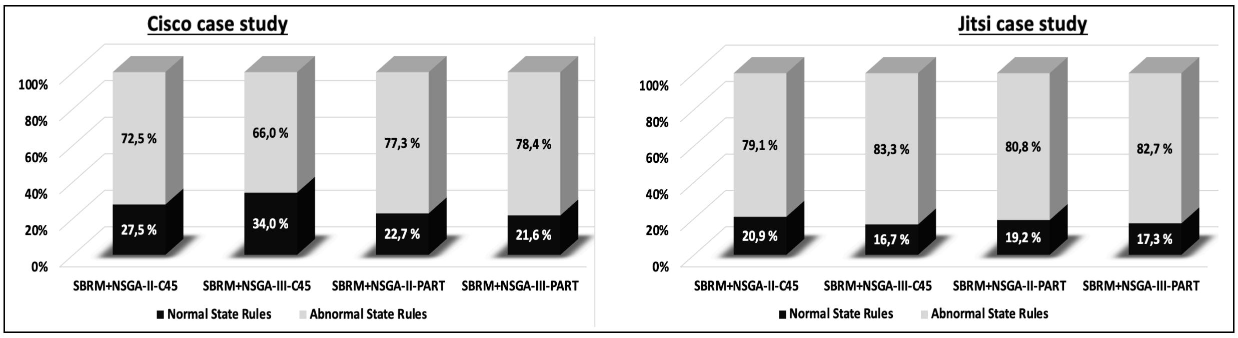


Figure D-11. Rules distribution w.r.t. system states

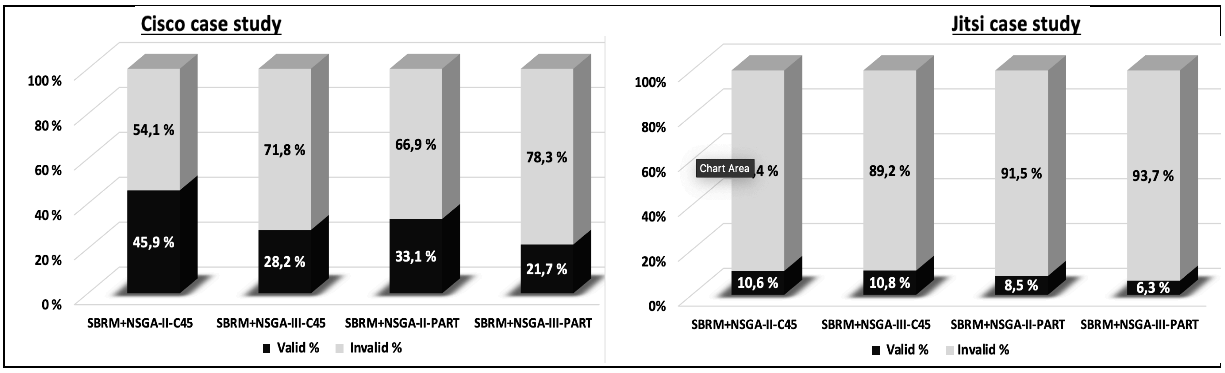
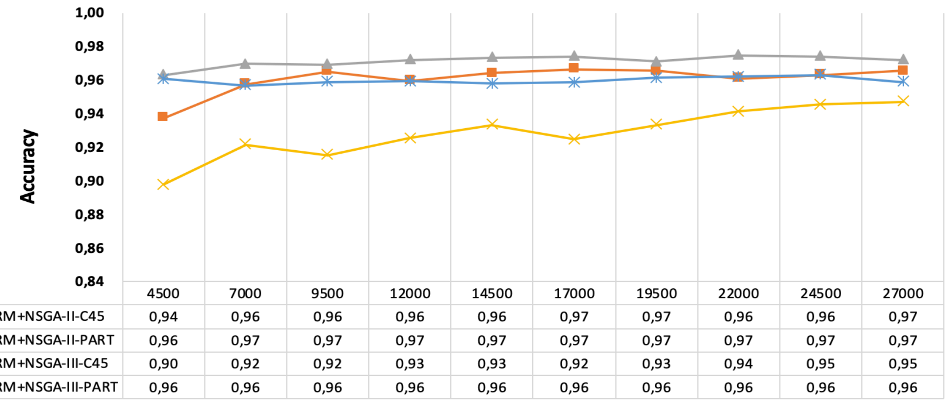


Figure D-12. Average numbers configurations with of valid and invalid system states per run

To see the distribution of configurations with respect to the corresponding system states (i.e., valid or invalid), we collected the statistics about configurations generated using *SBRM*$^+$ for both case studies, which are shown in Figure D-12. However, it is worth mentioning that the distribution of generated configurations is greatly influenced by the input rules provided to the search algorithms.

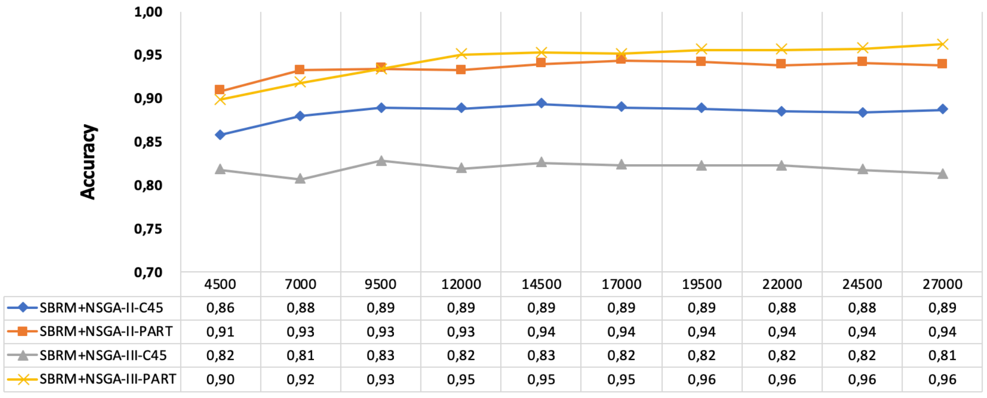
Moreover, we intend to assess if adding more iterations increases the quality of rules significantly. Due to high execution cost of the experiments, we combined configurations from 10 runs of already executed experiments and mine rules to see the trend of quality improvement of rules with respect to the dataset size. More specifically, first, we mine the rules using configurations of the first run and then incrementally add the configurations from other nine runs and mine the rules. Note, for the first run we used all the 4500 configurations whereas, for other 9 runs, we have added only 2500 configurations per run (i.e., for five iterations) because the initial 2000 configurations (i.e., randomly generated) are common across all the runs. To show the trend, we plotted the MLQMs against the number of instances (i.e., configurations) in the dataset. Due to limited space, we have selected *Accuracy* as a representative MLQM to illustrate the trend (Figure D-13 and Figure D-14).



	4500	7000	9500	12000	14500	17000	19500	22000	24500	27000
SBRM+NSGA-II-C45	0,94	0,96	0,96	0,96	0,96	0,97	0,97	0,96	0,96	0,97
SBRM+NSGA-II-PART	0,96	0,97	0,97	0,97	0,97	0,97	0,97	0,97	0,97	0,97
SBRM+NSGA-III-C45	0,90	0,92	0,92	0,93	0,93	0,92	0,93	0,94	0,95	0,95
SBRM+NSGA-III-PART	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96

Figure D-13. Accuracy vs. number of instances in the dataset for Cisco

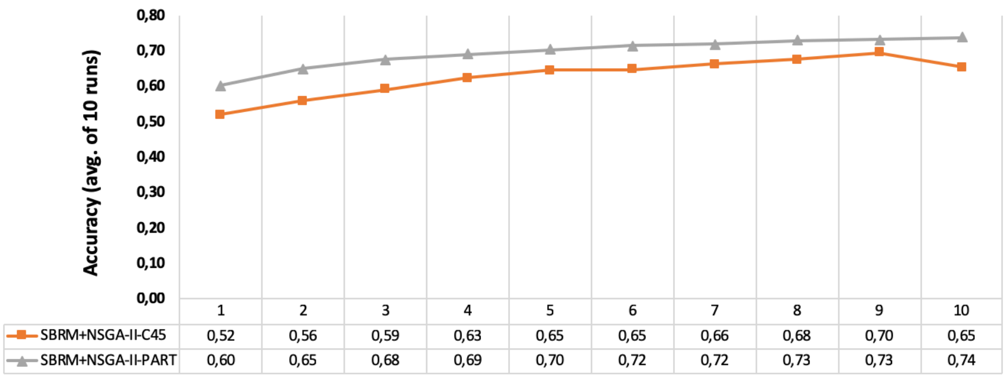
As shown in Figure D-13 and Figure D-14, for both of the case studies, there is an improvement in the quality of rules, but not significant. From 4500 to 22,500 instances (i.e., configurations), we get up to 5% of improvement for the Cisco case study and 6% for the Jitsi case study. Note, for the other MLQMs, we also observed similar results.



	4500	7000	9500	12000	14500	17000	19500	22000	24500	27000
SBRM+NSGA-II-C45	0,86	0,88	0,89	0,89	0,89	0,89	0,89	0,88	0,88	0,89
SBRM+NSGA-II-PART	0,91	0,93	0,93	0,93	0,94	0,94	0,94	0,94	0,94	0,94
SBRM+NSGA-III-C45	0,82	0,81	0,83	0,82	0,83	0,82	0,82	0,82	0,82	0,81
SBRM+NSGA-III-PART	0,90	0,92	0,93	0,95	0,95	0,95	0,96	0,96	0,96	0,96

Figure D-14. Accuracy vs. number of instances in the dataset for Jitsi

We assess the trend of quality of rules against different dataset sizes. However, one can argue that we added the configurations generated using rules from iteration zero to iteration-4 and adding configurations generated using rules from iteration 5 and onwards would have improved the quality of rules significantly. To cater this argument, we selected the best performing approaches $SBRM^{+}_{NSGA\text{-}II}$-*C45* for the Cisco case study and $SBRM^{+}_{NSGA\text{-}II}$-*PART* for the Jitsi case study and conducted the experiment with these two approaches to obtain five more iterations (i.e., in total 10 iterations) with the Jitsi case study. This is done only for the Jitsi case study because the experiment can be run on a cluster. However, for the Cisco case study, running the experiment needs dedicated hardware equipment and we cannot run the experiment in parallel due to the limited number of VCSs available, which makes the experiment extremely time-consuming. Figure D-15 shows the average *Accuracy* (i.e., calculated as the average of 10 runs for each iteration) across the 10 iterations. From Figure D-15, we can observe an improvement in the quality of rules across the iterations, however, we got an improvement of 4% at maximum for any approach from iteration-5 to iteration-10. On the other hand, when looking at the improvement from iteration-1 to iteration-5, we got an improvement of 13% for $SBRM^{+}_{NSGA\text{-}II}$-*C45* and 10% for $SBRM^{+}_{NSGA\text{-}II}$-*PART*. Thus, it would be fair to say that after a number of iterations (e.g., five in our case), the improvement will be very slow. This suggests that using a fixed number of iterations is a practical and wise approach to terminate the process.



	1	2	3	4	5	6	7	8	9	10
■ SBRM+NSGA-II-C45	0,52	0,56	0,59	0,63	0,65	0,65	0,66	0,68	0,70	0,65
▲ SBRM+NSGA-II-PART	0,60	0,65	0,68	0,69	0,70	0,72	0,72	0,73	0,73	0,74

Figure D-15. Average accuracy across 10 iterations for the Jitsi case study

6.10 Threats to Validity

In Section 6.10.1, we discuss threats to the internal validity followed by threats to the construct validity in Section 6.10.2. We discuss threats to the conclusion validity and external validity in Section 6.10.3 and Section 6.10.4, respectively.

6.10.1 Internal Validity

Threats to the internal validity exist when the results are influenced by the internal factors such as parameter settings [109]. The first threat to the internal validity is the selection of search algorithms in our study. To mitigate this threat, we selected the most widely used NSGA-II algorithm, which has shown promising results in different contexts [45, 78]. Moreover, we have selected a relatively new multi-objective search algorithms, i.e., NSGA-III, which also has good performance on addressing many objective problems [82]. The second threat is the selection of algorithms for rule mining. We selected PART as it has been proven to be more effective than many well-known algorithms [40, 59] and C4.5, the most popular algorithm in industry and the research community [86, 110]. The third threat is the selection of parameter settings for the selected search algorithm. To mitigate this threat, we used default parameter settings, which have exhibited promising results [58]. Similarly, for the machine-learning algorithms, we also used the default parameters settings, which perform reasonably well [74, 86]. Another threat is the selection of the *Confidence* measure for calculating fitness values, as there exist other measures (e.g., *Lift*). We acknowledge that this is a threat to the internal validity and dedicated experiments are needed for further investigation.

6.10.2 Construct Validity

Threats to the construct validity exist when the measurement metrics do not sufficiently cover the concepts they are supposed to measure [62, 109]. To mitigate this threat, we compared different approaches using the same comprehensive set of measures: fitness values, quality indicators, and 17 MLQMs, which are commonly used in the literature [51, 86, 111].

6.10.3 Conclusion Validity

Threats to the conclusion validity concern with the factors influencing the conclusion drawn from the results of the experiment [112]. The most probable threat to the conclusion validity is due to the random variation inherent in search algorithms. To minimize this threat, we repeated the experiment 10 times (i.e., total 50 runs of each search algorithm) to reduce the effect caused by randomness, as recommended in [100, 113]. Moreover, we also applied the Mann-Whitney test to determine the statistical significance of the results and the Vargha and Delaney $\hat{A}_{12}$ statistics as the effect size measure, which are recommended for randomized algorithms [100, 113].

6.10.4 External Validity

The external validity concerns with the generalization of the experiment results to other contexts [109]. The threat to the external validity for our experiment is the case studies selected for the evaluation. In our study, we used a real-world case study (i.e., Cisco Video Conferencing Systems) and an open source case study Jitsi of different sizes. Furthermore, one can argue that the complexity of case studies (i.e., a large number of configurable parameters and system states) may affect the performance of proposed approach. We would like to argue that multi-objective search algorithms such as NSGA-II and NSGA-III have been applied to problems of different complexity, and they have proven to be quite effective [60-62, 76, 82, 96]. However, higher dimensional datasets (more attributes) for complex case studies, may reduce the performance (e.g., accuracy, precision) of machine learning algorithms but the impact will be the same for both *SBRM*$^+$ and *RBRM*$^+$, as both approaches employ a machine learning algorithm.

7 Related Work

Search algorithms have been used to solve many problems in the context of PLE [60-62, 76, 96]. In this paper, we also combined the search with machine learning techniques to mine the rules in the context of PLE. The related work to this research stream focuses on existing studies presenting the approaches to mine the rules in the context of PLE. In Section 5.1, we discuss dedicated approaches that focus on mining rules from different artifacts (e.g., source code, configuration file, feature model) of product lines. Furthermore, in Section 5.2, we discuss approaches such as feature extraction, feature construction and feature recommendation, which mine crosstree constraints. Finally, in Section 7.3, we summarize the related work and compare it with our work.

7.1 Dedicated Rule Mining Approaches

The work in [37] applies Binary Decision Tree-J48 (machine learning algorithm) to infer the constraints from a set of randomly generated product configurations. To classify the configurations as faulty and non-faulty, a computer vision algorithm was used as an oracle. To validate the approach, it was applied to an industrial video generator product line. Rules were evaluated based on expert's opinion and machine-learning measurements such as Precision and Recall. Results show that on average 86% Precision and 80% Recall rate can be achieved using the proposed approach.

In [63], an approach for mining the crosstree binary constraints (i.e., requires, excludes) corresponding to a feature model is presented. The approach takes a feature model as input containing the features, their descriptions, and some known crosstree binary constraints. First, it trains LIBSVM classifier (an extension of support vector machine) with existing crosstree binary constraints where the parameters of the classifier are optimized using the genetic algorithm to minimize the error rate of the classifier. Second, it extracts all the feature pairs, and finally, the optimized classifier finds the candidate features of binary constraints. The approach was validated using two feature models collected from SPLOT repository. Results show that rules with high *Recall* (i.e., close to 100%) and the variable low *Precision* (on average 42%) can be achieved using proposed approach.

In [64], another approach is presented for mining the crosstree constraints. It constructs configuration matrix (i.e., product-features matrix) from configuration files and extracts crosstree constraints using an association rule mining technique (i.e., Apriori algorithm). Rules are pruned using minimum support and minimum confidence thresholds. The approach was evaluated using a large-scale industrial software product line for embedded systems. The evaluation shows that a large number of rules with variable support (i.e., 80% to 99%) and confidence (i.e., 90% to 100%) can be identified. The majority of the rules were identified with support ranging from 80% to 85%.

The work in [65] presents an approach to extract configuration constraints from existing C codebases using static analysis. It uses build time errors (e.g., preprocessor, parser, type, and link errors) as the oracle to classify the low-level system configurations (i.e., build and code files) and mine the constraints. To assess the accuracy of extracted rules, they were compared with the existing constraints specified in developer's created variability models. The approach was validated using four open-source case studies (uClibc, BusyBox, eCos, and the Linux kernel). Results show that up to 19% of the total constraints can be recovered automatically from the source code, which assures successful build with the accuracy of 93%. In [38], an extension of [65] is presented in which the authors improved the static analysis and increased the recoverability rate by 9%. Additionally, an empirical study is also presented that identifies the sources of constraints.

Table D-12. Characteristics of existing rule mining approaches*

Reference	Topic	Input	Output	ML Technique for mining rules	Configurable parameter type	Data generation/ selection	# of classes	Evaluation	Case study
[37]	Configuration constraint extraction for PL	A FM and an oracle (computer vision algorithm)	A set of configuration constraints	Binary Decision Tree-J48 (implementation of C4.5)	Numerical and Categorical	Randomly	2	Precision, Recall, and expert opinion	A real-world PL of video generator
[63]	Constraint extraction for FM	A FM (features, feature description, and known binary crosstree constraints)	A set of crosstree constraints	LIBSVM classifier and genetic algorithm	Categorical	Randomly	3	Precision, Recall, and FMeasure	Two open source feature models of Weather Station and Graph PL
[64]	Constraint extraction for FM	Configuration files	A set of crosstree constraints	Apriori algorithm	Categorical	NA	NA	Support and confidence	An industrial PL of embedded systems
[65], [38]	Configuration constraint extraction	C code	A set of hierarchy and crosstree constraints	Static analysis (Build and code analysis)	Categorical	NA	NA	Accuracy measured in reference to the rules defined by the expert and recoverability	Four open source case studies (uClibc, BusyBox, eCos, and the Linux kernel)
[66]	Introducing probabilistic FM and provide a process to extract the crosstree	Formally defined FM inform of propositional formula	A set of crosstree constraints	Apriori algorithm and an algorithm presented in [114]	Categorical	NA	NA	Support and confidence	A small sized PL of Java applets

	constraints for FM								
[67], [69]	Feature extraction and constraint extraction for recommending features	Product descriptions available on internet and user requirements	FM along with crosstree constraints	CFP-growth algorithm and Apriori algorithm	Categorical	NA	NA	Support and confidence	A PL of remote collaboration and description of 20 PL collected from SoftPedia
[32]	Feature extraction and constraint extraction	Product description, product comparison metrics, manually added domain knowledge	Attributed FM with crosstree constraints	Proposed algorithms implemented in Scala and SAT4J solver for computation of or-group	Numerical and Categorical	Randomly	NA	Quality of the rules was not evaluated	242 configuration matrices generated randomly and a real-world case study of bestbuy.com
[70]	Feature extraction and constraint extraction	Product descriptions	FM with crosstree constraints	CFP-growth algorithm and Apriori algorithm	Categorical	NA	NA	Support, confidence, accuracy with respect to manually constructed FM	A PL of antivirus collected from SoftPedia
[115]	Feature extraction and recommendati on	Feature description and user requirements	Feature recommendation based on the association rules	Apriori algorithm	Categorical	NA	NA	Support and confidence	A large number of datasets collected from three repositories SoftPedia, SourceForge, and FreeCode

* FM= feature model, PL= product line, NA= Not applicable, ML= Machine learning, LDA= Latent Dirichlet Allocation

7.2 Non-Dedicated Rule Mining Approaches

The work in [116] reported a Systematic Literature Review (SLR) of 13 approaches for feature extraction from natural language requirements. The results of SLR show that hybrid natural language processing approaches are commonly used in the overall feature extraction process. Various clustering approaches from data mining and information retrieval are used to group the common features. Moreover, several approaches have also employed association mining techniques to discover the pattern of the features to recommend the relevant features to the stakeholders. In [66], an extension of feature model called probabilistic feature model is introduced. To extract crosstree constraints from existing formally defined products, a rule mining process is presented that uses an association mining technique (i.e., Apriori Algorithm) to mine the conjunctive association rule and an algorithm proposed in [114] to mine the Disjunctive association rules. The proposed mining process was applied to a small case study of Java Applets. Rules were evaluated based on machine-learning measurements (i.e., support and confidence).

In [67], an approach is proposed to model and recommend product features for any particular domain based on the product description provided by the domain expert. To mine association rules between product features, association rule mining techniques are applied to configuration matrix (i.e., product-features matrix). The proposed approach was validated with 20 different product categories using product descriptions available at SoftPedia. Hariri et al. [69] extended the work presented in [67]. In [69], different clustering algorithms used to cluster the features and construct products by feature matrix were compared. The evaluation was also improved by applying the approach on diverse domains as well as a large project of a software suite for remote collaboration. Results show that rules with different *Precision* and *Recall* rates can be mined according to the threshold set for the confidence.

The work in [32] presents an approach to synthesize attributed feature models (AFM) from a set of product descriptions in the form of tables (i.e., configuration matrix). An algorithm is proposed that uses implication graph and mutex graph constructed from configuration matrix to extract the crosstree constraints. For extracting the relational constraints defined on values of attributes, the algorithm uses domain knowledge or selects the boundary values of attributes randomly when domain knowledge is not provided. The approach was validated using random configuration matrices as well as a real-world case study. Results show that the proposed algorithm can be used to mine a large number of rules for large-scale case studies.

The work in [70] proposed an approach to construct a feature model automatically from informal product descriptions available over the Internet. To mine the implication rules of features, CFP-growth algorithm and Apriori algorithm are applied to configuration matrix (i.e., product-features matrix). The proposed approach was applied to a case study of antivirus software using the product descriptions available at SoftPedia.

In [115], an approach is proposed to extract the features from multiple web repositories, organize, analyze, and recommend the high-quality features to the stakeholders. The proposed approach first extracts the information from the Internet repositories and then builds feature ontologies by employing Latent Dirichlet Allocation and clustering. To mine the hidden relationships among software features and to recommend high-quality features to the stakeholders, the proposed approach employs the association rule mining technique (i.e., the

Apriori algorithm). The proposed approach is validated using a large number of datasets collected from three repositories (i.e., SoftPedia, SourceForge, and FreeCode).

7.3 Summary

In Table D-12, we summarize the existing rule mining techniques and highlight their characteristics. From Table D-12, one can see that, (1) all the techniques except [37] are focusing on mining binary crosstree constraints (requires and excludes) between different features of a product line or rules constraining the values of features' attributes in the case of [32]; (2) the majority of the approaches except two ([37] and [63]) are using unsupervised learning based association mining techniques such as Apriori algorithm and FP-growth algorithm; (3) none of the existing approaches have any sophisticated way to select/generate the configurations, and usually, configurations are generated/selected randomly or used existing configurations; (4) the majority of the approaches except two are focusing on only categorical type configurable parameters, however, [37] and [32] are also catering numerical configurable parameters; (5) all the existing approaches are using machine learning quality measurements such as Precision, Recall, Support, and Confidence; and 6) above all, none of the existing approaches are mining the rules for interacting products within/across the product lines.

In contrast to the existing rule mining techniques, we have proposed an incremental and iterative approach in which we generate the configurations smartly and feed the configurations to the machine-learning tool and apply supervised learning based rule mining techniques (i.e., PART and C45), to mine the rules between configurable parameters and system behaviors of interacting products across product lines. The innovative part of our approach is the data generation strategy and incremental, iterative nature, which helps to achieve rules with higher quality as compared to randomly selected configurations based approaches. To generate the configurations, we defined three objectives (Section 3.2) and combined them with the search algorithms (i.e., NSGA-II and NSGA-III). To evaluate the quality of rules, we used machine learning quality measurements, which are also used by existing rule-mining approaches in the literature.

8 Conclusion and Future Work

Today, systems are being developed by integrating multiple products within/across the product lines that communicate with each other through different communication mediums (e.g., the Internet). The runtime behavior of these systems does not only depend on product configurations, but also on the communication medium. To identify the invalid configurations where these products may fail to communicate, we mine the Cross-Product Line (CPL) rules. To do so, in our previous work, we proposed an incremental and iterative approach named as Search-Based Rule Mining (*SBRM*), in which we combined the widely used multi-objective search algorithm (NSGA-II) with the machine learning algorithm (PART). To use the search in the rule mining process, we defined three objectives and integrated them with the multi-objective optimization algorithm NSGA-II. In this paper, we improved the previously proposed *SBRM* (named as *SBRM*$^+$) and incorporated two multi-objective search algorithms (i.e., NSGA-II and NSGA-III) and two machine learning algorithms (i.e., C4.5 and PART) to mine the rules. Moreover, in *SBRM*$^+$, we also integrated a clustering algorithm (i.e., *k*-means) to classify the CPL rules as high or low confidence rules, which are used for defining the three objectives to guide the search.

To evaluate the *SBRM*$^+$ (*SBRM*$^+_{NSGA\text{-}II}$*-C45*, *SBRM*$^+_{NSGA\text{-}III}$*-C45*, *SBRM*$^+_{NSGA\text{-}II}$*-PART*, and *SBRM*$^+_{NSGA\text{-}III}$*-PART*), we conducted experiments using two real case studies (Cisco and Jitsi) and performed three types of analyses: *difference analysis*, *correlation analysis*, and *trend analysis*. *Difference analysis* shows that *SBRM*$^+$ approaches performed significantly better than two random search-based approaches (*RBRM*$^+$*-C45* and *RBRM*$^+$*-PART*) in terms of the fitness values, six quality indicators, and 17 MLQMs corresponding to both case studies. Among the four *SBRM*$^+$ approaches, *SBRM*$^+_{NSGA\text{-}II}$*-C45* produced the highest quality rules based on MLQMs for the Cisco case study and *SBRM*$^+_{NSGA\text{-}II}$*-PART* for the Jitsi case study. *Correlation analysis* suggests that in most of the cases lower average fitness values and quality indicators (except for *HV*) and higher *HV* mean overall higher quality rules in terms of MLQMs. Furthermore, *trend analysis* shows an increasing trend of the quality of rules in terms of MLQMs for all the four *SBRM*$^+$ approaches across the five iterations.

Our future work includes: (1) Evaluating the performance of different search algorithms for generating configurations and mining the rules; (2) Using different parameter settings for machine learning algorithms and search algorithms; 3) Evaluating the performance of proposed approach using more complex case studies; and (4) Recommending configurations for the selected products based on the mined rules.

Acknowledgement

This work was supported by the Zen-Configurator project funded by the Research Council of Norway (grant no. 240024/F20) under the category of Young Research Talents of the FRIPO funding scheme. Tao Yue and Shaukat Ali are also supported by the Research Council of Norway funded MBT4CPS (grant no. 240013/O70) project.

References

1. Cyber-Physical Systems (CPSs). Available from: http://cyberphysicalsystems.org/.
2. Rawat, D.B., J.J. Rodrigues, and I. Stojmenovic, Cyber-Physical Systems: From Theory to Practice. 2015: CRC Press.
3. Nie, K., et al. Constraints: the core of supporting automated product configuration of cyber-physical systems. in Proceeding of International Conference on Model-Driven Engineering Languages and Systems (MODELS). 2013. Springer.
4. Yue, T., S. Ali, and B. Selic. Cyber-physical system product line engineering: comprehensive domain analysis and experience report. in Proceedings of the 19th International Conference on Software Product Line. 2015. ACM.
5. Kang, K., Cohen, Sholom., Hess, James., Novak, William., & Peterson, A., Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021), in Secondary Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021), Secondary Kang, K., Cohen, Sholom., Hess, James., Novak, William., & Peterson, A., Editor. 1990. RN. Available From:
6. Czarnecki, K., S. Helsen, and U. Eisenecker, Staged configuration using feature models, in Software Product Lines. 2004, Springer. p. 266-283.
7. Behjati, R., et al., SimPL: a product-line modeling methodology for families of integrated control systems. Information and Software Technology, 2013.
8. Haugen, O., Common Variability Language (CVL). OMG Revised Submission, 2012.
9. Berger, T., et al. A survey of variability modeling in industrial practice. in Proceedings of 7th International Workshop on Variability Modelling of Software intensive Systems. 2013. ACM.
10. Galster, M., et al., Variability in software systems-A systematic literature review. IEEE Transactions on Software Engineering, , 2014. **40**(3): p. 282-306.
11. Chen, L., M. Ali Babar, and N. Ali, Variability management in software product lines: A systematic review, in 13th International Software Product Line Conference. 2009. p. 81-90.

12. Arrieta, A., G. Sagardui, and L. Etxeberria, A comparative on variability modelling and management approach in simulink for embedded systems. V Jornadas de Computación Empotrada, ser. JCE, 2014.
13. Djebbi, O. and C. Salinesi. Criteria for comparing requirements variability modeling notations for product lines. in 4th International Workshop on Comparative Evaluation in Requirements Engineering. 2006. IEEE.
14. Eichelberger, H. and K. Schmid, A systematic analysis of textual variability modeling languages, in Software Product Line Conference. 2013, ACM. p. 12-21.
15. Sinnema, M. and S. Deelstra, Classifying variability modeling techniques. Information and Software Technology, 2007. **49**(7): p. 717-739.
16. Czarnecki, K., et al. Cool features and tough decisions: a comparison of variability modeling approaches. in 6th international workshop on variability modeling of software intensive systems. 2012. ACM.
17. Berger, T., et al., Variability modeling in the real: a perspective from the operating systems domain, in International conference on Automated software engineering. 2010, ACM. p. 73-82.
18. www.zen-tools.com/SAM2016.html. Available from: www.zen-tools.com/SAM2016.html.
19. http://www.pure-systems.com/. Available from: http://www.pure-systems.com.
20. http://modelbased.net/tools/ct-cvl/. Available from: http://modelbased.net/tools/ct-cvl/.
21. Safdar, S.A., M.Z. Iqbal, and M.U. Khan, Empirical Evaluation of UML Modeling Tools–A Controlled Experiment, in European Conference on Modeling Foundations and Applications. 2015, Springer: Italy. p. 33-44.
22. The UML MARTE profile, http://www.omgmarte.org/.
23. OMG, Systems Modeling Language (SysML) v1.4, http://sysml.org/. 2015.
24. Selic, B. and S. Gérard, Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. 2013: Elsevier.
25. Derler, P., E.A. Lee, and A.S. Vincentelli, Modeling Cyber–Physical Systems. Proceedings of the IEEE Special issue on CPS, 2012. **100**(1): p. 13-28.
26. Murguzur, A., et al. Context variability modeling for runtime configuration of service-based dynamic software product lines. in Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools. 2014. ACM.
27. Holl, G., P. Grünbacher, and R. Rabiser, A systematic review and an expert survey on capabilities supporting multi product lines. Information and Software Technology (IST), 2012. **54**(8): p. 828-852.
28. Rosenmüller, M. and N. Siegmund. Automating the Configuration of Multi Software Product Lines. in Proceeding of International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). 2010. Elsevier.
29. Video Conferencing Systems Available from: http://www.cisco.com/.
30. ULMA Handling Systems. Available from: http://www.ulmahandling.com.
31. Yue, T., S. Ali, and B. Selic. Cyber-Physical System Product Line Engineering: Comprehensive Domain Analysis and Experience Report. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2015. ACM.
32. Bécan, G., et al. Synthesis of attributed feature models from product descriptions. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2015. ACM.
33. Safdar, S.A., et al. Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering. in Proceeding of International Conference on System Analysis and Modeling (SAM). 2016. Springer.
34. Lu, H., et al., Model-based Incremental Conformance Checking to Enable Interactive Product Configuration. Information and Software Technology (IST), 2015. **72**: p. 68-89.
35. Lu, H., et al., Nonconformity Resolving Recommendations for Product Line Configuration, in International Conference on Software Testing. 2016, IEEE. p. 57-68.
36. Lu, H., et al., Zen-CC: An Automated and Incremental Conformance Checking Solution to Support Interactive Product Configuration, in 25th International Symposium on Software Reliability Engineering. 2014, IEEE. p. 13-22.
37. Temple, P., et al. Using Machine Learning to Infer Constraints for Product Lines. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2016. ACM.
38. Nadi, S., et al., Where do configuration constraints stem from? an extraction approach and an empirical study. IEEE Transactions on Software Engineering (TSE), 2015. **41**(8): p. 820-841.
39. Witten, I.H., E. Frank, and M.A. Hall, Data Mining: Practical machine learning tools and techniques. Third ed. 2011: Morgan Kaufmann.
40. Frank, E. and I.H. Witten. Generating accurate rule sets without global optimization. in Proceeding of International Conference on Machine Learning (ICML). 1998. University of Waikato, Department of Computer Science.

41. Satti, A., N. Cercone, and V. Keselj, Experiments in Web Page Classification for Semantic Web, in Workshop on Web-based Support Systems. 2004. p. 137-141.
42. McMinn, P., Search-based software test data generation: a survey. Software Testing Verification and Reliability (STVR), 2004. **14**(2): p. 105-156.
43. Ali, S., et al., Generating test data from OCL constraints with search techniques. IEEE Transactions on Software Engineering (TSE), 2013. **39**(10): p. 1376-1402.
44. Deb, K., et al., A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions on, 2002. **6**(2): p. 182-197.
45. Sarro, F., A. Petrozziello, and M. Harman. Multi-objective software effort estimation. in Proceeding of International Conference on Software Engineering (ICSE). 2016. ACM.
46. Pradhan, D., et al., Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study, in Genetic and Evolutionary Computation Conference. 2016, ACM. p. 1085-1092.
47. Pradhan, D., et al. STIPI: Using Search to Prioritize Test Cases Based on Multi-objectives Derived from Industrial Practice. in Proceeding of International Conference on Testing Software and Systems (ICTSS). 2016. Springer.
48. Wang, S., et al. Multi-objective test prioritization in software product line testing: an industrial case study. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2014. ACM.
49. Wang, S., et al., A Practical Guide to Select Quality Indicators for Assessing Pareto-based Search Algorithms in Search-Based Software Engineering, in International Conference on Software Engineering (ICSE). 2016.
50. Nebro, A.J., et al., AbYSS: Adapting scatter search to multiobjective optimization. Evolutionary Computation, IEEE Transactions on, 2008. **12**(4): p. 439-457.
51. Sokolova, M. and G. Lapalme, A systematic analysis of performance measures for classification tasks. Information Processing & Management (IPM), 2009. **45**(4): p. 427-437.
52. Han, J., J. Pei, and M. Kamber, Data mining: concepts and techniques. 2011: Elsevier.
53. Durillo, J.J. and A.J. Nebro, jMetal: A Java framework for multi-objective optimization. Advances in Engineering Software, 2011. **42**(10): p. 760-771.
54. Arcuri, A. and L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in 33rd International Conference on Software Engineering. 2011, IEEE. p. 1-10.
55. Ali, S. and K.A. Smith, On learning algorithm selection for classification. Applied Soft Computing, 2006. **6**(2): p. 119-138.
56. Mann, H.B. and D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other. The Annals of Mathematical Statistics, 1947. **18**(1): p. 50-60.
57. Vargha, A. and H.D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics (JEBS), 2000. **25**(2): p. 101-132.
58. Arcuri, A. and G. Fraser. On parameter tuning in search based software engineering. in Proceeding of International Symposium On Search Based Software Engineering (SSBSE). 2011. Springer.
59. Holmes, G., M. Hall, and E. Prank. Generating rule sets from model trees. in Proceeding of Australasian Joint Conference on Artificial Intelligence (AI). 1999. Springer.
60. Lopez-Herrejon, R.E., L. Linsbauer, and A. Egyed, A systematic mapping study of search-based software engineering for software product lines. Information and Software Technology (IST), 2015. **61**: p. 33-51.
61. Harman, M., et al. Search based software engineering for software product line engineering: a survey and directions for future work. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2014. ACM.
62. Wang, S., S. Ali, and A. Gotlieb, Cost-effective test suite minimization in product lines using search techniques. Journal of Systems and Software (JSS), 2014. **103**: p. 370-391.
63. Yi, L., et al. Mining binary constraints in the construction of feature models. in Proceeding of International Requirements Engineering Conference (RE). 2012. IEEE.
64. Zhang, B. and M. Becker. Mining complex feature correlations from software product line configurations. in Proceeding of International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). 2013. ACM.
65. Nadi, S., et al. Mining configuration constraints: Static analyses and empirical results. in Proceeding of International Conference on Software Engineering (ICSE). 2014. ACM.
66. Czarnecki, K., S. She, and A. Wasowski. Sample spaces and feature models: There and back again. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2008. IEEE.
67. Dumitru, H., et al. On-demand feature recommendations derived from mining public product descriptions. in Proceeding of International Conference on Software Engineering (ICSE). 2011. IEEE.
68. Softpedia. Available from: http://www.softpedia.com.

69. Hariri, N., et al., Supporting domain analysis through mining and recommending features from online product listings. IEEE Transactions on Software Engineering (TSE), 2013. **39**(12): p. 1736-1752.
70. Davril, J.-M., et al. Feature model extraction from large collections of informal product descriptions. in Proceeding of Joint Meeting on Foundations of Software Engineering (FSE). 2013. ACM.
71. Wang, S., et al., Automatic selection of test execution plans from a video conferencing system product line, in Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone. 2012, ACM: Innsbruck, Austria. p. 32-37.
72. Bagheri, E., et al. Configuring software product line feature models based on stakeholders' soft and hard requirements. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2010. Springer.
73. Mazo, R., et al., Recommendation heuristics for improving product line configuration processes, in Recommendation Systems in Software Engineering (RSSE). 2014, Springer. p. 511-537.
74. Witten, I.H., E. Frank, and M.A. Hall, Data Mining: Practical machine learning tools and techniques. 4th ed. 2016, Switzerland: Morgan Kaufmann. 734.
75. Safdar, S.A., et al. Mining Cross Product Line Rules with Multi-Objective Search and Machine Learning in Proceeding of The Genetic and Evolutionary Computation Conference (GECCO). 2017. Berlin, Germany: ACM.
76. Sayyad, A.S., et al. Scalable product line configuration: A straw to break the camel's back. in Proceeding of International Conference on Automated Software Engineering (ASE). 2013. IEEE.
77. Guo, J., et al., SMTIBEA: A hybrid multi-objective optimization algorithm for configuring large constrained software product lines. Software & Systems Modeling (SoSyM), 2017. **16**(4): p. 1-20.
78. Deb, K., et al., A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 2002. **6**(2): p. 182-197.
79. Konak, A., D.W. Coit, and A.E. Smith, Multi-objective optimization using genetic algorithms: A tutorial. Reliability Engineering & System Safety (RESS), 2006. **91**(9): p. 992-1007.
80. Deb, K. and H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. IEEE Trans. Evolutionary Computation, 2014. **18**(4): p. 577-601.
81. Jain, H. and K. Deb, An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach. IEEE Trans. Evolutionary Computation, 2014. **18**(4): p. 602-622.
82. Mkaouer, M.W., et al. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. in Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. 2014. ACM.
83. Han, J., M. Kamber, and J. Pei, Data mining: concepts and techniques. 3rd ed. 2012: Elsevier. 703.
84. Quinlan, J.R., C4.5: Programming for machine learning. 1st ed. 1993, London, UK: Morgan Kauffmann. 302.
85. Cohen, W.W. Fast effective rule induction. in Proceeding of International Conference on Machine Learning (ICML). 1995. Morgan Kaufmann.
86. Witten, I.H. and E. Frank, Data Mining: Practical machine learning tools and techniques. 2nd ed. 2005, San Francisco,USA: Diane Cerra. 525.
87. Lloyd, S., Least squares quantization in PCM. IEEE Transactions on Information Theory, 1982. **28**(2): p. 129-137.
88. Euclidean distance. 2002 2017; Available from: https://wikipedia.org/wiki/Euclidean_distance.
89. Guérin, J., et al., Clustering for different scales of measurement-the gap-ratio weighted k-means algorithm. arXiv preprint arXiv:1703.07625, 2017.
90. Henard, C., et al. Multi-objective test generation for software product lines. in Proceeding of International Systems and Software Product Line Conference (SPLC). 2013. ACM.
91. Marler, R.T. and J.S. Arora, Survey of multi-objective optimization methods for engineering. Structural and Multidisciplinary Optimization (SMO), 2004. **26**(6): p. 369-395.
92. Ali, S., et al. Empowering Testing Activities with Modeling-Achievements and Insights from Nine Years of Collaboration with Cisco. in Proceeding of International Conference on Model-Driven Engineering and Software Development (MODELSWARD). 2017. Springer.
93. Jitsi 2003; Available from: http://www.jitsi.org/.
94. Pradhan, D., et al. CBGA-ES: a cluster-based genetic algorithm with elitist selection for supporting multi-objective test optimization. in Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on. 2017. IEEE.
95. Henard, C., et al. Combining multi-objective search and constraint solving for configuring large software product lines. in Proceedings of the 37th International Conference on Software Engineering-Volume 1. 2015. IEEE Press.

96. Sayyad, A.S., T. Menzies, and H. Ammar. On the Value of User Preferences in Search-Based Software Engineering: A Case Study in Software Product Lines. in Proceeding of International Conference on Software Engineering (ICSE). 2013. IEEE.
97. Chhabra, J.K. An empirical study of the sensitivity of quality indicator for software module clustering. in Contemporary Computing (IC3), 2014 Seventh International Conference on. 2014. IEEE.
98. Mkaouer, M.W., et al. Preference-based multi-objective software modelling. in Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering. 2013. IEEE Press.
99. Ouni, A., et al. The use of development history in software refactoring using a multi-objective evolutionary algorithm. in Proceedings of the 15th annual conference on Genetic and evolutionary computation. 2013. ACM.
100. Arcuri, A. and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. in Proceeding of International Conference on Software Engineering (ICSE). 2011. IEEE.
101. Wu, J., et al., Assessing the quality of industrial avionics software: an extensive empirical evaluation. Empirical Software Engineering (EMSE), 2016. **22**(4): p. 1-50.
102. Sheskin, D.J., Handbook of Parametric and Nonparametric Statistical Procedures. 3rd ed. 2007, London,UK: Chapman and Hall, CRC Press. 1776.
103. Safdar, S.A., et al., Employing Multi-Objective Search and Machine Learning to Mine Cross Product Line Rules – A Technical Report, in Secondary Employing Multi-Objective Search and Machine Learning to Mine Cross Product Line Rules – A Technical Report, Secondary Safdar, S.A., et al., Editors. 2018, S.R. Laboratory: Oslo, Norway. p. 1-54. RN. 2018-05. Available From: https://www.simula.no/file/employingmulti-objectivesearchandmachinelearningtominecrossproductlinerulespdf/download
104. Kollat, J.B. and P.M. Reed, Comparing state-of-the-art evolutionary multi-objective algorithms for long-term groundwater monitoring design. Advances in Water Resources, 2006. **29**(6): p. 792-807.
105. Pradhan, D., et al., CBGA-ES+: A Cluster-Based Genetic Algorithm with Non-Dominated Elitist Selection for Supporting Multi-Objective Test Optimization. IEEE Transactions on Software Engineering, 2018.
106. Eiben, A.E. and S.K. Smit, Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation, 2011. **1**(1): p. 19-31.
107. Baars, A., et al. Symbolic search-based testing. in Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. 2011. IEEE Computer Society.
108. Alarcón-Jaén, J.N., Multi-objective approach for the minimization of test cases in Software Production Lines. 2018, University of Malaga, Spain. .
109. Runeson, P., et al., Case study research in software engineering: Guidelines and examples. 1st ed. 2012, New Jersey, USA: John Wiley & Sons. 237.
110. Wu, X., et al., Top 10 algorithms in data mining. Knowledge and Information Systems (KAIS), 2008. **14**(1): p. 1-37.
111. Pradhan, D., et al., Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study, in Proceedings of the Genetic and Evolutionary Computation Conference 2016. 2016, ACM: Denver, Colorado, USA. p. 1085-1092.
112. Wohlin, C., et al., Experimentation in software engineering: an introduction. 1st ed. 2000, Berlin, Germany: Kluwer Academic Publishers. 204.
113. Wang, S., et al. A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. in Proceeding of International Conference on Software Engineering (ICSE). 2016. ACM.
114. Zhao, L., M.J. Zaki, and N. Ramakrishnan. BLOSOM: A framework for mining arbitrary boolean expressions. in Proceeding of International Conference on Knowledge Discovery and Data Mining (KDD). 2006. ACM.
115. Yu, Y., et al. Mining and recommending software features across multiple web repositories. in Proceeding of Asia-Pacific Symposium on Internetware. 2013. ACM.
116. Bakar, N.H., Z.M. Kasirun, and N. Salleh, Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. Journal of Systems and Software (JSS), 2015. **106**: p. 132-149.

9 Appendix A: Examples of Generated Rules Using SBRM^{+}

Table D-13. Examples of CPL rules from Cisco and Jitsi case studies

Case study	Rule example
Rule format	Product.ConfigurableParameter = ConfigurableParameterValue **AND** … **AND** Product.ConfigurableParameter = ConfigurableParameterValue : SystemState (Support/Violation)
Cisco	VCS1.IP-Protocol = Sip **AND** VCS2.Listen-Port = Off **AND** VCS2.IP-Protocol = Sip **AND** Default-Transport = Tls : FailedFailed (34/5)
	VCS2.Max-Transmit-Callrate <= 5982 **AND** VCS1. IP-Protocol = Auto **AND** VCS1.Encryption = Off **AND** VCS2.Encryption = BestEffort **AND** VCS3.Encryption = BestEffort **AND** VCS3.Max-Transmit-Callrate > 135 : ConnectedConnected (103/1)
Jitsi	VCS1.IP-Pprotocol = AIM **AND** VCS3.Video-Codec = rtx **AND** VCS3.Audio-Codec = AMR-WB-16000 **AND** VCS2.Audio-Codec = SILK-12000 **AND** VCS1.Video-Codec = VP8 **AND** VCS1.Encryption = On **AND** VCS2.Encryption = BestEffort **AND** VCS1.Default-Callrate <= 5744 **AND** VCS2.Max-Receive-Callrate > 1680 **AND** VCS3.Max-Transmit-Callrate > 3005 : ConnectedFailed (30/1)
	VCS2.Video-Codec = VP8 **AND** VCS3.Video-Codec = h264 **AND** VCS2.MTU > 702 **AND** VCS1.MTU > 760 **AND** VCS1.Audio-Codec = SILK-16000 **AND** VCS1.SIP-Listen-Port = Off **AND** VCS1.Encryption = BestEffort **AND** VCS1.Video-Codec = VP8 **AND** VCS2.MTU > 806 : FailedConnected (32/9)