Evolving, Adapting and Optimizing Configurable Software Systems

Clément Quinton

Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

---

Habilitation à Diriger Des Recherches
Université de Lille

École Graduée MADIS-631
Spécialité : Informatique

Clément Quinton

Université de Lille & CRIStAL
Équipe-Projet Spirals

Evolving, Adapting and Optimizing Configurable Software Systems

Soutenue le 26 janvier 2024 devant le jury composé de

Rapporteurs:	Mathieu Acher	-	Professeur à l'Université de Rennes
	Marianne Huchard	-	Professeure à l'Université de Montpellier
	Olga Kouchnarenko	-	Professeure à l'Université de Franche-Comté
Examinateurs:	Jean-Christophe Routier	-	Professeur à l'Université de Lille
Garant:	Romain Rouvoy	-	Professeur à l'Université de Lille

Abstract

Software systems now hold a central role in our daily lives, and modern societies have undeniably become heavily dependent on them. Such software systems collaborate seamlessly with people, adapting to their various needs to support key societal activities. To accommodate the wide spectrum of user demands and adapt to diverse execution environments, modern software-intensive systems exhibit variability. The research activities presented in this manuscript address large, variable software systems. I give a partial view of my contributions in this domain, focusing on their evolution, adaptation, and optimization. In the first part, we deal with changes that these software systems undergo over time. In the second part, we investigate how to develop self-adaptation logic for such systems and in the third part, we propose methods to measure and reduce their energy consumption.

Résumé

Les systèmes logiciels occupent désormais une place centrale dans notre vie quotidienne, et les sociétés modernes en dépendent incontestablement. Ces systèmes logiciels collaborent de manière transparente avec les individus, s'adaptant à leurs besoins divers pour soutenir des activités sociétales essentielles. Afin de répondre au large éventail des demandes des utilisateurs et de s'adapter à divers environnements d'exécution, les systèmes logiciels intensifs sont devenus de plus en plus variables. Les activités de recherche présentées dans ce manuscrit traitent de grands systèmes logiciels variables. Je donne un aperçu partiel de mes contributions dans ce domaine, en mettant l'accent sur leur évolution, leur adaptation et leur optimisation. Dans la première partie, nous abordons les changements auxquels ces systèmes logiciels sont soumis au fil du temps. Dans la deuxième partie, nous examinons comment développer une logique d'auto-adaptation pour de tels systèmes, et dans la troisième partie, nous proposons des méthodes pour mesurer et réduire leur consommation d'énergie.

Contents

List of Figures

List of Tables

This document synthesizes the research work I have conducted since my PhD defense in October 2014. After a 3-year postdoc at Politecnico di Milano in Italy, I joined the Spirals research team as an associate professor in September 2017. Since then, I investigated various research directions in the area of configurable software systems. This journey, fruit of numerous encounters and exchanges, was exciting and thrilling but also challenging, having to comprehend the multiple sides of this great new job while dealing with two house moves and a pandemic – not to mention the birth of two children.

Chapter 1

Introduction

My research activities cover a variety of topics related to software engineering ranging from modeling to optimization, yet with software configuration and variability management as the common, central concerns. More specifically, this manuscript presents the results of the research work that, together with national and international colleagues and students I have had the chance to collaborate with, I carried out in the area of configurable software regarding their evolution, adaptation and optimization. The decision to focus on these topics is a practical consideration; it fits together, follows on the research work conducted during my PhD and builds the foundations of my research activity for the near future. This choice is, of course, not intended to reduce the importance nor the quality of our results in other areas, such as recent work on software maintenance or knowledge compilation [Abou Khalil 2019, Bourhis 2022, Bourhis 2023].

Configurable Software. Twelve years ago, Marc Andreessen claimed that "*software* (was) *eating the world*[1]" and indeed, it is now undeniable that software systems play a crucial role in our lives. Not a day goes by that we do not interact with a software whether watching TV, driving a car, ordering food, checking our heartbeat on a smartwatch or being woken up by the alarm on our smartphone. This diversified yet simple and reduced series of usages show how modern societies now heavily rely on software systems that collaborate together with people and adapt to their various needs to support key societal activities. To meet various user's requirements and adapt to different execution environments, modern software-intensive systems exhibit *variability*. Software variability is the ability to create software variants for different market segments or contexts of use [Czarnecki 2013]. In other words, modern software-intensive systems are highly configurable [Jin 2014], as there is no unique way to use them. Software engineers thus have to develop, test and maintain a significant number of options, or *features*, that are then combined together to produce a specific software configuration fitting a specific user need. Software configuration is a hot topic in research and industry [Siegmund 2020]. The research community in this field now hosts two dedicated international conferences and several top-tiers software engineering conferences also include software configuration related topics in their call for papers. Variability management has become such a crucial concern for developers (*e.g.*, with the advent of microservices, software is now divided into multiple configurable and inter-dependent microservices, packaged up into various containers, and hosted in a distributed cloud environment) that addressing configuration failures is considered as one of the most important research directions [Sayagh 2020]. The last decade has also seen the growth of large, dynamic self-adaptive systems such as IoT systems, cloud/edge-based systems or cyber-physical ones. These systems provide runtime adaptation and reconfiguration capabilities to react to changes in their environment, usually run continuously, and cannot be shut down for reconfiguration or maintenance tasks. For instance, cyber-physical systems must frequently reconfigure their software components at runtime to take into consideration the addition, removal or update of physical devices.

Software Product Line (SPL) engineering is a commonly adopted approach to deal with software variability. SPL engineering promotes the production of similar software *variants* by composing reusable and inter-dependent features, thus enabling cost and time-to-market reduction. Features and their dependencies are usually described in a variability model, which defines the possible variants of a system together with domain-specific constraints and dependencies. The

[1] PDF version accessible at https://osr.cs.fau.de/wp-content/uploads/2016/08/marc.pdf. Last accessed on September 2023.

variability model encodes the system's problem space, *i.e.*, stakeholder needs and desired features, and is associated with its solution space, *i.e.*, the components realizing the solution architecture. Mappings between these two spaces then allows to assemble and configure a software variant based on customers' requirements [Berg 2005, Seidl 2012]. *Dynamic Software Product Lines* (DSPL) provide the conceptual framework for managing the variability in software systems at runtime [Hallsteinsen 2008, Hinchey 2012]. A DSPL borrows the means to define and manage variability from conventional software product lines, but additionally supports system reconfiguration at runtime by enabling activation or deactivation of certain features according to the changing context. Precisely, unlike traditional SPL engineering where features are bound together at design time and where multiple variants can coexist, the DSPL engineering process extends the SPL process by adding post-deployment and reconfiguration activities to manage the reconfiguration of a single software variant [Capilla 2014].

Challenges. Dealing with software variability is complex. This complexity is due to the potentially large number of features, which can lead to a combinatorial explosion of possible product configurations, but also to the variability management itself such as its discovery, modeling, maintenance, testing or validation. In this manuscript, we will focus on three particular concerns regarding variability management.

- ◇ *Evolution.* DSPL, like any software system, undergoes evolution [Ghezzi 2017]. For example, new features may need to be added or existing features may need to be removed. Managing evolution is particularly difficult in a DSPL context, as changes are made at runtime, which can easily lead to inconsistencies among running components. Specifically, it is challenging to maintain the consistency between the problem and the solution spaces, the variability model and the running system, as well as the runtime adaptation mechanisms. Many approaches have been proposed for managing the evolution of software product lines [Marques 2019], ranging from verification techniques to ensure consistent evolution, to model-based frameworks dedicated to the evolution of feature-based variability models [Pleuss 2012]. For example, an interesting research thread proposes evolution templates for co-evolving a variability model and related software artifacts [Seidl 2012, Passos 2013, Neves 2015]. Model-checking approaches are used to guarantee the consistency of a variability model after evolution [Guo 2012, Quinton 2014]. Furthermore, approaches for comparing the set of possible products before and after the evolution of a product line have been proposed [Thüm 2009, Neves 2011]. These approaches, however, are limited regarding support for DSPL evolution, as they focus on guaranteeing the consistency of the evolved DSPL variability model but fail to ensure that the evolution is consistent with the DSPL actual implementation and adaptation mechanisms.

- ◇ *Adaptation.* To build a self-adaptive system, software engineers have to develop *self-adaptation logic* that encodes when and how the system should adapt itself. However, in doing so, software engineers face the challenge of *design time uncertainty* [Weyns 2013, Calinescu 2020]. Among other concerns, developing the adaptation logic requires anticipating the potential environment states the system may encounter at runtime to define *when* the system should adapt itself. Yet, anticipating all potential environment states is in most cases infeasible due to incomplete information at design time. As an example, consider a microservice-oriented system which dynamically binds microservices at runtime. What concrete microservices will be bound at runtime and thus their quality of service are typically not known at design time. As a further concern, the precise effect of an adaptation action may not be known and thus accurately determining *how* the system should adapt itself is difficult. For instance, while software engineers may know in principle that activating more features will have a negative impact on performance, exactly determining the performance impact is more challenging [Siegmund 2012].

- ◇ *Optimization.* Dealing with large configuration spaces is challenging, especially when configurations must comply with both functional constraints and non-functional performance goals. In particular, the large number of configurations makes picking the *best* configuration on the first try almost impossible, unless having the proper background knowledge of the configuration space. Developers usually do not have this background knowledge and only consider less than 20% of the available configurations [Xu 2015]. Another reason is the use of the default configuration or a legacy one, *e.g.*, to make sure functional requirements are met. Running such a configuration does not guarantee running the optimal one; On the contrary, it may result in running worst or incorrect configurations [Nair 2020, Pereira 2021]. By measuring the performance of a SPL, one can then reconfigure the system to switch to a better configuration. But such a measurement is a challenging task due to the large number of features and products that must be considered. Yet, measuring performance is crucial, especially regarding energy consumption, as several studies showed that software has a significant impact on the energy consumed and consider green software design as a key development concern to improve the energy efficiency of software systems at large [Islam 2016, Jagroep 2016, Pereira 2020].

Contributions overview. In the first chapter of this manuscript, we show through concrete examples how evolution can affect the consistency of a DSPL. We then propose a flexible approach, based on a reference architecture, to implement evolution support for a DSPL, together with two implementations of the reference architecture for two different DSPLs in different domains – one cyber-physical system and one runtime monitoring system, both using different means for variability management. We perform an evaluation of the feasibility and performance of our approach by simulating common evolution scenarios for both DSPLs to demonstrate that both implementations are capable of detecting inconsistencies introduced in a DSPL at runtime, and we demonstrate the industrial applicability of our approach by applying it to a real-world automation software system DSPL from the injection molding domain.

In the second chapter, we investigate variability-driven reinforcement learning approaches to realize self-adaptation in the presence of design time uncertainty. We focus on two problems related to how adaptation actions are explored: *(i)* existing solutions randomly explore adaptation actions and thus may exhibit slow learning if there are many possible adaptation actions to choose from; *(ii)* existing solutions are unaware of evolution, and thus may explore new adaptation actions introduced during such evolution rather late. We propose novel exploration strategies that leverage the variability model to guide exploration in the presence of many adaptation actions and in the presence of service evolution. Our evaluation indicates an average speed-up of the learning process in the presence of many adaptation actions, and in the presence of evolution.

In the third chapter of this manuscript, we study performance variations in configurable systems. In particular, we investigate the impact of feature interactions on the system performance, with a specific focus on energy consumption. We propose a method to measure and reduce the energy consumption of multiple variants at once by sampling and analyzing a minimal set of variants. This method provides means to estimate the energy consumption of individual features, to highlight how feature interactions impact the energy consumed by variants and to propose variants with lower energy consumption. Relying on these findings, we then propose an approach that, regarding multiple performance objectives, guides the optimization of a software performances by suggesting a better performing configuration. This optimization is performed by altering as little as possible the initial configuration, thus preventing from moving away from initial functional requirements.

Supervision

Over the last five years, I have been the co-advisor of one successful doctorate, Zeinab Abou Khalil (defended February 2021) and, after obtaining a dispensation, I also have been the sole advisor of another successful doctorate, Edouard Guégain (defended September 2023). I currently co-advise three second-year doctoral students since October 2022: Alexandre Bonvoisin and Tristan Coignion (co-supervised with Romain Rouvoy) and Maxime Huyghe (co-supervised with Walter Rudametkin).

Maxime HUYGHE, 2022-2025 (50%, co-advised with Walter Rudametkin)
Automated software testing to improve the privacy of browsers.
Doctoral contract, University of Lille and Région Hauts-de-France.

Tristan COIGNION, 2022-2025 (50%, co-advised with Romain Rouvoy)
New development models for cloud-native deployed microservices.
Financed through the ANR DISTILLER project (Leader: R. Rouvoy).

Alexandre BONVOISIN, 2022-2025 (50%, co-advised with Romain Rouvoy)
Frugal software architectures for deploying cloud-native microservices.
Financed through the ANR DISTILLER project (Leader: R. Rouvoy).

Jérémy DUSART (post-doc), 2021-2022 (50%, co-advised with Pierre Bourhis)
Reasoning on Large-Scale Configuration Spaces.
Financed through the Région Hauts-de-France CPER DATA COMMODE project.

Edouard GUEGAIN, 2020-2023 (100%, sole advisor)
Taming the Complexity of Fog Environments.
Financed through the ANR JCJC KOALA project.

Zeinab ABOU-KHALIL, 2017-2020 (33%, co-advised with Laurence Duchien & Tom Mens, UMons)
Studying the evolution of the bug handling process in large open source ecosystems.
Defended February 2021. Currently data engineer at Sanofi.

Fundings & research grants

The research work I conducted over the last years has mainly be supported by the following fundings and research grants :

2022 **University of Lille** doctoral grant to work on *Automated software testing to improve the privacy of browsers*. This grant finances Maxime Huyghe's doctoral contract, started in October 2022.

2022 **ANR PRCE** funding. *DISTILLER : recommenDer servIce for SusTaInabLe cLoud nativE softwaRe)* - PI : Romain Rouvoy. This grant finances Alexandre Bonvoisin and Tristan Coignion doctoral contracts, started in October 2022.

2019 **I-Site ULNE** joint doctorate grant - Leader. *Dependable adaptive software systems for the digital world*, joint supervision with KU Leuven (Belgium). This grant financed Omid Gheibi's doctoral contract, started in October 2020.

2019 **ANR JCJC** funding. *KOALA : Knowledge-based fog-scale configurations* — Leader. The project's main objective is to deliver a series of innovative tools, methods and software to deal with the complexity of software-intensive systems' configurations and adaptations. The project started in August 2020 with the recruitment of Edouard Guégain and ends in December 2023. Total funding: 183k€.

2019 **CPER DATA** funding. *COMMODE : Knowledge compilation for feature models* — Leader. Research and collaboration contract with the CRIL lab at Lens on the topic of Reasoning on Large-Scale Configuration Spaces. Jérémy Dusart's postdoc was funded under this contract. Total funding: 62k€.

Publications

The following is my list of publications, since my postdoctoral stay, presented in reverse chronological order.

Journal publications

2022 **Realizing Self-Adaptive Systems via Online Reinforcement Learning and Feature-Model-guided Exploration**. Andreas Metzger, Clément Quinton, Zoltán Mann, Luciano Baresi, Klaus Pohl. Computing, 2022. https://hal.science/hal-03595102v1

2020 **Evolution in Dynamic Software Product Lines**. Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, Christian Schuhmayer. In Journal of Software Evolution & Process, vol. 33, issue 2, 2020. https://hal.science/hal-02952741v2

2016 **SALOON, a Platform for Selecting and Configuring Cloud Environments**. Clément Quinton, Daniel Romero, Laurence Duchien. In Software: Practice & Experience, vol. 46, issue 1, 2016. https://hal.science/hal-01103560v1

Peer-reviewed conference publications

2023 **Configuration Optimization with Limited Functional Impact**. Édouard Guégain, Amir Taherkordi, Clément Quinton. International Conference on Advanced Information Systems Engineering, CAiSE 2023. *Core rank: A*. https://hal.science/hal-04034888v1

2021 **On Reducing the Energy Consumption of Software Product Lines**. Édouard Guégain, Clément Quinton, Romain Rouvoy. International Conference on Systems and Software Product Lines, SPLC 2021. https://hal.science/hal-03269168v1

2020 **Feature Model-Guided Online Reinforcement Learning for Self-Adaptive Services**. Andreas Metzger, Clément Quinton, Zoltan Adam-Mann, Luciano Baresi, Klaus Pohl. International Conference on Service Oriented Computing, ICSOC 2020. *Core rank: A*. https://hal.science/hal-02982029v2
🏆 **Best Paper Award**.

2019 **A Longitudinal Analysis of Bug Handling Across Eclipse Releases**. Zeinab Abou Khalil, Eleni Constantinou, Tom Mens, Laurence Duchien and Clément Quinton. IEEE International Conference on Software Maintenance and Evolution, ICSME 2019. *Core rank: A*. https://hal.science/hal-02179172v1

2016 **Learning and Evolution in Dynamic Software Product Lines**. Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, Klaus Pohl. In 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2016. *Core rank: A*. https://hal.science/hal-01280837v1

2015 **SmartyCo: Managing Cyber-Physical Systems for Smart Environments**. Daniel Romero, Clément Quinton, Laurence Duchien, Lionel Seinturier, and Carolina Valdez. In 9th European Conference on Software Architecture, ECSA 2015. *Core rank: A*. https://hal.science/hal-01172057v1

2015 **Evolution in Dynamic Software Product Lines: Challenges and Perspectives**. Clément Quinton, Rick Rabiser, Michael Vierhauser, Paul Grünbacher, Luciano Baresi. In 19th International Software Product Line Conference, SPLC 2015. https://hal.science/hal-01180935v1

2015 **Dynamically Evolving the Structural Variability of Dynamic Software Product Lines**. Luciano Baresi, Clément Quinton. In 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015. *Core rank: A*. https://hal.science/hal-01120248v1

Peer-reviewed workshop publications

2023 **ICO : A Platform for Optimizing Highly Configurable Systems**. Edouard Guégain, Amir Taherkordi, Clément Quinton. International Workshop on Automated and verifiable Software sYstem Development, ASYDE 2023. https://hal.science/hal-04213194

2018 **Giving Students a Glimpse of the SPL Lifecycle in Six Hours: Challenge Accepted!** Clément Quinton. In 22th International Software Product Line Conference, vol 2. SPLC 2018.

Non peer-reviewed articles

2023 **Reasoning on Feature Models: Compilation-Based vs. Direct Approaches**. Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, Clément Quinton. https://arxiv.org/abs/2302.06867

2022 **Pseudo Polynomial-Time Top-k Algorithms for d-DNNF Circuits**. Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, Clément Quinton. https://arxiv.org/abs/2202.05938

Organization of the manuscript

Chapter 2 describes the challenges of evolving DSPL and presents a reference architecture to support DSPL evolution.

Chapter 3 proposes novel exploration strategies that use feature models to guide exploration in the presence of many adaptation actions and in the presence of evolution.

Chapter 4 shows the impact a configuration can have on system performance and highlights the need for better understanding configuration choices to optimize performances.

Finally, **Chapter 5** concludes the manuscript while providing research perspectives.

Each chapter contextualizes and aggregates material published in different venues. This material is explicitly listed at the end of each chapter, either as a publication (marked with the icon), a software ( ) or a replication package ( ).

Chapter 2

Evolution

Contents

Over the last year of my PhD, I studied evolution and consistency checking for software product lines, especially regarding the configuration and deployment of cloud applications. It was then natural to delve into the future of these applications, investigating how to manage them once up and running. Relying on dynamic software product line engineering was the logical choice and, in the rapidly changing landscape of cloud computing, issues related to the evolution of such DSPLs arose. This chapter elaborates on the research that we conducted to address these issues, together with colleagues from the Johannes Kepler University of Linz, Austria.

2.1 Background and Motivation

Feature Models and Modeling Spaces. In the remainder of this manuscript, we will mainly rely on a specific type of variability model to encode variability, namely *feature models*. A feature model is a tree of features organized hierarchically that describes the possible and allowed feature combinations [Metzger 2014]. A feature f can be decomposed into mandatory, optional or alternative sub-features. If feature f is activated, its mandatory sub-features have to be activated, its optional sub-feature may or may not be activated, and at least one of its alternative sub-features has to be activated. Additional cross-tree constraints express inter-feature dependencies. As depicted by Fig. 2.1, modeling product line variability concerns the problem space (*i.e.*, features and capabilities), the solution space (*i.e.*, components of the solution architecture), and the mapping space (*i.e.*, links between problem and solution space elements) [Apel 2009]. The problem space includes domain-specific abstractions describing the requirements on a software system. The solution space refers to the concrete artifacts of the product line. There is thus a mapping between both spaces, describing which artifact belongs to which requirement or abstraction, *i.e.*, feature.

Open Issues. A DSPL, like any SPL, is a long-term investment that is in use for many years and needs to be continuously evolved *e.g.*, to meet new requirements or to adopt new technologies [Botterweck 2014]. There are three evolution scenarios in a DSPL context. First, the change and the related adaptation are triggered manually, *e.g.*, when updating the code base. Second, the change can be performed manually, while the adaptation is automated *e.g.*, in a build automation or DevOps process. Finally, both the change and the adaptation can be triggered automatically, which is the case for self-adaptive systems where evolving environment implies a

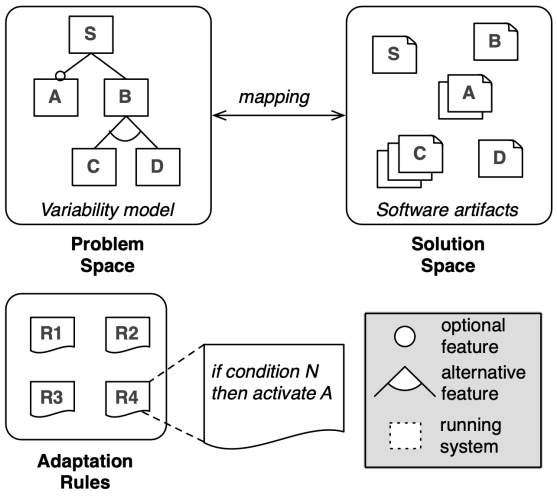


Figure 2.1: Problem, solution and mapping spaces involved in a DSPL. Adaptation rules drive the reconfiguration, which activates software artifacts depending on the selected features.

reconfiguration of the system. Evolving a DSPL poses significant challenges as both problem and solution space must co-evolve with the system they describe to avoid inconsistencies during runtime adaptation. Most existing work on SPL evolution had focused on the evolution of the problem space only [Botterweck 2014]. However, evolving a variability model may also affect the related artifacts (*i.e.*, the solution space) and vice versa. Yet, limited work had been conducted to support such co-evolution. Furthermore, existing approaches typically only supported one particular variability management approach and were not flexible enough to allow their use in different domains and for different types of systems, using diverse implementation techniques. A variability model-agnostic approach was thus still missing that facilitated the evolution of problem and solution spaces, together with the runtime adaptation mechanisms, and also checked the consistency of the resulting products. This chapter presents the framework we elaborated to address the aforementioned problems.

To illustrate the challenges and issues faced when evolving a DSPL, we introduce a cyber-physical system (CPS) for home automation as an example of an adaptive system combining both hardware devices and software systems. The CPS consists of smart devices equipped with sensors and actuators interconnected through a software system. Sensors are used to retrieve information from the environment; reconfiguration plans are then carried out through the actuators. Figure 2.2 depicts an excerpt of the variability model of our CPS, along with related adaptation rules.

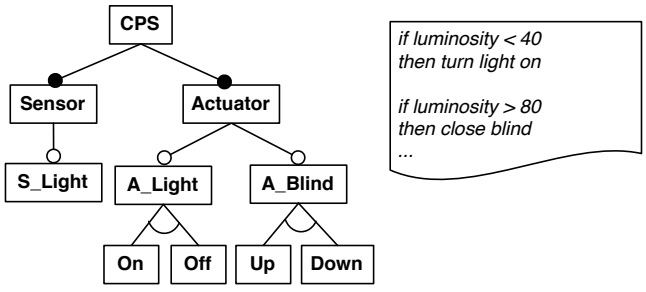
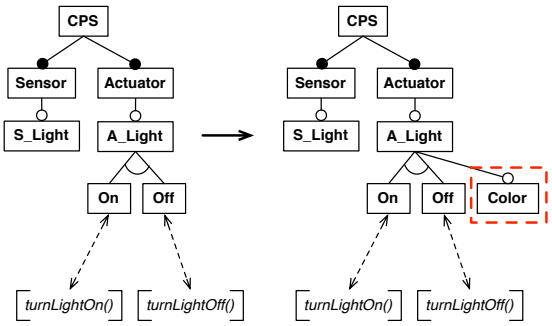


Figure 2.2: Excerpt of the variability model of a cyber-physical system and its related adaptation rules.

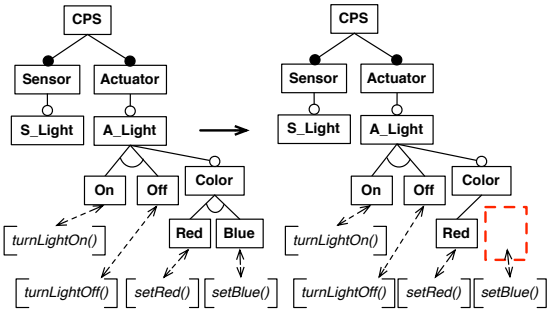
In this case, a possible reconfiguration action is to switch on light through the actuator A_Light whenever the luminosity in a given room, captured by sensor S_Light, is lower than 40 lumens, or to close the roller blinds of the window, whenever it is higher than 80 lumens. Obviously, a CPS automatically managing the luminosity, temperature, humidity, and energy consumption of a home relies on many more adaptation rules for different rooms and devices, and each adaptation rule is thus related to the variability model describing possible reconfigurations of the CPS.

2.2 Impact of Evolution on a DSPL

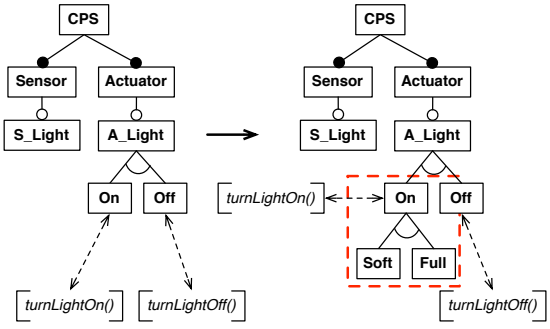
This section discusses all the possible changes that can occur in the three modeling spaces of the DSPL during evolution and their potential effect on the running system. We present examples of these effects on the runtime adaptations of the CPS DSPL, and then discuss the related challenges. Although we describe only one change per space in our examples, the changes can affect the consistency of the overall DSPL. The left-hand side of each sub-figure shows the initial DSPL, and the right-hand side the one after evolution. Elements depicted with square brackets are software artifacts from the solution space, *e.g.*, components or services. Throughout our examples, we assume that an operation, whose signature is given between the square brackets, is implemented by these software artifacts. Dashed lines with arrows on both sides represent the mapping between problem and solution spaces. Rectangles in dashed lines highlight the resulting inconsistencies.



(a) Adding an element to the problem space.



(b) Removing an element from the problem space.



(c) Updating an element of the problem space.

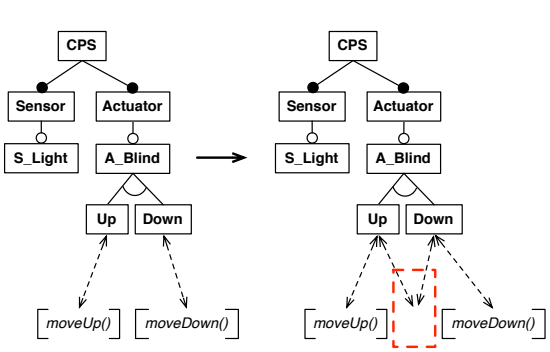
Figure 2.3: Problem space evolutions.

Problem Space. For instance, in Figure 2.3a the optional feature Color is added to the system to make the light actuator now configurable by allowing the selection of a light color. This feature may require specific colored light bulbs. The problem in this case is caused by the missing software artifact that needs to be mapped to the new feature. In our example, Color must be defined and included in the adaptation rule, since otherwise the Color feature is empty (has no associated

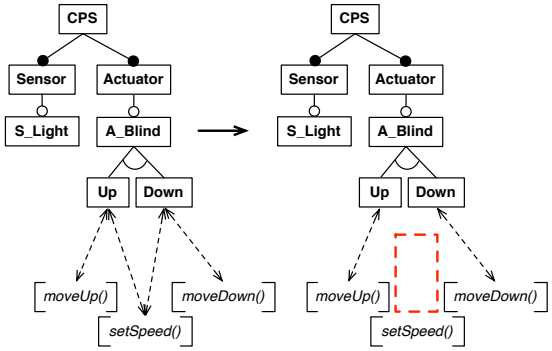
operation), *i.e.*, the reconfiguration will have no effect[1]. The problem occurs immediately if the feature is mandatory, or when activating an optional feature.

Removing a feature can easily impact the consistency of the DSPL. Let us consider a case where the color of the light could have been red or blue, but the latter cannot be set anymore after evolution, *e.g.*, because of a hardware problem (Figure 2.3b). If feature Color is involved in an adaptation rule, *e.g.*, turn on the blue light when it is 07:00 in the morning, then the reconfiguration would fail whenever this condition is met, even though the related solution space element is available.

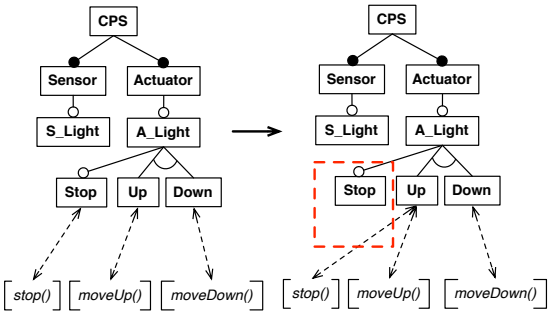
A feature, or any other element of the solution space, *e.g.*, a constraint, can also just be updated[2]. For example, one can think of a different way of switching lights on and off. The variability model is updated when feature On is upgraded, as a dimmer now allows the light to be turned on completely or partially (Full and Soft modes in Figure 2.3c). Adaptation rules that involve feature On (Figure 2.2) need to be updated to reflect this change and to avoid an inconsistency. Updating the rules is done by setting the correct action to perform, *e.g.*, when the luminosity becomes lower than 40 lumen, turn on the light in soft mode.



(a) Adding an element to the mapping space.



(b) Removing an element from the mapping space.



(c) Updating an element of the mapping space.

Figure 2.4: Mapping space evolutions.

Mapping Space. A new mapping can be added when a feature requires an extension and is then realized by composing two different software artifacts. For example, Figure 2.4a considers the case in which the implementation of features Up and Down for blinds requires two software artifacts after evolution: one for controlling the speed and another one for the direction (we assume both were handled by the same artifact before). If the mapping is not correctly defined and does not point to any artifact, the adaptation is partially inconsistent, *e.g.*, one can still

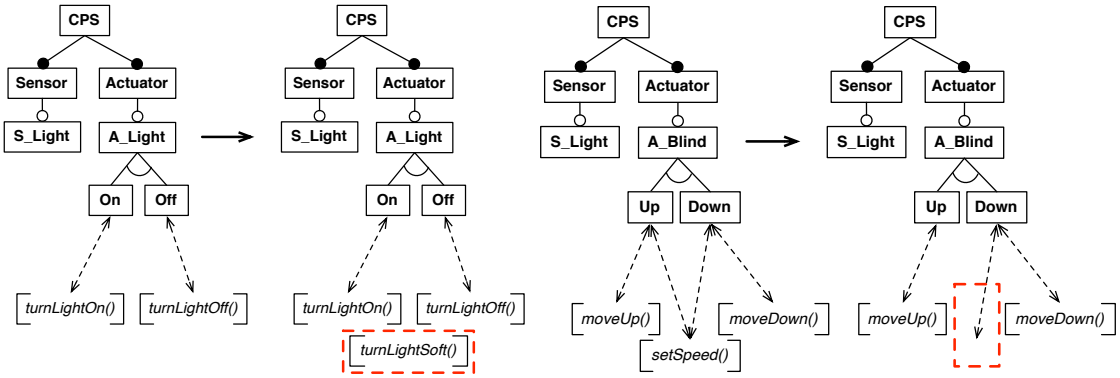
[1] We consider that a feature is empty when its selection has no effect on the expected adaptation.

[2] An update is considered as a transaction, that is, a sequence of atomic changes treated as one change (*e.g.*, Add + Rem). The DSPL is thus checked only once after the update instead of both after the Add and after the Rem.

control the direction (up/down) of the blinds but not their speed since the related artifact is not mapped correctly.

Removing a mapping element can prevent the proper reconfiguration of the running system. In Figure 2.4b, conversely to the previous scenario, artifacts *moveUp()* and *moveDown()* also control the speed after evolution, so the mapping related to speed is removed. In that case, if the software artifacts are not properly updated, the element of the solution space that controls the speed (*setSpeed()*) becomes a dead asset.

When updating a mapping, the reference to the element needs to be changed either in the problem space or in the solution space. For example, Figure 2.4c illustrates the case where artifact *Stop()*, which was initially bound to feature Stop, is now bound to feature Up: if one pushes button *Up* when the blind moves down, it stops, instead of having a dedicated button *Stop*. Adaptations related to feature Stop would now fail, as software artifacts are no longer bound to this feature. For instance, such an adaptation could require that the blind stops when there is too much wind or rain.



(a) Adding an element to the solution space.

(b) Removing an element from the solution space.

(c) Updating an element of the solution space.

Figure 2.5: Solution space evolutions.

Solution Space. Figure 2.5a considers a new artifact that implements a soft new way of turning on the light. If this new software artifact should be taken into consideration for the reconfiguration at runtime, related elements in the mapping as well as in the problem space must also be added.

Removing a software artifact when evolving the solution space can lead to problems when different features share the same artifact. Figure 2.5b illustrates this situation as artifact *setSpeed()* is removed. In this scenario, feature Up no longer needs such an artifact (we suppose this is now implemented by artifacts *moveUp()* and *moveDown()*). This can result in a partially inconsistent adaptation, *i.e.*, the blind can still move down but the speed cannot be controlled, which can be an issue.

Finally, Figure 2.5c illustrates an evolution in the solution space, where the implementation of feature On is changed. After the change, activating this feature turns on the light in soft mode. Although the resulting reconfiguration turns the light on (in soft mode, but on anyway), this may not be the expected behavior with respect to the semantics of feature On. The evolved software artifact could also be an implementation that is completely unrelated to the light system and turns on the TV, thus leading to another inconsistent adaptation.

Table 2.1 summarizes the different evolution scenarios in the problem and solution spaces and the mapping between the two, with respect to atomic tasks/changes as discussed above. It further summarizes the possible effects of each change on the consistency of the CPS DSPL and the impact on the running configuration.

	Change	Potential impact on the CPS DSPL
Problem Space	Add	**feature selection will have no effect** : the added Color feature (cf. Fig 2.3a) is empty[1], *i.e.*, not related to any solution space element, and selecting it will thus have no effect.
	Rem	**artifact cannot be activated anymore** : setBlue() (cf. Fig 2.3b) cannot be activated as it is no longer related to any problem space element.
	Upd	**feature selection will have no effect** : Soft and Full (cf. Fig 2.3c) are empty features, *i.e.*, not related to any solution space element, and selecting them will thus have no effect.
Mapping Space	Add	**artifact cannot be activated anymore** : the Blind speed cannot be controlled (cf. Fig 2.4a) as the new mapping does not point to a solution space element.
	Rem	**artifact cannot be activated anymore** : setSpeed() (cf. Fig 2.4b) cannot be activated as after removing the mapping, no feature is related with setSpeed() anymore.
	Upd	**feature selection will have no effect** : feature Stop (cf. Fig 2.4c) is empty, *i.e.*, not related with any solution space element, and selecting this feature now will not have any effect anymore.
Solution Space	Add	**artifact cannot be activated anymore** : turnLightSoft() (cf. Fig 2.5a) cannot be activated, as no feature is related to the new solution space element.
	Rem	**runtime adaptation will partly fail** : feature Down (cf. Fig 2.5b) is not fully implemented, *i.e.*, the speed can no longer be controlled as this solution space element has been removed.
	Upd	**unexpected runtime behavior** : wrong implementation (asset) for feature On (cf. Fig 2.5c) leads to unexpected behavior.

Table 2.1: Changes in different modeling spaces and examples of their potential effect on the CPS DSPL. Add stands for addiction, Rem for removal and Upd for update.

Challenges

The discussed cases show that evolution in DSPLs requires the consideration of multiple aspects [Quinton 2015]. We see three main challenges:

C_1: Supporting evolution independently of the domain, implementation technique, or modeling approach. Existing DSPL evolution approaches focus on a particular variability modeling approach, *e.g.*, feature models [Capilla 2014]. In practice, however, different approaches are used [Czarnecki 2012]. Also, different ways of modeling adaptation rules can be utilized in relation

to variability models. Furthermore, systems are often implemented using various technologies. The success of an approach for the evolution of DSPLs in practice thus depends on its flexibility to support different modeling and implementation techniques. This suggests a generic architecture that guides the implementation of concrete solutions, which use specific variability modeling approaches and adaptation mechanisms.

C_2: Ensuring consistency of models and the running system. The consistency of the DSPL must be checked whenever at least one of the spaces evolves. Consistency must thus be checked for each space (intra-space consistency) and across spaces (inter-space consistency). For instance, related elements from different spaces can become inconsistent with the actual software or hardware and prevent the derivation of products, despite the validity of these products in the variability model. Although patterns for keeping co-evolving both modeling spaces consistent have been studied [Tartler 2011, Passos 2015], most approaches so far have focused on the consistency of either the problem space [Thüm 2009, Benavides 2010, Botterweck 2014] or the solution space [McGregor 2003]. Also, existing work focuses on checking the consistency (of models) in software product lines without considering dynamic software product lines: while model consistency is similar in a SPL and a DSPL, consistency checking in a DSPL also needs to take into account the *running* system. Thus, it must be ensured that its adaptation rules are still consistent with the variability model and do not violate any possible reconfiguration. Detecting such inconsistencies is not straightforward as adaptation rules are typically defined independently of the variability model using diverse *ad-hoc* approaches such as event-condition-action rules [Bencomo 2010, De Lemos 2013, Capilla 2014]. When the model evolves, the rules should be updated automatically. Also, when evolving rules, the model must potentially also be updated.

C_3: Supporting evolution triggered by the running system or the model. The evolution of a DSPL can be driven from two different perspectives. First, the different spaces can evolve, *e.g.*, a feature might be added to the problem space or a new component might be developed in the solution space to address new requirements. Whenever one space evolves, the other space must evolve accordingly. Only then a configuration defined in the problem space can be materialized by composing elements from the solution space (*e.g.*, Fig. 2.3a–2.3c), and vice versa (*e.g.*, Fig. 2.5a–2.5c). Updating a running system – to reflect changes made to the modeling spaces also in the running system – can be challenging depending on the technologies used. Second, the evolution[3] of a DSPL can also be driven by the running system: if the system changes both modeling spaces may need to evolve to reflect these changes. Approaches have been proposed for dealing with the co-evolution of different modeling spaces [Seidl 2012, Borba 2012], for reflecting system evolution in the models [Acher 2011, She 2011], or vice versa [Font 2015]. However, they are typically only capable of dealing with a particular set of changes and cannot handle both evolution between modeling spaces and evolution driven by changes in the system.

2.3 A Reference Architecture to Support Evolution

To address the aforementioned challenges, we present a reference architecture [Bass 1998] that supports DSPL evolution.

Reference Architectures A number of approaches exist to inform the development of reference architectures [Nakagawa 2012, Galster 2011]. Following the types of reference architecture proposed by Angelov et al. [Angelov 2009] we regard our reference architecture as a Type 5. Such architectures are designed to facilitate the design of systems that will become needed in the future. Our reference architecture defines the key components required in a system implementing it, discusses algorithms supporting the operation of the components, and presents protocols demonstrating the interactions among the components.

[3]Please note that in our approach, an evolution results in a new version of the DSPL. The previous version of the variability model is replaced with the evolved one and earlier configurations may no longer be derivable.

A reference architecture aims at *providing guidance (i.e., instructions) on how to actually design a system* [Weinreich 2014]. Our reference architecture *(i)* is independent of specific domains, implementation techniques, or modeling approaches for DSPLs (cf. challenge C_1); *(ii)* it focuses on ensuring the consistency of the running system and of the models representing the DSPL (cf. challenge C_2); and *(iii)* it supports evolution triggered from different perspectives (cf. challenge C_3).

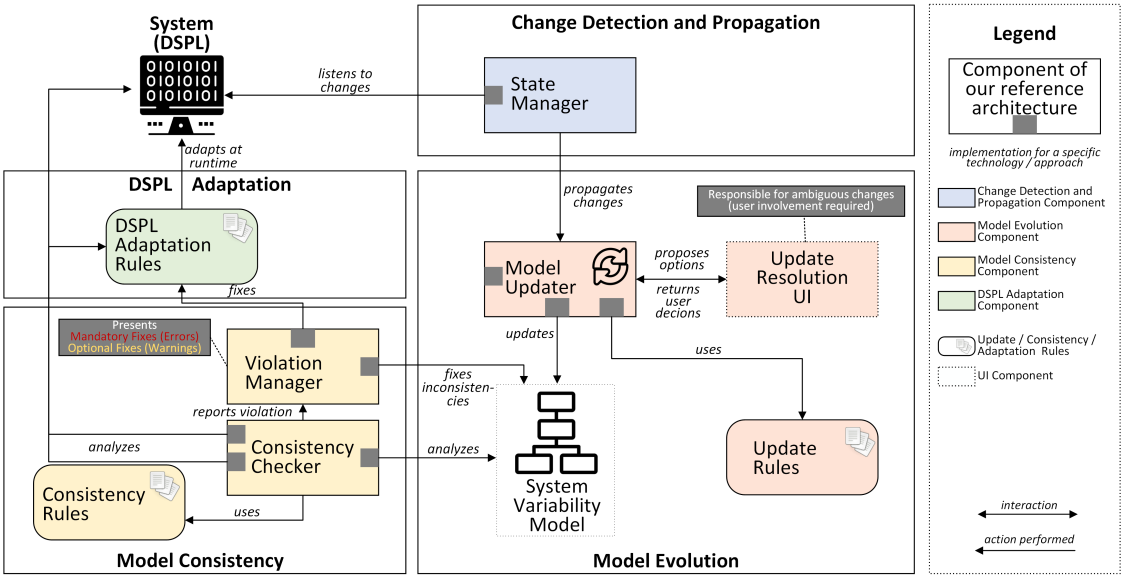


Figure 2.6: Reference architecture for DSPL evolution.

The reference architecture is divided into four parts, as depicted in Figure 2.6. *DSPL Adaptation* comprises the adaptation rules defined to automatically adapt the system at runtime. *Change Detection and Propagation* comprises a component that listens to the running system to detect any (relevant) change made to the system and subsequently propagates this change to the *Model Evolution* component. *Model Evolution* comprises components to evolve the variability model and/or the adaptation rules based on the changes received from Change Detection and Propagation, given the specific scenario (see Section 2.2). *Model Consistency* uses a consistency checker to ensure that performed evolution operations, especially if based on human decisions, do not introduce inconsistencies in the DSPL. The parts, each responsible for a given concern in the DSPL evolution process, and the components they comprise, are described independently of any concrete (modeling) approach and can be implemented for any systems. Small squares in the components indicate where such specific implementations are needed, *e.g.*, to let the Model Updater update the Variability Model for a concrete variability modeling approach. In Table 2.2, we list generic operations for each component, *e.g.*, to listen to changes made to the running system and to update the variability models accordingly. These operations can be implemented when creating a concrete solution for a DSPL. To this end, we describe two different implementations in Section 2.4.2 . Below, we describe each part of the reference architecture and the components it comprises, again referring to the CPS example.

DSPL Adaptation is responsible for managing all adaptation rules related to the DSPL. The specific way adaptation rules are described depends on the domain and the implementation of the DSPL. For instance, the rules may be encoded using a domain-specific language that describes event-condition-action rules [Bencomo 2010, De Lemos 2013, Capilla 2014]. When a change occurs in the running system, the adaptation rules are likely to evolve together with the rest of the DSPL, *e.g.*, as an adaptation rule may involve a feature that is no longer present in the

variability model after the change. For example, an adaptation rule defining the color of the light to be turned on under certain conditions will no longer be valid after the DSPL evolution scenario described in Figure 2.5b, and thus the rule must be adapted or removed from this component to properly evolve the DSPL (cf. challenges C_1 and C_2).

Change Detection and Propagation is responsible for detecting and propagating changes in the running system. A State Manager monitors the running system and receives notifications whenever changes occur. This could be achieved either by actively listening to system changes, or by periodically querying the system and calculating changes with respect to a previously observed state. When a change is detected by the State Manager, it is passed to the Model Updater, which determines relevant update actions for the variability model (cf. challenge C_3).

In **Model Evolution**, the variability model defines all possible configurations for the software managed as DSPL, and thus guides all adaptations, that is, it switches from one configuration to another given a particular adaptation rule. The variability model can be defined using any available variability modeling approach [Czarnecki 2012]. Our architecture distinguishes between two types of changes on the variability model: in case of *unambiguous* changes – *e.g.*, the CPS system gets a new feature to turn on a green light – the variability model can be updated immediately and automatically: a new feature Green, can be added as a child of feature Color (cf. Table 2.1: PS_Add). In practice, however, multiple stakeholders may maintain a DSPL. This can lead to ambiguous changes, which cannot be resolved automatically. The DSPL maintainer needs to be prompted through an interface – the Update Resolution UI – to decide on how to react and how to update the variability model by selecting among suggested possible actions or performing a custom one. For instance, the scenario described in Figure 2.3a is not trivial, as feature Color can be added as a child of feature A_Light or of feature On. An ambiguous change can also occur when a new component with a different name substitutes a running component, leading to semantic issues that could only be solved using ontologies to map components together, *e.g.*, as in [Quinton 2013]. Update Rules help automate model updates (cf. challenge C_3). For instance, an update rule may specify that an optional feature must be added to the variability model as a child of the root feature if a new optional functionality is available in the system. Whenever a change is ambiguous, the DSPL maintainer performs a manual evolution using the Update Resolution UI and can then define a new rule based on the change she has just made. Once stored with the other rules, the rule can be used to automate future evolution scenarios to avoid possible ambiguity. Update Rules and Model Updater depend on the used approach for modeling variability. Our architecture provides generic operations (cf. Table 2.2) that can be implemented for a specific approach, to specify update operations and update rules, as described in Section 2.4.2. The list of possible changes in different modeling spaces (cf. Table 2.1) is a useful basis to develop update rules, *i.e.*, changes in problem, mapping, and solution spaces can be automated in reaction to the evolution of the DSPL.

Model Consistency is essential when dealing with evolution in a DSPL, especially when different people are responsible for different parts of the system and or variability models. In our approach, different possible inconsistencies are defined as Consistency Rules (or constraints) that can be fed to a dedicated Consistency Checker (cf. challenges C_2 and C_3). The Consistency Checker analyzes the running system together with the variability model and the adaptation rules to detect any inconsistencies. The Consistency Checker thus needs to interface with the variability model but also requires status updates of the running system. A consistency rule could, for instance, specify that for each problem space feature there must be at least one solution space asset realizing that feature, *i.e.*, *for each feature at least one component must exist in the system that realizes that feature.* Another rule could define that *for each feature selected for the DSPL (*i.e.*, 'activated'), the respective component realizing the feature must be currently running ('active') in the system.* The concrete set of consistency rules will depend on the system and on the modeling approach used, as we will show in our evaluation.

In case of any detected inconsistencies, a Violation Manager prompts the user through a dedicated interface. This could involve critical errors that must be fixed (*e.g.*, if the removal of an element from the variability model breaks an existing adaptation rule), but also minor incon-

Component/Operation	Description
State Manager	
`detectChange(changeEvent)`	Monitors the running system and listens to any `changeEvent`, *e.g.*, new component deployed to the running system.
`notifyChanges(change[])`	Periodically sends the list of `change[]` to the Model Updater.
Model Updater	
`analyzeChanges()`	Reacts to `notifyChanges(change[])` and analyzes the required changes. Calls update methods for variability model and/or DSPL adaptation rules accordingly.
`updateVariabilityModel(modelChange)`	Searches for an update rule matching the required `modelChange`, *e.g.*, add feature. If found, it parses the rule to update the model automatically, *e.g.*, by adding a feature. If an automatic update is not possible, *e.g.*, in case of ambiguous changes, the operation prompts the user.
`readUpdateRule(rule)`	Parses the related update `rule` to infer the proper automated model evolution to be applied.
`updateAdaptationRule(rule, ruleChange)`	Searches for the DSPL adaptation `rule` to be evolved and performs the required `ruleChange`. *E.g.*, when a feature has been renamed in the model, the referring adaptation rule needs to be modified too.
Consistency Checker	
`parse(model)`	Loads and reads the variability `model`.
`parse(adaptationRules)`	Loads and reads the DSPL `adaptationRules`.
`checkConsistency(model)`	Checks the consistency of the variability `model`, *i.e.*, checks for issues in different modeling spaces and also compares variability model and running system with each other according to the defined consistency rules.
`checkConsistency(adaptationRules)`	Checks the consistency of the DSPL `adaptationRules`.
`checkConsistency(model, adaptationRules)`	Checks if the variability `model` is consistent with the DSPL `adaptationRules`
Violation Manager	
`fixInconsistencyInModel(modelChange)`	Updates the model to fix an inconsistency. Prompts the user if required.
`fixInconsistencyInAdaptationRule(ruleChange)`	Updates the adaptation rule to fix an inconsistency. Prompts the user if required.

Table 2.2: Generic operations of our DSPL evolution reference architecture.

sistencies needing attention (*e.g.*, if the addition of an asset duplicates an existing one). The Consistency Checker is thus in charge of detecting structural inconsistencies after they occurred, while semantic ones (*e.g.*, mapping to the wrong elements) are left to the user. Again, however,

ontologies could be applied to also cover at least basic semantic inconsistencies. The different parts of our reference architecture are independent of each other, leaving the concrete implementation and application scenario to decide the technique or modeling approach to be used.

2.4 Empirical Evaluation

We investigate three research questions regarding the feasibility and applicability of our approach. Specifically, we assess the general feasibility of our approach by implementing it for two different DSPLs, and determine for the two concrete implementations of the reference architecture if inconsistencies arising from different evolution scenarios are correctly managed. We further show how our reference architecture can be applied to a real-world DSPL in the domain of automation software for injection molding machines.

The first DSPL is a cyber-physical system providing capabilities for controlling and managing home automation devices such as sensors and actuators under certain conditions, *e.g.*, turning off the TV in case no one is watching it [Romero 2015]. To deal with the context-awareness of the devices in use, *e.g.*, a Belkin WeMo thermostat, adaptation rules are defined and managed by controllers deployed on set-top boxes. For instance, an adaptation rule for this device is *if someone turns on the thermostat, send an SMS to registered user #1*. Such systems are highly configurable and likely to evolve, as running devices may face failures, or new devices may be added to an existing system to provide new functionality. More details about the different spaces of the CPS DSPL are presented in [Romero 2015].

The second DSPL is an event-based runtime monitoring infrastructure ReMinds [Vierhauser 2016]. ReMinds has a client-server architecture: systems are instrumented using probes that send events and data from the systems to the ReMinds server, which aggregates and distributes these events and data to registered clients. Clients are, for instance, tools for checking constraints [Vierhauser 2015] on the expected behavior based on the monitored events or visualization components explaining constraint violations to facilitate diagnosis. In previous work, we emphasized the need for sophisticated runtime variability management mechanisms [Rabiser 2015] and support for automated evolution in ReMinds [Quinton 2015], since a monitoring infrastructure must co-evolve with the underlying system it monitors. In the context of this chapter, however, we focus on the evolution of the components of ReMinds itself: the probes, monitored events and data, and constraints being added, modified, or removed at runtime in the ReMinds monitoring infrastructure.

2.4.1 Research Questions

RQ 1: *Is the reference architecture flexible enough to support different DSPL implementations?*

To assess the feasibility of the proposed architecture, we implemented it for the two different DSPLs described above. For both DSPLs, we implemented the components and operations of our reference architecture and created variability models, partly based on existing ones [Rabiser 2015]. The two implementations use different technologies and different kinds of variability models, yet they both comply with the reference architecture. For example, the CPS DSPL uses Eclipse EMF [Steinberg 2009], Java, and extended feature models [Benavides 2010, Bąk 2011], while ReMinds uses Eclipse, Java, and DOPLER [Dhungana 2011] decision models.

RQ 2: *How well do the two reference architecture implementations perform?*

To assess how well the two implementations manage inconsistencies to support DSPL evolution, we simulated scenarios that are likely to happen in the two applications and evaluated their impact on the respective DSPL, to find out whether each implemented approach is able to detect inconsistencies and to react appropriately/fast enough. Table 2.3 provides an overview of our evaluation setup. We used problem space, mapping space, and solution space operations to cover a fair range of different possible inconsistencies and picked the following representative scenarios:

1. Scenario 1 (SC1) – A problem space element is removed to constrain possible reconfigurations of the DSPL. This can result in dead elements (assets) in the solution space that were related to the removed element of the problem space.
2. Scenario 2 (SC2) – The mapping space is updated after merging two existing assets into a single one. This can result in a problem space element with no effect (no relation to any solution space element).
3. Scenario 3 (SC3) – A solution space element is added to evolve the system, *i.e.*, a new component can now be configured. If not mapped to an existing or new problem space element, the new solution space element is dead and cannot be activated at runtime.

Table 2.3: Overview of our evaluation setup.

CPS DSPL [Romero 2015]	**ReMinds DSPL [Vierhauser 2016]**
Feature Model	Decision Model
2000 features, 200 adaptation rules, 3000 assets	400 decisions, 1000 assets
SC1: A feature is removed from the feature model	SC1: A decision is removed from the model
SC2: An existing mapping (feature to asset) is updated	SC2: An existing mapping (asset to decision) is updated
SC3: A new asset is added	SC3: A new asset is added to the model
Runs per scenario: 10	
Total number of changes: 1000	
Seeded inconsistencies: 1 random defect introduced in every 100 changes	

For each of these three scenarios we perform 100 changes that lead to an automated update of the respective variability model of the CPS and ReMinds DSPLs. 99 out of 100 changes lead to correct updates while 1 change (randomly, *e.g.*, the 67th change) leads to an incorrect update resulting in an inconsistency. For example, an incorrect update operation for scenario SC3 would add the solution space element without also relating it to any problem space element or other element. This simulates an error a user might actually make during DSPL evolution. Indeed, while adding a solution space element can be automated, the mapping of the new solution space element to an existing or a new problem space element requires user involvement or at least an update rule as described earlier, and errors can thus easily be introduced.

To evaluate the performance, we measured for the two DSPL implementations the time required to check the consistency after we seeded one inconsistency. We based our evaluation on significant variability models, *e.g.*, feature models with 2000 features, considered by Berger *et al.* [Berger 2013] as large feature models. Concrete performance measurements, which are specific for each implementation and were performed on different machines, are discussed in Section 2.4.2.

RQ 3: *Industrial applicability: can the reference architecture be used to support a real-world DSPL implementation?*

To assess the applicability of the proposed architecture, we implemented it for a real-world DSPL, *i.e.*, an automation software system for injection molding machines by an Upper Austrian company[4]. More specifically, the MoldingCompany was extending the architecture of its automation software to allow plugging external devices into machines at runtime. The DSPL approach

[4]Due to non-disclosure agreements we refer to this company as "MoldingCompany"

discussed in this chapter was regarded as promising to deal with the identified adaptation and reconfiguration scenarios and to better support new devices and vendors in the future. The goal was to shift the process of connecting new unknown peripheral devices to machines from design time (pre-deployment) to runtime (post-deployment). Depending on the capabilities of the connected devices, features are enabled or disabled. Furthermore, this approach allows to activate or deactivate features based on results from monitoring an injection molding machine. Specifically, an engineer from MoldingCompany developed capabilities for runtime adaptation of the automation software including adaptation rules guided by the reference architecture presented in this chapter. To support the reconfiguration of the system at runtime the engineer developed a feature model and a component model to represent the system components that can be (de-)activated or reconfigured at runtime. In Section 2.4.2.3, we present the created architecture instance, the models, and the DSPL adaptation rules. We further describe how our architecture supports future evolution scenarios (*e.g.*, adding new devices) in this domain and report initial feedback from MoldingCompany engineers.

2.4.2 Results

We implemented and customized our reference architecture for the CPS and ReMinds DSPL to demonstrate that it is sufficiently flexible to support different implementations of the various components. Both implementations use different technologies and different kinds of variability models but comply with the architecture. To find out whether our approach is capable of properly detecting inconsistencies, we performed experiments that reflect the three previously described evolution scenarios and measured the time[5] required to check the consistency of each DSPL for which we implemented our reference architecture.

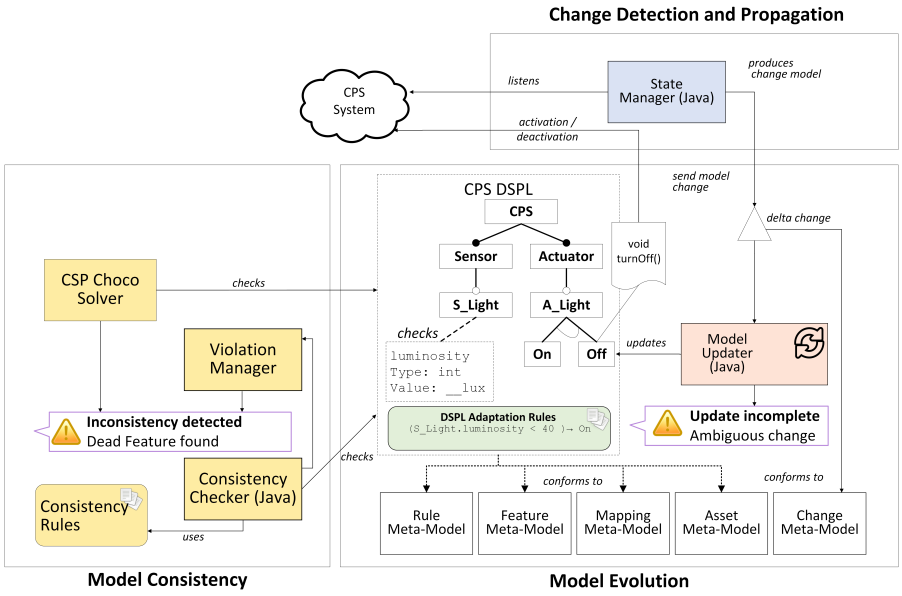
2.4.2.1 Reference Architecture Implementation for the CPS DSPL

Implementation (RQ 1) We first implemented the reference architecture for the CPS DSPL, which relies on feature models to manage runtime variability and adaptation rules as described below. Implementing and customizing the reference architecture for the CPS DSPL took about two person-weeks and was done by one developer. He could reuse initial variability models of this DSPL created in earlier work [Romero 2015] and a consistency checker developed for a different project [Quinton 2016]. As depicted in Figure 2.7a , each component of the CPS DSPL conforms to its respective meta-model. It thus provides a flexible means to define DSPLs for different domains or evolving a given DSPL, by switching model components whenever needed. The CPS DSPL is thus an instance of the DSPL meta-models, and is managed through Eclipse and EMF.

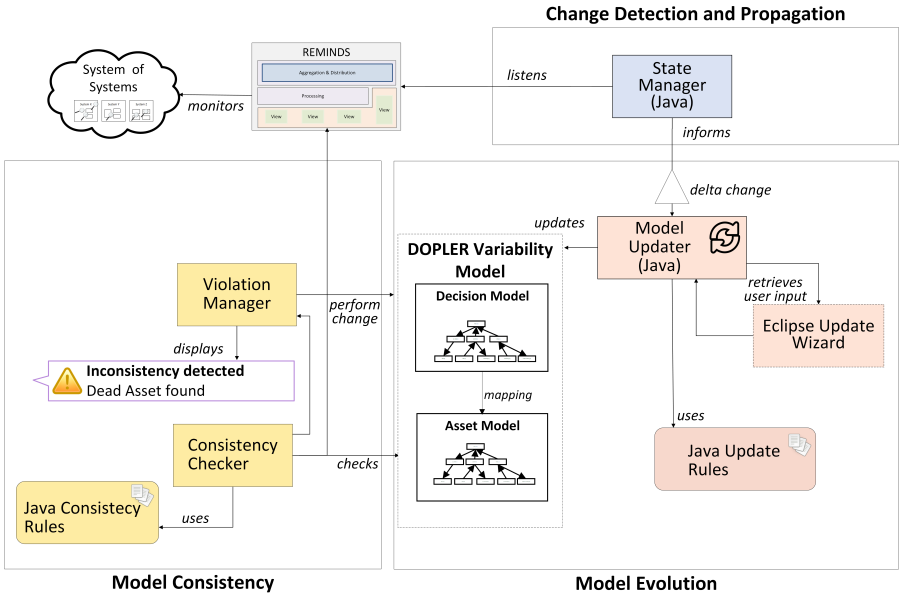
Adaptation. Adaptation rules are described in the feature model, such as the one shown in Figure 2.7a. In particular, we rely on *extended* feature models, *i.e.*, feature models whose additional information is defined in terms of feature *attributes* [Benavides 2010, Bąk 2011]. Specifically, adaptation rules are defined as attribute-based constraints. To distinguish between design-time model constraints and runtime adaptation rules, we rely on a slight extension of the feature meta-model proposed in [Quinton 2016]. This extension is twofold and consists of a Boolean attribute, runtime, added to the meta-classes Constraint and Attribute. The former enables the definition of runtime constraints, *i.e.*, adaptation rules, while the latter is used to define runtime attributes, *i.e.*, their value will only be taken into consideration at runtime. In practice, attributes and constraints flagged with runtime are not used during the initial configuration. They become active only once the system is up and running. Runtime attribute values remain unset at design time, and are then updated at runtime relying on an event listener that listens to changes that occur in the environment of the system.

Change Detection and Propagation. We developed an interface that handles change detection through a monitoring system implementing the operations proposed in Table 2.2. Once

[5]Please note that the numbers regarding the average evaluation times for the two approaches cannot be compared with each other since the approaches are implemented using different technologies for constraint solving and error reporting. However, the numbers provide a hint on the general performance of both approaches.



(a) Implementation for CPS



(b) Implementation for ReMinds

Figure 2.7: Implementations of our reference architecture for the CPS DSPL and the ReMinds DSPL.

a change is detected, it is propagated to State Manager that translates this change into two models, *(i)* a model fragment that comprises the model element(s) to be updated and *(ii)* a change model, which is an instance of the change meta-model. The change meta-model enables the definition of all types of changes, *i.e.*, the addition, update or removal of elements in any space. Once translated, the two models are used by Model Updater to evolve the related models accordingly.

Model Evolution. The Model Updater retrieves the models related to the change, and evolves the main models by relying on update rules. It parses the change model together with the adaptation rules, the mapping space model or the feature model depending on the scenario, and

performs the change. For instance, the change model can describe the removal of an element in the solution space (cf. Table 2.1: SS_Rem). Whenever the Model Updater retrieves such a model, it looks for an update rule named *removeSSElement* in the rules repository. This rule describes the guidelines of a proper removal, *i.e.*, it removes the solution space element itself and the related mapping space elements.

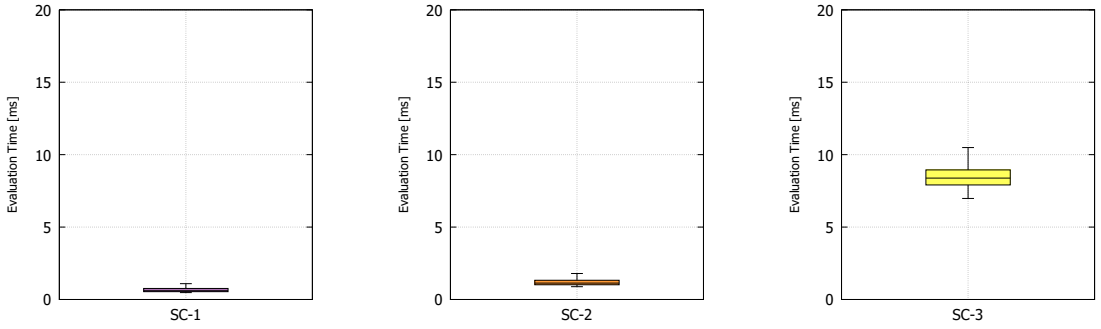
When the change model and the update rule make it feasible, the change is performed automatically, otherwise an output message is displayed within the Eclipse console to involve the developer. For instance, adding a new option (cf. Table 2.1: PS_Add) for an existing functionality can be handled automatically (*i.e.*, an update rule defines that a new variant feature $f2$ is added as a direct child feature of the existing one $f1$: Add $f2$ childOf $f1$), while moving features may not be straightforward (*e.g.*, the user has to define where to insert the moved feature and move sub-features accordingly). Once the evolution is performed (either manually by the user – *e.g.*, moving features; or automatically by the framework based on update rules – *e.g.*, new option), the consistency of the DSPL is checked.

Model Consistency. Once evolved, the feature model is translated into a *Constraint Satisfaction Problem* (CSP). We then use the Choco CSP solver [Prud'homme 2014] (but any Java-based solver would also integrate smoothly, *e.g.*, SAT4J) to check the consistency of the feature model. The proposed model-based approach helps ensuring the consistency of the overall DSPL and acts as Violation Manager. First, changes are translated into model fragments that conform to the related meta-model and are thus consistent with the rest of the model. Second, models directly refer to other models/model fragments, ensuring all models are consistent with each other. Indeed, the main issue with most existing feature-based DSPL approaches is that adaptation rules are defined independently of the feature model. The evolution of a DSPL is thus often error-prone as for such approaches there is no way to check whether adaptation rules and the feature model are consistent after evolution. Instead, we define adaptation rules as models and make them refer to features in the feature model, similar, *e.g.*, to Gamez and Fuentes [Gamez 2011] who define adaptation rules in a reconfiguration plan and relate it to a feature model.

Performance Evaluation (RQ 2) To assess the performance of our approach we programmatically generated models larger than the models manually created for the CPS. The generation process produces *(i)* random features (2000 features) and adaptation rules (200), *(ii)* 1-2 assets per feature (*i.e.*, a total of 3000 assets), and thus *(iii)* one or two mappings per feature. While these models are randomly generated, their size and structure can be compared with real feature models, *e.g.*, from the operating system domain as reported by Berger *et al.* [Berger 2013]. For each evolution scenario SC1-SC3, we then performed 99 change operations leading to valid model updates, and 1 change operation leading to an inconsistency. In addition to checking whether all inconsistencies were indeed detected we measured the average evaluation time for evaluating consistency after each change. More precisely, we measured the time required to retrieve the content of all involved EMF models, and to check whether all references of a given model are present in other models or not. All experiments were performed on a MacBook Pro with a 2,6 GHz Intel® Core™ i5 processor and 8 GB of DDR3 RAM.

By relying on the model-based approach used in the CPS DSPL, all generated inconsistencies were detected and their cause was explained. Depending on the scenario, the time required to check the consistency of the CPS DSPL varies from 0.75ms up to 8.5ms on average (0.75ms for SC1, 1.3ms for SC2, and 8.5ms for SC3). Details on the evaluation times for the three scenarios are presented in Figure 2.8 (a–c). We observed that the time required to check the consistency is negligible and is not a threat to the scalability of the model-based support in the CPS implementation of our reference architecture. The main advantage of our implementation is that the Consistency Checker can rely on methods from the EMF API to load and parse the models. While for SC1 and SC2 only 2 elements of the DSPL are involved (feature model and adaptation rules for SC1, feature model and mappings for SC2), SC3 requires the feature model, the mappings, and the assets to be loaded, parsed and checked, which explains the increased time for checking the consistency for SC3 (Figure 2.8 (c)). Overall, our empirical evaluation indicates

that our approach is well-suited for dealing with DSPLs with a substantial number of features, adaptation rules, and assets.



(a) CPS - SC1 (b) CPS - SC2 (c) CPS - SC3

Figure 2.8: Average consistency constraint evaluation times when detecting seeded inconsistencies (1 out of 100 change operations) for 1000 performed changes for 3 scenarios for the CPS implementation of our reference architecture.

2.4.2.2 Reference Architecture Implementation for the ReMinds DSPL

Implementation (RQ 1) To support DSPL evolution for ReMinds, which uses decision models to manage variability, we have implemented our reference architecture as described below (cf. Figure 2.7b). Implementing and customizing the reference architecture for the ReMinds DSPL also took about two person-weeks and was done by one of the main developers of ReMinds. He could reuse initial variability models of ReMinds created in earlier work [Rabiser 2015] and a consistency checker developed for a different project [Vierhauser 2012].

Adaptation. In our earlier work, we developed an approach for variability management that provided the starting point for our implementation of the reference architecture to support the reconfiguration of ReMinds at runtime [Rabiser 2015]. Specifically, we describe the variability of the key components of ReMinds (probes, event types, constraints) using decision-oriented DOPLER variability models [Dhungana 2011]. In DOPLER, decisions define configuration options (to be set by end users or programmatically via the DOPLER API). The decision type defines what values can be set on decisions (Boolean, string, number, or enumeration). A decision can depend on other decisions hierarchically, if it needs to be made before other decisions, or logically, if the answer affects other decisions. Decisions are related to assets in DOPLER models (this is the problem to solution space mapping): answering decision questions allows selecting and (re-)configuring components. Assets in DOPLER models represent the core product line artifacts (*e.g.*, software components) in the solution space. Assets can depend on each other functionally (*e.g.*, one asset requires another asset) or structurally (*e.g.*, if an asset is part of a another asset), *i.e.*, assets can have solution space dependencies. Using DOPLER, users can create domain-specific meta-models to define the asset types, attributes, and dependencies for their domain or system. In the ReMinds case study, for instance, the asset types are probe, event type, and constraint. They have different attributes and are related to each other, specifically, probes provide events of different types and constraints check events.

Users or programs can set decision values for decisions defined in the DOPLER variability model, thereby (de-)activating probes, constraints, and related event types at runtime through an interface that connects DOPLER and ReMinds. Decisions thus represent the possible adaptations that can be made at runtime, *i.e.*, the adaptation rules. If, for instance, a ReMinds probe is used to instrument an archiving component (persistence management) of a system, the decision could be called *monitoring_archiving*, with the question 'Do you want to monitor the archiving process?', and related to the archiving probe represented by an asset in the DOPLER model with name *archiving_probe*. More examples can be found in [Rabiser 2015].

Change Detection and Propagation. The ReMinds framework provides interfaces for retrieving state information of the elements of the running monitoring infrastructure (*i.e.*, probes, constraints, event types) to determine whether the elements are active or inactive, and also for retrieving information on elements being added, removed or modified. We developed the component State Manager of our reference architecture to implement these interfaces. It keeps track of the current state of the elements in the running ReMinds infrastructure, periodically checks which elements have been added, removed or modified, and forwards this information to component Model Updater.

Model Evolution. We implemented the Model Updater as an Eclipse Plug-in in the DOPLER variability modeling IDE [Dhungana 2011]. It is triggered by State Manager whenever a change in ReMinds occurs. Based on Update Rules, Model Updater checks whether update actions can be performed automatically or user input is required. More specifically, our implementation of the Model Updater distinguishes unambiguous and ambiguous changes. In case of unambiguous changes, an update is triggered directly by the Model Updater to automate the changes to the variability model. For ambiguous changes, user feedback is required. In such cases, the Update UI (a simple Eclipse Wizard) is triggered, allowing the user to select a particular resolution strategy. For example, when a probe is removed from ReMinds, an update rule could specify to simply remove the respective asset from the variability model. However, as the asset might be mapped to one or more other model elements, *e.g.*, decisions, it will typically make sense to involve the user via the Update UI and ask her to decide whether to really remove the asset together with all mappings to it.

Model Consistency. To check model consistency, we rely on an existing Consistency Checker [Vierhauser 2012] that allows to check the consistency of a decision model and arbitrary artifacts based on consistency rules (not to be confused with the constraints used by ReMinds to check system behavior at runtime). For the purpose of the ReMinds case study we extended this existing Consistency Checker to check the conformance between the DOPLER variability model and ReMinds. We implemented the consistency rules for ReMinds as an extension of the existing Consistency Checker in Java. Facades provide access to the DOPLER variability model on the one hand – to retrieve model elements such as assets and decisions – and to ReMinds on the other hand - to retrieve registered and running probes and constraints. Internally, the Consistency Checker employs an incremental approach [Egyed 2006] thus reducing the overhead and providing instant feedback to users on emerging violations.

The Violation Manager is integrated within the DOPLER IDE and retrieves information on occurring inconsistencies from the Consistency Checker. It reports details to the user on the violation of each consistency rule, *e.g.*, the origin within the model and the cause of the violation. In case of a possible resolution a (semi-)automated fix can be applied to the model. In the mentioned example of adding a new asset to the variability model due to a new probe added to ReMinds, the Consistency Checker *(i)* immediately detects this asset as dead if it has not also been mapped to a decision as described above, and *(ii)* informs the user about this inconsistency via the Violation Manager. Figure 2.9 depicts a screenshot of the DOPLER modeling IDE showing constraint violations (upper part) and currently active consistency rule (lower part). Different types of violation are highlighted differently depending on their severity (*e.g.*, Errors vs. Warnings). The engineer can review each violation and navigate to its origin for an in-depth inspection, *e.g.*, of the model element not defined properly. Up to this point we have not implemented support for automatically fixing detected model inconsistencies. This is part of our future work.

Performance Evaluation (RQ 2) For the simulation, we extended the existing DOPLER variability model [Rabiser 2015] with additional elements through duplication of existing elements, *i.e.*, we generated additional decisions and assets to produce a model with overall 1000 assets and 400 decisions, which however has the same structure as the manually created original model. This by far exceeds the typical size of decision models which can be expected in a typical industrial scenario [Dhungana 2011]. We assessed whether the inconsistencies where detected, instrumented the Consistency Checker to measure evaluation times and eventually cal-

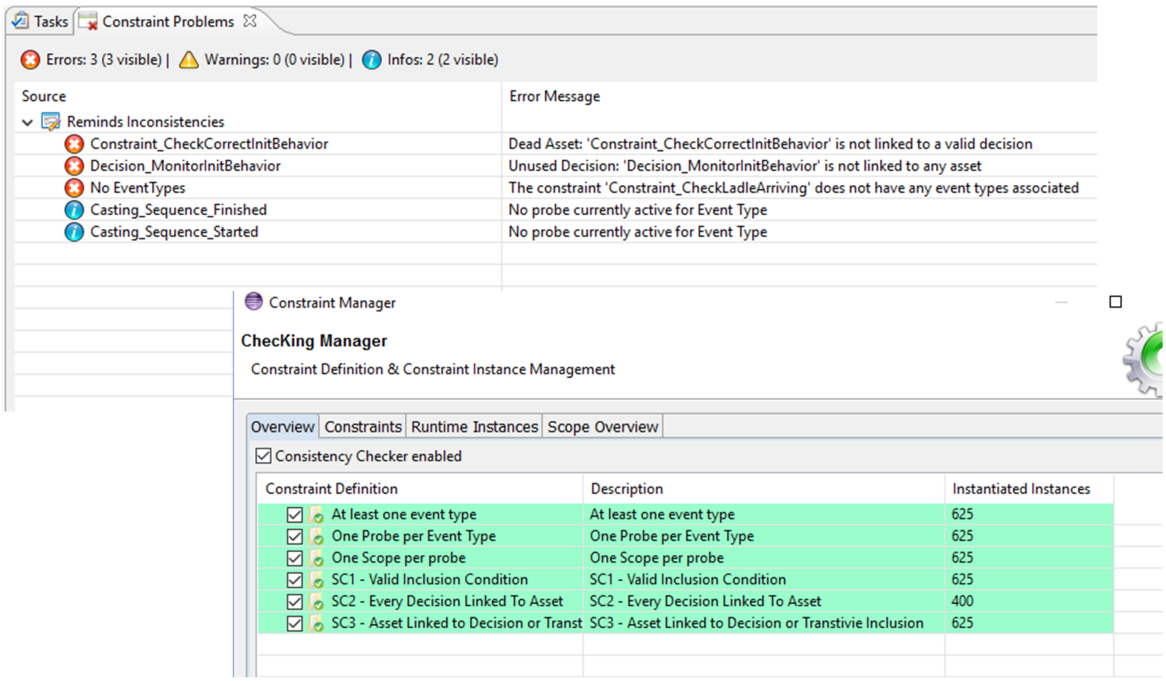


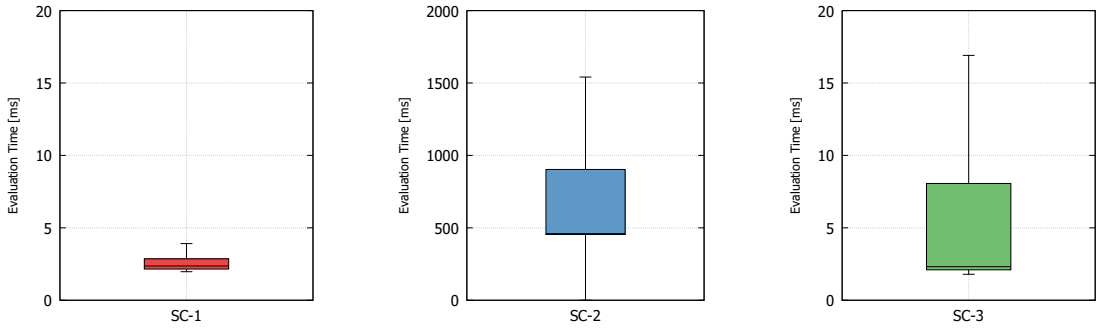
Figure 2.9: DOPLER IDE showing constraint violations (upper part) and currently activated constraints (lower part).

culated the average evaluation time per scenario. In this case, the evaluation time is the time required to evaluate a single constraint instance, more precisely, the time the method `evaluate()` needs to complete, return 'consistent' or 'inconsistent', and generate violations which are then forwarded to the user interface. We performed the evaluation runs using the latest version of the DOPLER IDE and Eclipse 3.8 on a standard Desktop machine with an Intel® Core™ i5 CPU @2.60GHz 16GB RAM running Windows 10 64-Bit.

For all three scenarios all seeded inconsistencies were detected. Depending on the type of check performed, the evaluation times vary between 3ms and 600ms on average (3.0ms for SC-1, 609ms for SC-2, and 6.74ms for SC-3). Details on the evaluation times for the three scenarios are presented in Figure 2.10 (a–c). For SC-1 and SC-3 the average evaluation times per constraint instance are below 7ms while for SC-2 the average evaluation time rises above 600ms. This increased evaluation time is due to the fact that the constraint for this scenario requires evaluating all mappings between problem and solution space, *i.e.*, all assets linked to decisions. DOPLER was not optimized for querying such mappings in the first place resulting in the need to iterate over all assets contained in the variability model to evaluate the constraint. This is the reason for the higher evaluation time for this constraint. Optimizing the access to decisions and assets in DOPLER would greatly reduce the resulting evaluation time and could easily resolve this issue. However, in all the three scenarios, the time needed to report the inconsistency to the user is still acceptable and allows for (almost) instant feedback.

2.4.2.3 Industrial Applicability

To assess the applicability of our proposed architecture (RQ 3), we instantiated it for a real-world DSPL, *i.e.*, MoldingCompany's automation software system for injection molding machines. Plastic products range from small everyday life-products such as toothbrushes or toys, up to big products such as water pipes or garbage containers. Injection molding is a manufacturing method where material (*e.g.*, thermoplastic polymer) is heated until it is molten and injected into a mold



(a) REMINDS - SC1 (b) REMINDS - SC2 (c) REMINDS - SC3

Figure 2.10: Average consistency constraint evaluation times when detecting seeded inconsistencies (1 out of 100 change operations) for 1000 performed changes for 3 scenarios for the ReMinds implementation of our reference architecture.

cavity, where the part is cooled and hardened. Injection molding machines are widely used to produce plastic products for many different markets. Example areas include automotive (car body, interior, glazing), packaging (containers, buckets, pallets), medical (healthcare, diagnostics) or teletronics (mobile communication, displays).

In the injection molding industry, various peripheral devices with different capabilities are mounted on a single machine to satisfy different requirements. For instance, in the temperature control process, specific parts of the machine have to be heated to a defined temperature, while other machine parts have to be cooled down. This can be achieved with peripheral temperature control devices provided by external manufacturers. For many years, the approach to connect peripheral devices has been to use a serial interface and a standardized, manufacturer-dependent protocol. A few years ago, the Euromap council proposed the standardized interface based on the OPC Unified Architecture (OPC UA) to better cope with this situation. OPC UA is a machine-to-machine communication protocol for industrial automation developed by the OPC Foundation[6]. The standard Euromap 82 and Euromap 82.1 define an OPC UA Information model, *e.g.*, for temperature control devices and peripheral devices in general. MoldingCompany uses OPC UA to connect and switch among multiple peripheral devices via an Ethernet connection. This allows for dynamic variability (*i.e.*, reconfiguration at runtime) and provides the technical foundation for applying our DSPL architecture in an industrial context. As part of our evaluation, an engineer of MoldingCompany created a feature model describing the Temperature Control Device (TCD) capabilities for MoldingCompany's injection molding machines. According to this model (and its mapping to OPC UA nodes) we can dynamically identify which OPC UA nodes are available on a connected device (nodes are defined as mandatory/optional). When specific nodes are present, features can be enabled. The values of OPC UA nodes are the basis for adaptation rule checks performed during runtime.

Figure 2.11 depicts our DSPL reference architecture instantiated for the MoldingCompany case study. The engineer used the same approach as we used for the CPS DSPL described above in Section 2.4.2.1. We describe the feature model and the adaptation rules the industrial engineer developed using our reference architecture. We cannot provide details of the component model and the mapping of features to components due to non-disclosure agreements, but we present general numbers and specific examples below. The middleware implemented by the engineer uses the adaptation rules and the component model to perform (re-)configuration of the system at runtime. The other parts of the reference architecture are the same as for the CPS instance described above.

[6] https://opcfoundation.org/

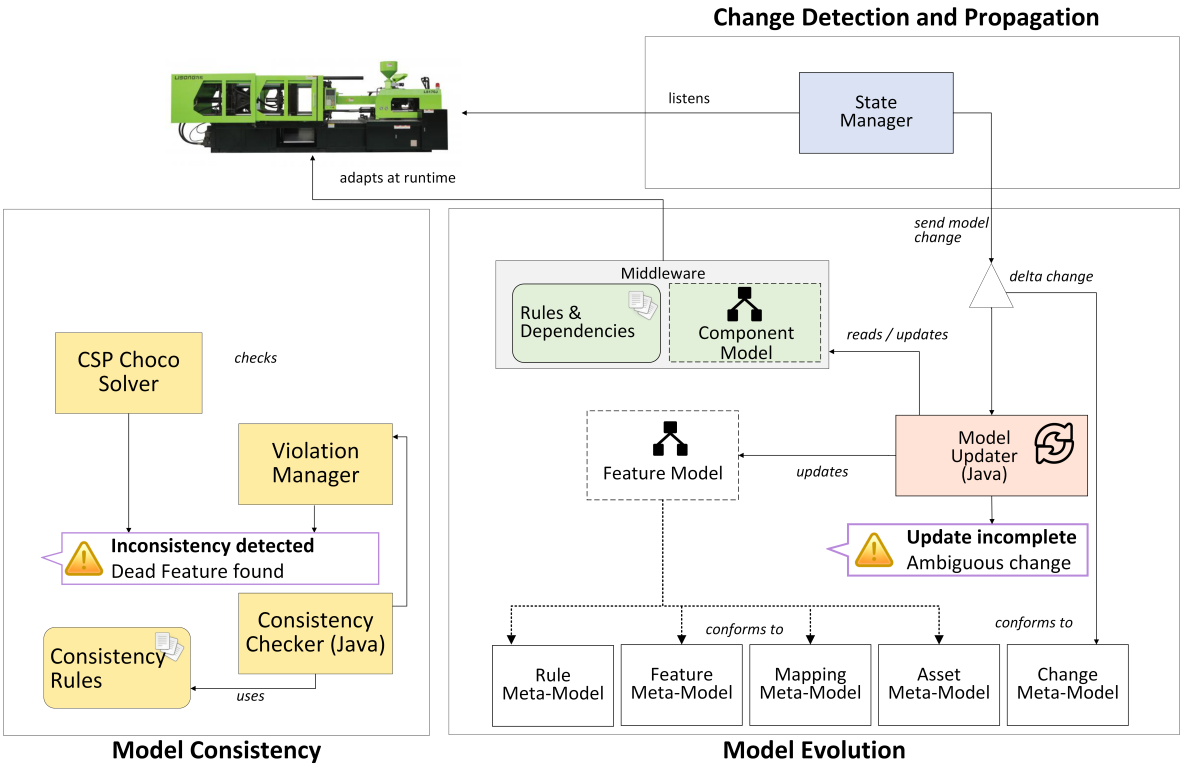


Figure 2.11: Implementation of our reference architecture for the injection molding automation system of MoldingCompany.

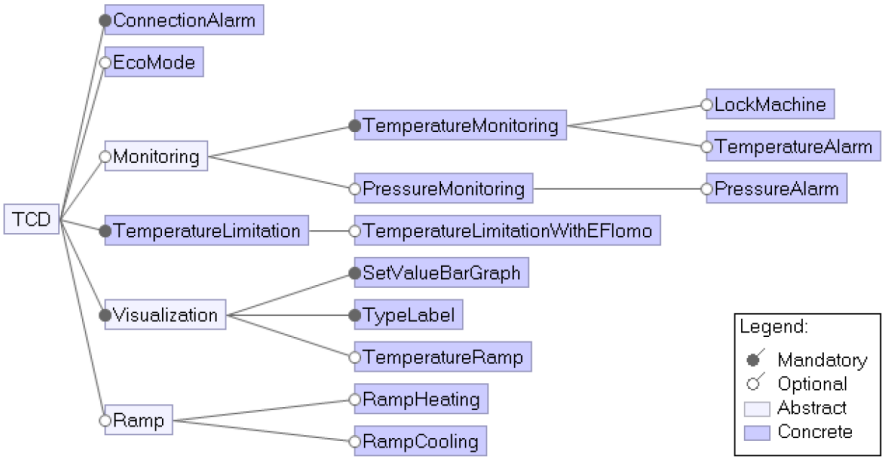


Figure 2.12: Feature Model for injection molding automation system temperature control devices DSPL.

Figure 2.12 depicts the feature model for the temperature control device and Figure 2.13 shows two example adaptation rules. The first rule is for the temperature alarm feature. Given that temperature monitoring is available, it can trigger alarms of different severity (*i.e.*, a positive integer value greater than zero). The second rule is related to the EcoMode feature, which allows

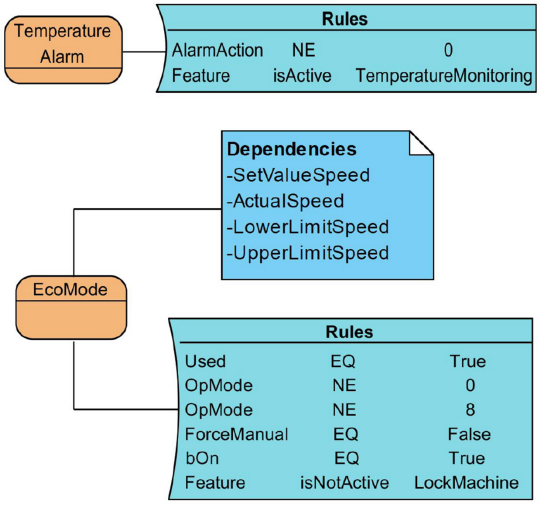


Figure 2.13: Two example adaptation rules based on features of the temperature control devices feature model.

to dynamically determine the best valve position for the water distribution system and the TCD pump speed set value to ensure optimal power usage. When the EcoMode feature is activated, the pump speed is determined automatically and the input field on the visualization is deactivated, indicating that it can currently not be changed manually. The adaptation rule defines that the connected TCD must be set to Used (activated by the operator) and the OperatingMode must not be 0 (switched off/ready) or 8 (connection problem/undefined). The ForceManual Boolean, which would deactivate the automatic calculation if set to true, must be false. Finally, the water distribution system component must be switched on, which is determined by the Boolean value bOn. Overall, the engineer created 25 adaptation rules (for 15 features), which are in turn mapped to 78 solution space components. While we did not use the entire reference architecture in this example due to the focus on the modeling and adaptation capabilities, we were able to reuse the consistency checking and model update parts from the CPS example. We received initial positive feedback, regarding the approach, from MoldingCompany. Specifically, they confirmed that without the DSPL approach it took developers several days until a prototype was running for every new device. The DSPL architecture, on the other hand, reduced the time necessary to implement a similar device of an established product line (*e.g.*, a new TCD of a different manufacturer) to approximately one working day. Our experience with our industrial partner thus shows the usefulness and applicability of our approach in a real-world scenario from the injection molding machines domain, in addition to the two DSPLs previously described.

2.5 Discussion

Our evaluation demonstrates that our reference architecture was a useful basis to implement support for DSPL evolution in at least two different cases (RQ 1) in two different domains. Both implementations exhibit significant differences: one uses a model-driven, feature-based approach (CPS) and the other one a tool-driven, decision-oriented approach (ReMinds). However, both implementations rely on Eclipse (RCP or EMF) and Java, and different technologies – specifically, technologies not supporting component-based or object-oriented development – could make the implementation of our reference architecture significantly more difficult. In the two presented cases, based on our reference architecture and the proposed generic operations, each implementation has been done by one person in a rather short time, *i.e.*, around two weeks.

We also provided initial evidence on the practicality of our reference architecture by implementing a real-world DSPL from the domain of automation software for injection molding machines, which was not done by the developers of the reference architecture, but by an industrial engineer. He could reuse several components from the CPS DSPL, which further demonstrates its feasibility.

Overall, the reference architecture was very useful to guide the implementation of evolution support for three different DSPLs in different domains. This was only possible because we kept the description of the reference architecture components, their interactions, and the operations rather abstract. However, this also has the drawback, that the implementation effort for each DSPL is significant, even if one can reuse existing models and tools as described above. It would thus make sense to further automate this process of implementing the reference architecture, *e.g.*, by providing several template implementations for different types of variability models. Also, the actual reference architecture implementation process needs to be better formalized. Even though the reference architecture components and the generic operations support the implementation, the concrete activities and their inputs and outputs should be made more explicit.

Internal Validity. Since both approaches build on prior work and tools, we cannot claim that all parts have been implemented from scratch. However, both the original solutions have been adapted and extended significantly to fit the needs of the CPS and ReMinds DSPLs. In both cases, it took less than two weeks to implement the components State Manager and Model Updater, and the facades to interact with the running systems. We implemented extensions to our own architecture basing our work on existing tool environments that have been published before [Quinton 2016, Dhungana 2011] and using the reference architecture to guide our extensions of these existing environments. While other developers could follow a different implementation approach, we still think we could sufficiently demonstrate the flexibility and practicality of our approach. Our reference architecture worked for two quite different modeling approaches. While we cannot guarantee it would work for any given approach or technology, we believe that following the architecture can guide developers in creating DSPL evolution support as we could initially demonstrate with our industrial applicability study.

Considering the evaluation runs, we measured how well our consistency checker implementations work, which indicates their scalability. However, we randomly generated large variability models – though based on real, smaller models such as the one described in [Rabiser 2015] – which might still not correctly reflect how such models would really look like in practice. Also, we randomly seeded selected inconsistencies in the three evolution scenarios representing one edit per space. In practice, many more scenarios are possible and might occur in all the three spaces (even in combination), making the exhaustiveness of the experiments difficult to reach. We thus cannot claim our approach scales in any possible case, but we can still argue that we found initial evidence for its scalability and flexibility.

External validity refers to how well data, processes, theories can be applied to other domains and application scenarios and how generalizable the results and findings are. With the presented reference architecture we aim to provide a generic architecture that can be applied to different domains and technologies. Applying our reference architecture to other approaches will require detailed knowledge about the specific system and the respective variability management approach. In terms of industrial applications, we are confident that the three instances of the reference architecture (one of which was a large industrial system) provide evidence that it can be easily adopted for different types of systems and different types of DSPL, meaning different types of variability modeling and management approaches. Regarding our evaluation, we have performed lab experiments for two different implementations of the reference architecture both representing large-scale systems. While we can not claim full generalizability in terms of performance and scalability of the approach, we think that this provides a solid basis for the applicability of the approach. Further work will be needed assessing the applicability and performance characteristics for different types of systems.

2.6 Summary

In this chapter, we explained how inconsistencies may arise when dealing with DSPL evolution. To deal with such inconsistencies, we introduced a flexible approach to support DSPL evolution based on a reference architecture, that can be implemented to support the evolution of a concrete (type of) DSPL. The reference architecture describes the capabilities needed for detecting changes made to running systems, for automating the update of variability models, and for detecting inconsistencies among all the relevant elements of a DSPL. Specifically, our reference architecture guides the development of change detection, model update, and consistency checking solutions to support evolution in a concrete DSPL. We described the implementation of evolution support for two different DSPL based on the reference architecture and reported on performance experiments we conducted. We then showed how our proposed reference architecture was implemented for a real-world automation software system from the injection molding domain by an engineer of MoldingCompany.

Material related to this chapter

- **Evolution in Dynamic Software Product Lines**. Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, Christian Schuhmayer. In Journal of Software Evolution & Process, vol. 33, issue 2, 2020.
- **Dynamically Evolving the Structural Variability of Dynamic Software Product Lines**. Luciano Baresi, Clément Quinton. In 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015. *Core rank: A*.
- **Evolution in Dynamic Software Product Lines: Challenges and Perspectives**. Clément Quinton, Rick Rabiser, Michael Vierhauser, Paul Grünbacher, Luciano Baresi. In 19th International Software Product Line Conference, SPLC 2015.

Chapter 3

Adaptation

Contents

As seen in the previous chapter, variability undergoes evolution. Consequently, novel capabilities (*e.g.*, new features or removed constraints) may be offered for a running system to adapt. Together with colleagues from University of Duisburg-Essen, Germany, we thus conducted research to assess the impact of evolution on runtime adaptation of configurable systems. In particular, we investigated the integration of reinforcement learning to evaluate such impact. This chapter covers this work.

3.1 Background and Motivation

Case Study. DSPL are well-suited to manage self-adaptive systems *i.e.*, systems that adapt to context changes dynamically. As seen in the previous chapter, feature models are the de facto standard for specifying the variability of a DSPL, and define all permissible reconfigurations (the self-adaptive system's adaptation space) enabling the running system to adapt to changes in its environment. Figure 3.1 shows the feature model of a self-adaptive web service as a running example.

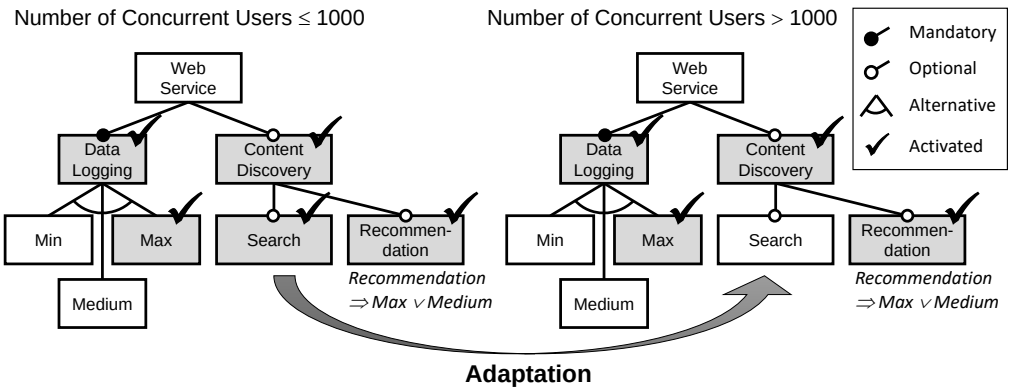


Figure 3.1: Feature model and adaptation of example web service

The DataLogging feature is mandatory, while the ContentDiscovery feature is optional. The DataLogging feature has three alternative sub-features, *i.e.*, at least one data logging sub-feature must be active: Min, Medium or Max. The ContentDiscovery feature has two optional sub-features Search and Recommendation. The cross-tree constraint Recommendation $\Rightarrow$ Max $\vee$ Medium specifies that a sufficient level of data logging is required to collect enough information about the web service's users and transactions to make good recommendations. Let us consider that the above web service should adapt to a changing number of concurrent users to keep its response time below 500ms. A software engineer may express an adaptation rule for the web service such that it turns off some of its features in the presence of more users, thereby reducing the resource needs of the service. The right-hand side of Figure 3.1 shows a concrete example for such an adaptation. If the service faces an environment state of more than 1,000 concurrent users, the service self-adapts by deactivating the Search feature.

Reinforcement Learning. *Online reinforcement learning* is a suitable approach to realize self-adaptive systems in the presence of design time uncertainty. Online reinforcement learning means that machine learning is employed at runtime for the system to learn from actual operational data and thereby leverage information only available at runtime. In general, reinforcement learning aims to learn suitable actions via an agent interacting with its environment as follows: *(i)* the agent receives a reward for executing an action, *(ii)* this reward expresses how suitable that action was [Sutton 2018]. The goal of reinforcement learning is to optimize cumulative rewards *i.e.*, actions should be selected that have shown to be suitable, which is known as *exploitation*. However, to discover such actions in the first place, actions that were not selected before should be selected, which is known as *exploration*. How exploration happens has an impact on the performance of the learning process [Bu 2013, Filho 2017, Sutton 2018].

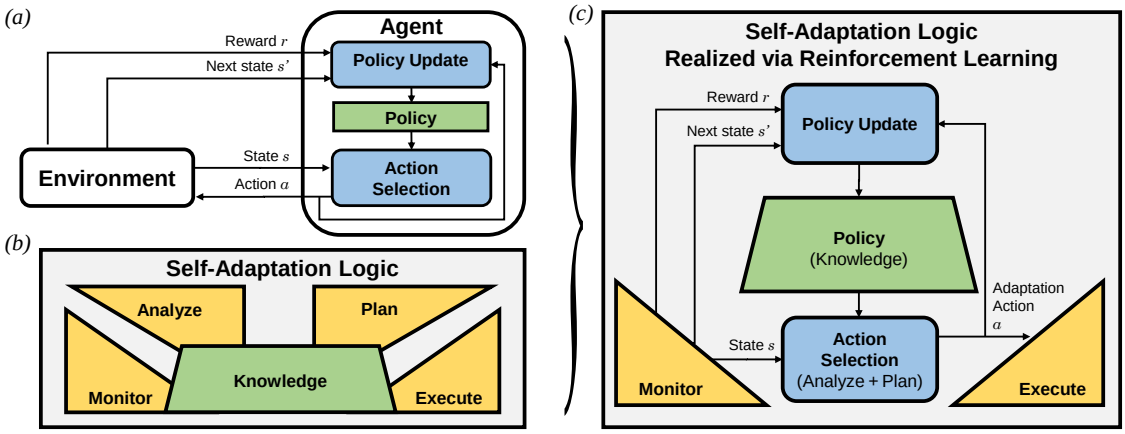


Figure 3.2: Integration of reinforcement learning into the MAPE-K reference model: *(a)* basic reinforcement learning model, *(b)* MAPE-K model, *(c)* integrated model

Figure 3.2*(a)* illustrates agent's interactions in a reinforcement learning process. At a given time step t, the agent selects an action a from its adaptation space to be executed in environment state s. As a result, the environment transitions to s' at time step $t+1$ and the agent receives a reward r for executing the action. The reward r together with the information about the next state s' are used to update the action selection policy of the agent. The goal of reinforcement learning is to optimize cumulative rewards. As explained above, a trade-off between exploitation (using current knowledge) and exploration (gathering new knowledge) must be made when selecting an action. A self-adaptive system can conceptually be structured into two main elements [Kephart 2003, Salehie 2009]: the *system logic* (aka. the managed element) and the *self-adaptation logic* (aka. the autonomic manager). To understand how reinforcement learning can be leveraged for realizing the self-adaptation logic, we use the well-established MAPE-K refer-

ence model for self-adaptive systems [De Lemos 2013, Weyns 2021]. As depicted in Figure 3.2*(b)*, MAPE-K structures the self-adaptation logic into four main conceptual activities that rely on a common *knowledge* base [de la Iglesia 2015]. These activities *monitor* the system and its environment, *analyze* monitored data to determine adaptation needs, *plan* adaptation actions, and *execute* these adaptation actions at runtime. Figure 3.2*(c)* depicts how the elements of reinforcement learning are integrated into the MAPE-K loop. For a self-adaptive system, "agent" refers to the self-adaptation logic of the system and "action" refers to an adaptation action [Palm 2020]. In the integrated model, *action selection* of reinforcement learning takes the place of the *analyze* and *plan* activities of MAPE-K. The learned *policy* takes the place of the self-adaptive system's *knowledge* base. At runtime, the policy is used by the self-adaptation logic to select an adaptation action a based on the current state s determined by *monitoring*. The action selected using the policy may be either to leave the system in the current state (*i.e.*, no need for adaptation), or a specific adaptation, which is then *executed*.

Open Issues. Existing online reinforcement learning solutions for self-adaptive systems propose randomly selected adaptation actions for exploration. The effectiveness of exploration therefore directly depends on the size of the adaptation space, because each adaptation action has an equal chance of being selected. Self-adaptive systems may have large, *discrete* adaptation spaces. In the presence of such large, discrete adaptation spaces, random exploration may lead to slow learning at runtime [Bu 2013, Filho 2017, Sutton 2018]. Some reinforcement learning algorithms can cope with a large space of actions, but require that the space of actions is continuous in order to generalize over unseen actions [Nachum 2017]. Existing online reinforcement learning solutions are also unaware of system evolution [Kinneer 2018]. Yet, due to evolution, the adaptation space may change, *e.g.*, existing adaptation actions may be removed or new adaptation actions may be added. Some reinforcement learning algorithms can cope with environments that change over time (non-stationary environments) [Nachum 2017, Sutton 2018]. However, they cannot cope with changes of the adaptation space. Existing solutions thus explore new adaptation actions only with low probability (as all adaptation actions have an equal chance of being selected), and thus may take quite long until new adaptation actions have been explored.

3.2 Feature-Model-guided Exploration

To address the issues discussed above, we proposed novel exploration strategies for online reinforcement learning. Our exploration strategies use feature models to give structure to the system's adaptation space and thereby leverage additional information to guide exploration. An adaptation action is represented by a valid feature combination specifying the target run-time configuration of the system. Our strategies traverse the system's feature model to select the next adaptation action to be explored. In addition, our strategies detect added and removed adaptation actions by analyzing the differences between the feature models of the system before and after an evolution step. Adaptation actions removed as a result of evolution are no longer explored, while added adaptation actions are explored first. We first explain how these FM-guided exploration strategies can be integrated into existing reinforcement learning algorithms. Thereby, we also provide a realization of the integrated conceptual model from Section 3.1. We then introduce the realization of the actual FM-guided exploration strategies.

3.2.1 Integration into Reinforcement Learning

We used two well-known reinforcement learning algorithms for integrating our FM-guided exploration strategies: Q-Learning and SARSA. We chose Q-Learning as it is the most widely used algorithm in the literature [Barrett 2013, Moustafa 2014, Caporuscio 2016, Arabnejad 2017, Wang 2017, Zhao 2017, Moustafa 2018, Wang 2019, Shaw 2022] and SARSA, as it differs from Q-Learning with respect to how the knowledge is updated during learning. Q-Learning (an *off*-policy algorithm) updates the knowledge based on selecting the next action which has the highest

Algorithm 1 Q-Learning with FM-guided Exploration

```
1: function FMQ-Learning(FeatureModel M; Double α, γ, ε_d, δ_d)
2:     Initialize Q(s,a) with lowest possible reward ∀s ∈ S (state space), ∀a ∈ A (adaptation space);
3:     Determine current state s; ε ← 1; δ ← 1;
4:     repeat
5:         Set<Feature> a = getNextAction(M, s); // Action Selection
6:         Adapt service to configuration a; Observe reward r; Observe new state s';
7:         Q(s,a) ← Q(s,a) + α[r + γmax_{a'∈A}Q(s',a') − Q(s,a)]; // Knowledge Update
8:         s ← s'; ε ← ε · ε_d; δ ← δ · δ_d;
9:     until last time step
10: end function
11:
12: function getNextAction(FeatureModel M, State s)
13:     Set<Feature> a ← argmax_a Q(s,a); // Exploit existing knowledge
14:     InitFMExploration(M, a); // initialize the FM-guided strategies, see Algorithm 3
15:     if random() < ε then // Explore new actions
16:         if random() < δ then return getRandomConfiguration(M);
17:         else return getNextConfiguration(); // see Algorithm 3
18:         end if
19:     end if
20:     return a;
21: end function
```

expected reward [Sutton 2018]. SARSA (an *on*-policy algorithm) updates the knowledge based on selecting the next action by following the already learned action selection policy. As a result, Q-Learning tends to perform better in the long run, but SARSA is better in avoiding expensive adaptations. To summarize, if, for a given system, executing "wrong" adaptations is expensive, then SARSA is more appropriate, otherwise Q-Learning is preferable.

Algorithm 1 shows the extended Q-Learning algorithm. A value function $Q(s,a)$ represents the learned knowledge, which gives the expected cumulative reward when performing an action a in a state s [Sutton 2018]. There are two hyper-parameters: the learning rate α, which defines to what extent newly acquired knowledge overwrites old knowledge, and the discount factor γ, which defines the relevance of future rewards. After the initialization (lines 2-3), the algorithm repeatedly selects the next action (line 5), performs the action and observes its results (line 6), and updates its learned knowledge and other variables (lines 7-8). Algorithm 2 shows the extended SARSA algorithm, which follows a similar logic. However, while Q-Learning updates the knowledge by selecting the action with the highest Q value (Algorithm 1, line 7), SARSA selects the action according to the current policy (Algorithm 2, line 8).

Our strategies are integrated into reinforcement learning in the getNextAction function, which selects the next adaptation action while trading off exploration and exploitation. We use the ε-greedy strategy as a baseline, as a standard action selection strategy in reinforcement learning, widely used in the related work. With probability $1-\varepsilon$, ε-greedy exploits existing knowledge, while with probability ε, it selects a random action. In contrast to random exploration, we use our FM-guided exploration strategies by calling the getNextConfiguration function (Algorithm 1, line 17). To prevent FM-guided exploration from prematurely converging to a local minimum, we follow the literature and use a little randomness [Plappert 2018], *i.e.*, perform random exploration with probability $\delta \cdot \varepsilon$ (lines 15-16). Here, $0 \leq \delta \leq 1$ is the probability for choosing a random action, given that we have chosen to perform exploration. To facilitate convergence of the learning process, we use the ε-decay approach. This is a typical approach in reinforcement learning, which starts at $\varepsilon = 1$ and decreases it at a predefined rate ε_d after each time step. We also follow this decay approach for the FM-guided strategies to incrementally decrease δ with rate δ_d.

Algorithm 2 SARSA with FM-guided Exploration

```
1: function FMSARSA(FeatureModel M; Double α, γ, ε_d, δ_d)
2:     Initialize Q(s, a) with lowest possible reward ∀s ∈ S (state space), ∀a ∈ A (adaptation space);
3:     Determine current state s; ε ← 1; δ ← 1;
4:     Set<Feature> a = getNextAction(M, s); // Action Selection
5:     repeat
6:         Adapt service to configuration a; Observe reward r; Observe new state s';
7:         Set<Feature> a' = getNextAction(M, s');
8:         Q(s, a) ← Q(s, a) + α[r + γQ(s', a') − Q(s, a)]; // Knowledge Update
9:         s ← s'; a ← a'; ε ← ε · ε_d; δ ← δ · δ_d;
10:    until last time step
11: end function
```

3.2.2 FM-Structure Exploration for Large Adaptation Spaces

To capture large adaptation spaces, we proposed the FM-structure exploration strategy, which takes advantage of the semantics typically encoded in the structure of feature models. Non-leaf features are typically abstract features used to better structure variability [Thüm 2011]. Abstract features do not directly impact the implementation, but delegate their implementation to their sub-features. Sub-features thereby offer different implementations of their abstract parent feature. As such, the sub-features of a common parent feature, *i.e.*, *sibling* features, can be considered semantically connected.

In the feature model depicted in Figure 3.1, the ContentDiscovery feature has two sub-features Search and Recommendation offering different concrete ways how a user may discover online content. The idea behind FM-structure exploration is to exploit the information about these potentially semantically connected sibling features and explore them first before exploring other features[1]. Table 3.1 shows an excerpt of a typical exploration sequence of the FM-structure exploration strategy with the step-wise exploration of sibling features highlighted in gray. Exploration starts with a randomly selected leaf feature, here: Recommendation. Then all configurations involving this leaf feature are explored before moving to its sibling feature, here: Search.

	Logging	Min	Medium	Max	Content Disc.	Search	Recommend.
Start	✓			✓	✓		✓
1	✓		✓		✓		✓
2	✓		✓		✓	✓	✓
3	✓			✓	✓	✓	✓
4	✓		✓		✓	✓	
5	✓	✓			✓	✓	
	...	...	...	...	...	...	...

Table 3.1: Example for FM-structure exploration (excerpt)

FM-structure exploration is realized by Algorithm 3, which starts by randomly selecting an arbitrary leaf feature f among all leaf features that are part of the current configuration (lines 5–6). Then, the set of configurations $\mathcal{C}_f$ containing feature f is computed, while the sibling features of feature f are gathered into a dedicated *siblings* set (line 7). While $\mathcal{C}_f$ is non-empty, the strategy explores one randomly selected configuration from $\mathcal{C}_f$ and removes the selected configuration from $\mathcal{C}_f$ (lines 11–13). If $\mathcal{C}_f$ is empty, then a new set of configurations containing a sibling feature of f is randomly explored, provided such sibling feature exists (lines 15–17). If no configuration containing f or a sibling feature of f is found, the strategy moves on to the parent feature of f, which is repeated until a configuration is found (line 13) or the root feature is reached (line 22).

[1] Note that this entails a random selection of the order of sub-features.

Algorithm 3 FM-Structure Exploration Strategy

```
1: Set<Feature> leaves, configuration, siblings;
2: Set<Set<Feature>> C_f; Feature f;
3:
4: function InitFMExploration(FeatureModel M, Set<Feature> currentConfiguration)
5:     leaves ← getLeaves(currentConfiguration);
6:     f ← randomSelect(leaves);
7:     C_f ← getConfigurationsWithFeature(f); siblings ← siblings(f);
8: end function
9:
10: function getNextConfiguration()
11:     if C_f ≠ ∅ then
12:         configuration ← randomSelect(C_f); C_f ← C_f \ {configuration};
13:         return configuration;
14:     else
15:         if siblings ≠ ∅ then
16:             f ← randomSelect(siblings);
17:             siblings ← siblings \ {f}; C_f ← getConfigurationsWithFeature(f);
18:         else
19:             if parent(f) ≠ ∅ then
20:                 f ← parent(f); siblings ← siblings(f);
21:                 C_f ← getConfigurationsWithFeature(f);
22:             else // Root feature reached
23:                 return ∅;
24:             end if
25:         end if
26:         return getNextConfiguration();
27:     end if
28: end function
```

3.2.3 FM-Difference Exploration Strategy for System Evolution

To capture changes in the system's adaptation space due to system evolution, we proposed the FM-difference exploration strategy, which leverages the differences in feature models before ($\mathcal{M}$) and after ($\mathcal{M}'$) an evolution step. Following the product line literature, we considered two main types of feature model differences [Thüm 2009]:

Added configurations (feature model generalization). New configurations may be added to the adaptation space by (*i*) introducing new features to $\mathcal{M}'$, or (*ii*) removing or relaxing existing constraints (*e.g.*, by changing a sub-feature from mandatory to optional, or by removing cross-tree constraints). In our running example, a new sub-feature Optimized might be added to the DataLogging feature, providing a more resource efficient logging implementation. Thereby, new configurations are added to the adaptation space, such as {DataLogging, Optimized, ContentDiscovery, Search}. As another example, the Recommendation implementation may have been improved and it now can work with the Min logging feature. This removes the cross-tree constraint shown in Figure 3.1, and adds new configurations such as {DataLogging, Min, ContentDiscovery, Recommendation}.

Removed configurations (feature model specialization). Symmetrical to above, configurations may be removed from the adaptation space by (*i*) removing features from $\mathcal{M}$, or (*ii*) by adding or tightening constraints in $\mathcal{M}'$.

To determine these changes of feature models, we compute a set-theoretic difference between valid configurations expressed by feature model $\mathcal{M}$ and feature model $\mathcal{M}'$. Detailed descriptions of feature model differencing as well as efficient tool support can be found

in [Acher 2012, Bürdek 2016]. The feature model differences provide us with adaptation actions added to the adaptation space ($\mathcal{M}' \setminus \mathcal{M}$), as well as adaptation actions removed from the adaptation space ($\mathcal{M} \setminus \mathcal{M}'$). Our FM-difference exploration strategies first explore the configurations that were added to the adaptation space, and then explore the remaining configurations if needed. The rationale is that added configurations might offer new opportunities for finding suitable adaptation actions and thus should be explored first. Configurations that were removed are no longer executed and thus the learning knowledge can be pruned accordingly. In the reinforcement learning realization (see Section 3.2.1), we remove all tuples (s, a) from Q, where a represents a removed configuration. FM-difference exploration can be combined with FM-structure exploration, but also with ε-greedy. In both cases, this means that instead of exploring the whole new adaptation space, exploration is limited to the set of new configurations.

3.3 Empirical Evaluation

3.3.1 Research Questions

We experimentally assessed our FM-guided exploration strategies and compared them with ε-greedy as the strategy used in the literature. In particular, we aimed to answer the following research questions:

RQ 1: *How does learning performance and system quality using FM-structure exploration (from Sect. 3.2.2) compare to using ε-greedy?*

RQ 2: *How does learning performance and system quality using FM-difference exploration (from Sect. 3.2.3) compare to evolution-unaware exploration?*

Subject Systems. Our experiments build on two real-world systems and datasets. The CloudRM system is an adaptive cloud resource management service offering 63 features, 344 adaptation actions, and a feature model that is 3 levels deep. The BerkeleyDB-J system is an open source reconfigurable database written in Java with 26 features, 180 adaptation actions and 5 levels.

CloudRM System. CloudRM [Mann 2016] controls the allocation of computational tasks to virtual machines (VMs) and the allocation of virtual machines to physical machines in a cloud data center[2]. CloudRM can be adapted by reconfiguring it to use different allocation algorithms, and the algorithms can be adapted by using different sets of parameters. We implemented a separate adaptation logic for CloudRM by using the extended learning algorithms as introduced in Section 3.2.1. We defined the reward function as $r = -(\rho \cdot e + (1-\rho) \cdot m)$, with energy consumption e and number of VM manipulations m (*i.e.*, migrations and launches), each normalized to be within $[0, 1]$. We used $\rho = 0.8$, meaning we give priority to reducing energy consumption, while still maintaining a low number of VM manipulations. Our experiments are based on a real-world workload trace with 10,000 tasks, in total spanning over a time frame of 29 days [Mann 2018]. The CloudRM algorithms decide on the placement of new tasks whenever they are entered into the system (as driven by the workload trace). For RQ 2, the same workload was replayed after each evolution step to ensure consistency among the results. To emulate system evolution, we used a 3-step evolution scenario. Starting from a system that offers 26 adaptation actions, these three evolution steps respectively add 30, 72 and 216 adaptation actions.

BerkeleyDB-J. The BerkeleyDB-J dataset was collected by Siegmund *et al.* [Siegmund 2012] and was used for experimentation with reconfigurable systems to predict their response times[3]. We chose this system because the configurations are expressed as a feature model and the dataset includes performance measurements for all system configurations, which were measured using standard benchmarks[4]. Adaptation actions are the possible runtime reconfigurations of the system. We define the reward function as $r = -t$, with t being the response time normalized to

[2] https://sourceforge.net/p/vm-alloc/task_vm_pm

[3] https://www.se.cs.uni-saarland.de/projects/splconqueror/icse2012.php

[4] Other datasets from [Siegmund 2012] had feature models with only 1 level, had many configurations associated with the same response time, or did not include performance measurements for all configurations.

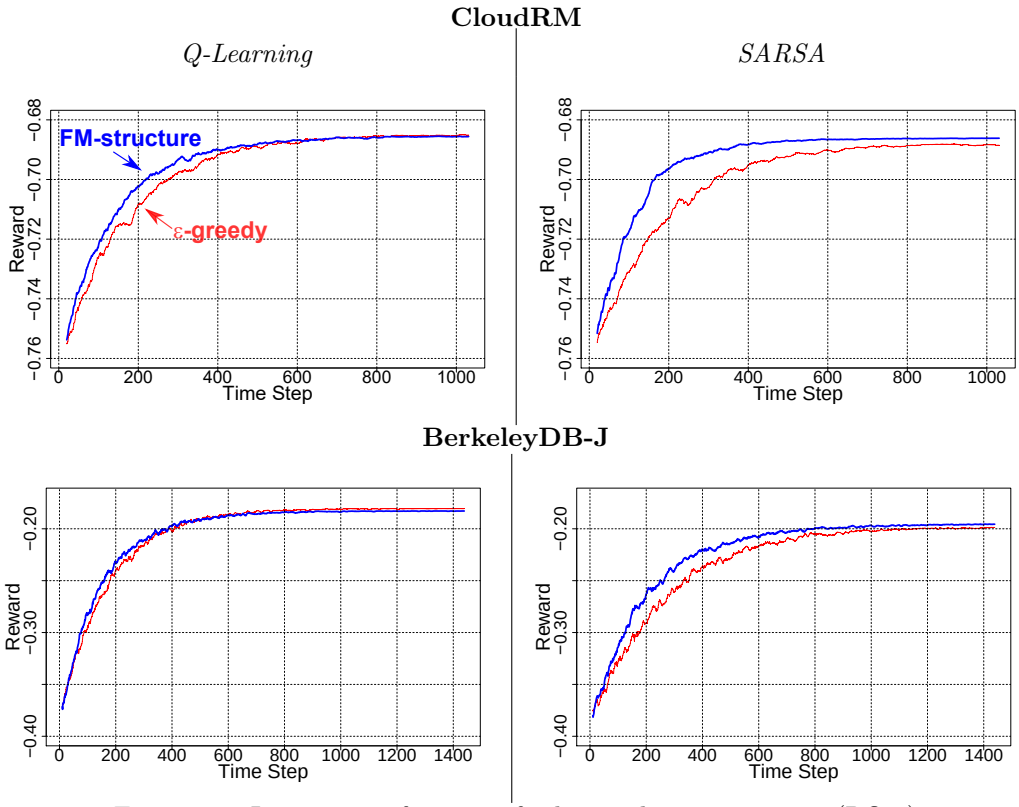


Figure 3.3: Learning performance for large adaptation spaces (RQ 1)

be within $[0, 1]$[5]. Given the smaller size of BerkeleyDB-J's adaptation space, we used a 2-step evolution scenario to emulate system evolution. We first randomly change two of the five optional features into mandatory ones, thereby reducing the size of the adaptation space. We start from this reduced adaptation space and, randomly change the mandatory features back into optional ones. Starting from a system that offers 39 adaptation actions, these two evolution steps respectively add 20 and 121 adaptation actions.

Measuring Learning Performance. We characterize the performance of the learning process by using the following metrics from [Taylor 2009]: *Asymptotic performance* measures the reward achieved at the end of the learning process; *Time to threshold* measures the number of time steps the learning process takes to reach a predefined reward threshold (in our case 90% of the difference between maximum and minimum performance); *Total performance* measures the overall learning performance by computing the area between the reward curve and the asymptotic reward. Given the stochastic nature of the learning strategies (both ε-greedy and to a lesser degree our strategies involve random decisions), we repeated the measurements 500 times and averaged the results.

Prototypical Realization. The learning algorithms, as well as the ε-greedy and FM-based exploration strategies were implemented in Java. Feature model management and analysis were performed using the FeatureIDE framework[6], which we used to efficiently compute possible feature combinations from a feature model.

Hyper-parameter Optimization. To determine suitable hyper-parameter values (see Section 3.2.1), we performed hyper-parameter tuning via exhaustive grid search for each of the subject systems and each of the reinforcement learning algorithms. We measured the learning

[5]For CloudRM, the reward function was the opposite of the weighted sum of the metrics to be minimized, where the sum of the weights is 1. Regarding BerkeleyDB-J, the same logic is applied, but since there is only one metric to be minimized, the formula becomes simpler.

[6]https://featureide.github.io/

	Asymptotic performance	Time to Threshold	Total performance	Effect on Quality	
Q-Learning		**CloudRM**		Energy	VM Manip.
ε-greedy:	-0.6851	286	-8.8023	7084	2281
FM-structure:	-0.6854	219	-5.4431	7077	2103
Improvement	*-0.04 %*	*23.43 %*	*38.16 %*	*0.10 %*	*7.80 %*
SARSA					
ε-greedy:	-0.6885	390	-11.631	10602	3398
FM-structure:	-0.6862	200	-4.9673	10578	3087
Improvement	*0.33 %*	*48.72 %*	*57.29 %*	*0.23 %*	*9.15 %*

Q-Learning		**BerkeleyDB-J**		Avg. Response Time
ε-greedy:	-0.1834	383	-31.2457	3606
FM-structure:	-0.1847	357	-28.0466	3550
Improvement	*-0.71 %*	*6.79 %*	*10.24 %*	*1.55 %*
SARSA				
ε-greedy:	-0.1993	592	-46.8978	3824
FM-structure:	-0.1958	457	-33.2211	3666
Improvement	*1.76 %*	*22.80 %*	*29.16 %*	*4.13 %*

Avg. Improv. Q-Learning	*-0.38 %*	*15.1 %*	*24.2 %*
Avg. Improv. SARSA	*1.05 %*	*35.8 %*	*43.2 %*
Total Avg. Improvement	***0.33 %***	***25.4 %***	***33.7 %***

Table 3.2: Comparison of exploration strategies for large adaptation spaces (RQ 1)

performance for our baseline ε-greedy strategy for 11,000 combinations of learning rate α, discount factor γ, and ε-decay rate. For each of the subject systems and reinforcement learning algorithms we chose the hyper-parameter combination that led to the highest asymptotic performance. We used these hyper-parameters also for our FM-guided strategies.

3.3.2 Results

Results for RQ 1 (FM-structure exploration). Figure 3.3 visualizes the learning process by showing how rewards develop over time, while Table 3.2 quantifies the learning performance using the metrics introduced above. Across the two systems and learning algorithms, FM-structure exploration performs better than ε-greedy wrt. total performance (33.7% on average) and time to threshold (25.4%), while performing comparably wrt. asymptotic performance (0.33%). A higher improvement is visible for CloudRM than for BerkeleyDB-J, which we attribute to the much larger adaptation space of CloudRM, whereby the effects of systematically exploring the adaptation space become more pronounced.

For CloudRM, FM-structure exploration consistently leads to less VM manipulations and lower energy consumption. While savings in energy are rather small (0.1% resp. 0.23%), FM-structure exploration reduces the number of virtual machine manipulations by 7.8% resp. 9.15%. This is due to the placement algorithms of CloudRM having a small difference wrt. energy optimization, but a much larger difference wrt. the number of virtual machine manipulations. For BerkeleyDB-J, we observe an improvement in response times of 1.55% (resp. 4.13%). This smaller improvement is consistent with the smaller improvement in learning performance.

Analyzing the improvement of FM-structure exploration for the different learning algorithms, we observe an improvement of 24.2% (total performance) resp. 15.1% (time to threshold) for Q-Learning, and a much higher improvement of 43.2% resp. 35.8% for SARSA. Note, however, that the overall learning performance of SARSA is much lower than that of Q-Learning. SARSA performs worse wrt. total performance (-23% on average), time to threshold (-27.6% on average), and asymptotic performance (-3.82% on average). In addition, SARSA requires around 19.4% more episodes than Q-Learning to reach the same asymptotic performance. The reason is that SARSA is more conservative during exploration [Sutton 2018]. If there is an adaptation action that leads to a large negative reward which is close to an adaptation action that leads to the optimal reward, Q-Learning exhibits the risk of choosing the adaptation action with the large negative reward. In contrast, SARSA will avoid that adaptation action, but will more slowly learn the optimal adaptation actions. So, in practice one may choose between Q-Learning and SARSA depending on how expensive it is to execute "wrong" adaptations.

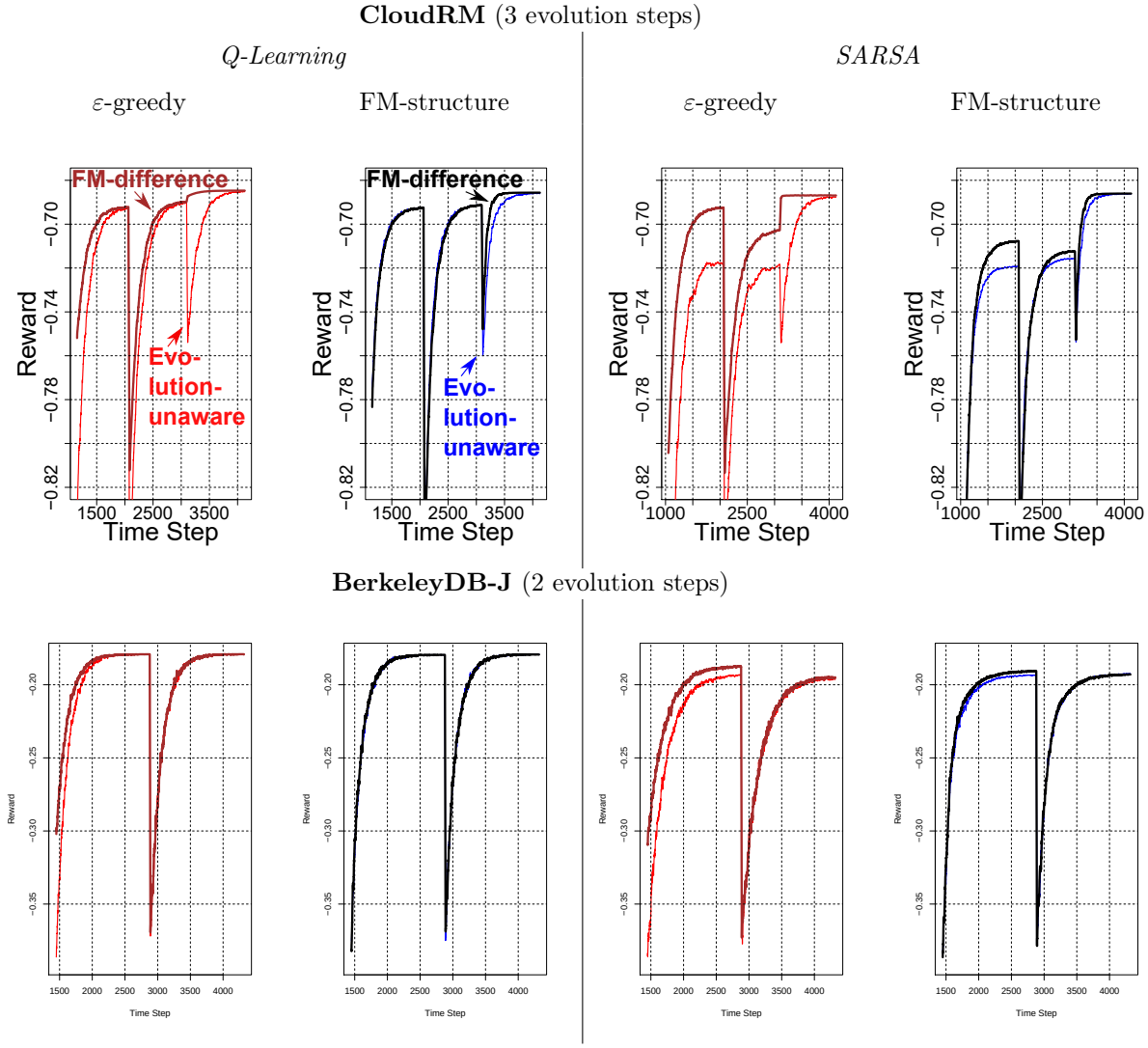


Figure 3.4: Learning performance across system evolution (RQ 2)

Results for RQ 2 (FM-difference exploration). We compared FM-difference exploration combined with ε-greedy and FM-structure exploration with their respective evolution-unaware counterparts (*i.e.*, the strategies used for RQ 1). It should be noted that even though we provided

the evolution-unaware strategies with the information about the changed adaptation space (so they can fully explore it), we have not modified them such as to differentiate between old and new adaptation actions. Like for RQ 1, Figure 3.4 visualizes the learning process, while Table 3.3 quantifies learning performance. We computed the metrics separately for each of the evolution steps and report their averages. After each evolution step, learning proceeds for a given number of time steps, before moving to the next evolution step.

	Asymptotic performance	Time to Threshold	Total performance	Effect on Quality	
Q-Learning		**CloudRM**		Energy	VM Manip.
ε-greedy:					
FM-difference	-2.0670	571	-39.1095	32393	10074
Evolution-unaware	-2.0688	1147	-84.9052	32147	13745
FM-structure:					
FM-difference	-2.0697	756	-57.5157	32351	11439
Evolution-unaware	-2.0699	866	-59.0660	32273	11798
Avg. Improvement	*0.05 %*	*57.7 %*	*59.9 %*	*-0.50 %*	*19.8 %*
SARSA					
ε-greedy:					
FM-difference	-2.1489	607	-39.3645	32566	10374
Evolution-unaware	-2.2530	2018	-117.6577	32618	15756
FM-structure:					
FM-difference	-2.1723	955	-74.8560	32660	12641
Evolution-unaware	-2.1834	723	-86.3582	32695	13911
Avg. Improvement	*2.68 %*	*104.1 %*	*107.1 %*	*0.13 %*	*30.9 %*

Q-Learning		**BerkeleyDB-J**		Avg. Response Time
ε-greedy:				
FM-difference	-0.3583	661	-52.7786	3270
Evolution-unaware	-0.3582	774	-66.2526	3348
FM-structure:				
FM-difference	-0.3589	675	-58.2346	3301
Evolution-unaware	-0.3588	693	-58.7939	3305
Avg. Improvement	*-0.02 %*	*9.88 %*	*13.2 %*	*1.24 %*
SARSA				
ε-greedy:				
FM-difference	-0.5111	999	-74.5953	3588
Evolution-unaware	-0.5465	1195	-101.2446	3741
FM-structure:				
FM-difference	-0.4685	726	-61.6969	3514
Evolution-unaware	-0.4732	818	-66.8781	3544
Avg. Improvement	*3.97 %*	*16.2 %*	*22.1 %*	*2.56 %*
Avg. Improv. ε-greedy	*2.96 %*	*92.5 %*	*94.3 %*	
Avg. Improv. FM-structure	*0.38 %*	*1.4 %*	*6.85 %*	
Total Avg. Improvement	***1.67 %***	***47 %***	***50.6 %***	

Table 3.3: Comparison of exploration strategies across evolution steps (RQ 2)

The FM-difference exploration strategies consistently performed better than their evolution-unaware counterparts wrt. total performance (50.6% on average) and time to threshold (47%), and perform comparably wrt. asymptotic performance (1.7%). Like for RQ 1, the improvements are more pronounced for CloudRM, which exhibits a larger action space than BerkeleyDB-J.

For CloudRM, FM-difference exploration reduces the number of virtual machine manipulations by 19.8% resp. 30.9%, while keeping energy consumption around the same as the non-evolution-aware strategies. For BerkeleyDB-J, FM-difference exploration leads a reduction in response time by 1.24% resp. 2.56%. Like for RQ 1, this smaller reduction is consistent with the smaller learning performance.

The improvement of FM-difference exploration is more pronounced for ε-greedy than for FM-structure exploration; *e.g.*, showing a 94.4% improvement in total performance for ε-greedy compared with an improvement of only 6.85% for FM-structure exploration. This suggests that, during evolution, considering the changes of the adaptation space has a much larger effect than considering the structure of the adaptation space. In addition, we note that due to the way we emulate evolution in our experiments, the number of adaptations introduced after an evolution step is much smaller (66 on average) than the size of the whole adaptation space of the subject systems (262 on average), thus diminishing the effect of FM-structure exploration.

Analyzing the improvement of FM-difference exploration for the different learning algorithms, we observed the same effect as for RQ 1. While FM-difference exploration shows a much higher improvement for SARSA, the overall learning performance for SARSA is much lower than for Q-Learning.

To facilitate reproducibility and replicability, our code, the used data and our experimental results are available online[7].

3.4 Discussion

Validity Risks. We used two realistic subject systems and employed real-world workload traces and benchmarks to measure learning performance and the impact of the different exploration strategies on the systems' quality characteristics. The results indicate that the size of the adaptation space may have an impact on how much improvement may be gained from FM-structure exploration.

We chose ε-greedy as a baseline, because it was the exploration strategy used in existing online reinforcement learning approaches for self-adaptive systems [Barrett 2013, Caporuscio 2016, Wang 2017, Zhao 2017, Wang 2019]. Alternative exploration strategies were proposed in the broader field of machine learning. Examples include Boltzmann exploration, where actions with a higher expected reward (e.g., Q value) have a higher chance of being explored, or UCB action selection, where actions are favored that have been less frequently explored [Sutton 2018]. A comparison among those alternatives is beyond the scope of this article, because a fair comparison would require the careful variation and analysis of a range of many additional hyper-parameters.

Completeness of Feature Models. We assume that feature models are complete with respect to the coverage of the adaptation space and that during an evolution step they are always consistent and up to date. A further possible change during service evolution can be the modification of a feature's implementation, which is currently not visible in the feature models. Encoding such kind of modification thus could further improve our FM-guided exploration strategies.

Structure of Feature Models. One aspect that impacts FM-structure exploration is how the feature model is structured. As an example, if a feature model has only few levels (and thus little structure), FM-structure exploration behaves similar to random exploration, because such a "flat" feature model does not provide enough structural information. On the other hand, providing reinforcement learning with too much structural information might hinder the learning process. As case in point, we realized during our experiments that the alternative FM-structure

[7] https://gitlab.com/cquinton/fmlearning

exploration strategy from our earlier work [Metzger 2020] indeed had such negative effect for the BerkeleyDB-J system. This alternative strategy used the concept of "feature degree"[8] to increase the amount of structural information used during learning.

Types of Features. Our approach currently only supports discrete features in the feature models, and thus only discrete adaptation actions. Capturing feature cardinalities or allowing numeric feature values is currently not possible, and thus continuous adaptation actions cannot be captured.

Adaptation Constraints. When realizing the exploration strategies (both ε-greedy and FM-guided), we assumed we can always switch from a configuration to any other possible configuration. We were not concerned with the technicalities of how to reconfigure the running system (which, for example, is addressed in [Chen 2014]). We also did not consider constraints concerning the order of adaptations. In practice, only certain paths may be permissible to reach a configuration from the current one. To consider such paths, our strategies may be enhanced by building on work such as [Ramirez 2010].

3.5 Summary

In this chapter, we described exploration strategies for online reinforcement learning that use feature models to give structure to the system's adaptation space and thereby leverage additional information to guide exploration. Our strategies traverse the system's feature model to select the next adaptation action to be explored and, by leveraging the structure of the feature model, guide the exploration process. In addition, these strategies detect added and removed adaptation actions by analyzing the differences between the feature models of the system before and after an evolution step. Adaptation actions removed as a result of evolution are no longer explored, while added adaptation actions are explored first. We thus provided a conceptual framework for integrating reinforcement learning into the MAPE-K reference model of self-adaptive systems. In particular, we integrated our strategies into both the Q-Learning and SARSA algorithms. We validated our approach with an adaptive cloud service and a reconfigurable database system, two systems differing in terms of their adaptation space, the structure of their feature model, and their quality characteristics (response time instead of energy and virtual machine migrations).

Material related to this chapter

- **Realizing Self-Adaptive Systems via Online Reinforcement Learning and Feature-Model-guided Exploration**. Andreas Metzger, Clément Quinton, Zoltán Mann, Luciano Baresi, Klaus Pohl. Computing, 2022.
- **Feature Model-Guided Online Reinforcement Learning for Self-Adaptive Services**. Andreas Metzger, Clément Quinton, Zoltan Adam-Mann, Luciano Baresi, Klaus Pohl. International Conference on Service Oriented Computing, ICSOC 2020. *Core rank: A.*
  Best Paper Award.
- **Learning and Evolution in Dynamic Software Product Lines**. Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, Klaus Pohl. In 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2016. *Core rank: A.*
- **Realizing Self-Adaptive Systems via Online Reinforcement Learning and Feature-Model-guided Exploration**. Source code, reusable artifacts, datasets and experimental results available at https://gitlab.com/cquinton/fmlearning.

[8] The feature degree for a given feature f is the number of configurations that contain f.

Chapter 4

Optimization

Contents

We have conducted research on evolution and adaptation of configurable systems, which encompasses changes driven by new capabilities or by changing contexts, respectively. Another compelling factor motivating change is performance optimization. As the number of features grows, the number of configurations (*i.e.*, the configuration space) also grows exponentially. Consequently, there is a non-negligible probability that somewhere in this configuration space lies a better-performing configuration. We thus needed a way to measure configuration performances – with a specific focus on energy consumption (Section 4.2), in order to then suggest better configurations to the developer (Section 4.3).

4.1 Background and Motivation

Measuring Energy Consumption of Software. While the energy consumption of hardware components has been widely studied, software consumption only recently gained interest. By driving and managing such hardware, software is now considered as a central concern when aiming at reducing energy consumption [Noureddine 2015]. When dealing with green concerns of software systems, the impact of such systems is often measured as *power* or *energy* consumption. While power (P) measures the instantaneous consumption in Watts, energy (E) reports on a accumulated consumption over a given period in Joules [Pang 2016]. In this chapter, we will present an approach to estimate and reduce energy consumption of SPL, expressed in Joules. Thus, all our measurements represent the total energy consumed by products from this SPL, independently of their execution time, which can vary depending on products.

Energy measurement tools usually estimate the energy consumption of the CPU rather than the one of a specific software [Noureddine 2013]. Thus, identifying the share of the software under study among the total energy consumption is not straightforward. To address this issue, we first sample the CPU consumption for one second before the program starts. We define this measurement as the *idle* energy consumption P_{idle}, which refers to the average power consumption at rest. The general idea is to measure the energy consumed by the running software, E_{raw}, and then substract $E_{idle} = P_{idle} \times T_{measure}$ from E_{raw} to get rid of the environment consumption.

$$E_{net} = E_{raw} - (P_{idle} \times T_{measure}) \tag{4.1}$$

The resulting energy consumption, E_{net}, can then be associated to the software under study, as depicted in Figure 4.1 and presented in Equation (4.1).

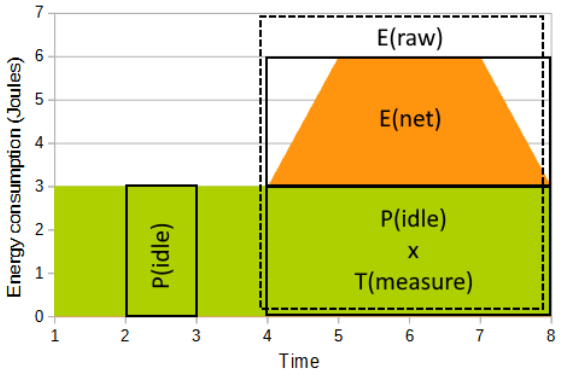


Figure 4.1: Raw *vs.* net energy consumption.

Open issues. Figure 4.2 presents an excerpt of the GPL-FH feature model that serves as use case throughout Section 4.3. GPL-FH is a testbed, used in particular to evaluate different implementations and algorithms that can be executed on a graph. The graph under test is generated at runtime through the `TestProg` feature. GPL-FH exhibits 156 configurations for 37 features and 14 constraints. These features represent different characteristics of the generated graph, such as `Weighted` or `Unweighted`, and cross-tree constraints define what algorithm can be run depending on the implementation of the graph, *e.g.*, `MSTKruskal` can only be run with a `WithEdges` implementation.

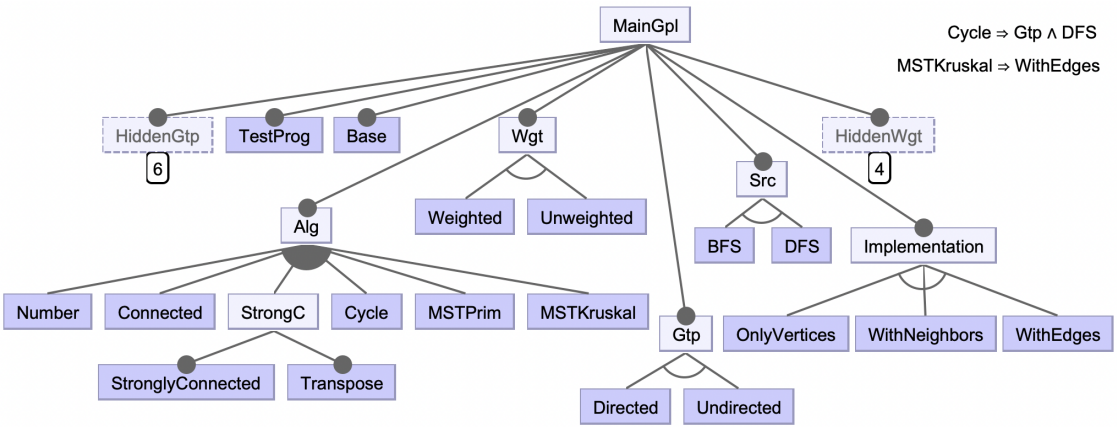


Figure 4.2: Excerpt of the feature model of GPL-FH-Java.

When running a configuration, questions arise regarding its performance, such as: *Are there better (*e.g.*, faster or less consuming) configurations? If yes, is there one that is close enough to the running one so it still complies with the user's requirements? What would be the gain of running this configuration? How to make sure changing feature(s) will not result in a worst configuration?* In particular, those questions are due to the fact that the large number of configurations makes picking the *best* configuration on the first try almost impossible, unless having the proper

background knowledge of the configuration space. Developers usually do not have this background knowledge and only consider less than 20% of the available configurations [Xu 2015]. Another reason is the use of the default configuration or a legacy one, *e.g.*, to make sure functional requirements are met. Running such a configuration does not guarantee running the optimal one; On the contrary, it may result in running worst or incorrect configurations [Pereira 2021, Nair 2020].

In both cases, it is necessary to explore the configuration space to seek configurations providing better performance. Yet, the size of the configuration space increases exponentially with the number of functionalities, making this exploration impractical manually. There is thus a need for an approach that optimizes the performance of an existing configuration while minimizing the impact on functional requirements for such a configuration.

4.2 Reducing the Energy Consumption of a SPL

Unlike measuring the energy consumption of a software, measuring the energy consumption of a SPL is a non-trivial task, as multiple related software—*i.e.*, the products of the SPL—must be measured. These products exhibit different properties, including energy consumption, while sharing several features that perform differently in different contexts. The context of a feature can either be external to the product containing this feature—*i.e.*, the environment hosting the product—or internal to the product. That is, a feature can exhibit different performances when combined with different sets of features. Inferring the energy consumption of a single feature by measuring it while running in *one* given product is therefore irrelevant and does not reflect the energy consumption of that feature in the SPL. On the other hand, measuring the energy consumption of a feature in each product individually is not feasible as some products may be complex to measure, while measuring the consumption of each product from a large SPL is not an option. To tackle these issues, we thus propose two approaches that estimate the energy consumption of features by measuring products *sampled* from the configuration space of the SPL, and then exploiting such sampled measures to reduce the energy consumption of any product from the SPL.

Both approaches improve energy consumption of products by removing features or substituting them with other ones. However, some features are included in a product to ensure its validity with regard to the feature model, *e.g.*, in or and xor relationships. We thus define $F_f \subset F$ as the set of features that are valid substitute features for a given feature f. These substitute features are either sibling features of f in or and xor relationships, or features involved in or and xor used in cross-tree constraints. On the other hand, some products may contain features due to functional constraints (*e.g.*, stakeholder's requirements). Such features cannot be removed or substituted and are hereafter referred to as *required features*. Thus, from the stakeholder standpoint, all products are functionally equivalent if they contains the features required by this stakeholder.

4.2.1 Feature-wise Energy Analysis

Energy impact of individual features. To estimate the energy consumption of each feature from the SPL, we first measure the energy consumption of every product from the sample, using the method presented in Section 4.1. The energy consumption of each product is then reported in a matrix $n \times m$ with n the features and m the products, by copying the energy consumption of the product in the columns of each included feature. For instance, Matrix (4.2) defines f_1 to f_n as the available features, p_1 to p_m the sampled products, and E_{xy} represents the energy consumption of p_x if it includes f_y, or is left empty otherwise.

By computing the median value of each column of the matrix, the relative energy consumption $\tilde{E}(f)$ of the feature f represented by this column can be estimated. The expected behavior is that extreme energy consumption will cancel out and all features will have similar median energy consumption. However, if the median consumption of a feature is higher or lower than the other medians, then the presence of this feature tends to impact the performance of the products that contain it. Although such a measure does not provide a very accurate reading, it

can nevertheless be used to compare the energy consumption of different features and perform preliminary optimizations, *e.g.*, by selecting the less consuming feature among the substitutes of each feature F_f. In the remainder of the chapter, we will refer to this method as the *feature-wise analysis*.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_m \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} f_1 & f_2 & \cdots & f_n \end{array} \\ \begin{bmatrix} E_{11} & E_{12} & \dots & E_{1n} \\ E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{m1} & E_{m2} & \dots & E_{mn} \end{bmatrix} \\ \begin{array}{cccc} \tilde{E}(f_1) & \tilde{E}(f_2) & \cdots & \tilde{E}(f_n) \end{array} \end{array} \tag{4.2}$$

Feature-wise mitigation. Getting the most energy-efficient product including the required features follows a two-steps process. First, all optional features of the products are removed, thus only including the required features and the features requiring a substitution. Then, by leveraging the feature-wise analysis, the energy consumption of the remaining non-required features can be compared with their respective substitutes, to identify the one with the lowest consumption among them. Each of the non-required features are replaced by the most energy efficient substitute. This approach always converges toward an efficient product composed of no optional feature, including only the features with the lowest energy consumption within each feature substitution set. The product resulting from this mitigation strategy is the one with the lowest energy consumption that can be obtained given an initial configuration.

4.2.2 Pairwise Energy Analysis

Energy impact of pairwise interactions. Although measuring the energy consumption of each feature in isolation gives a general trend, it cannot be used to compute the energy consumption of a combination of features (*e.g.*, as the mean or the median of several individual consumption) due to the feature interactions phenomenon [Siegmund 2012]. Indeed, numerous work have shown that features interact with each other, hence impacting performances of products [Apel 2010, Apel 2011, Batory 2011, Siegmund 2012, Apel 2013, Siegmund 2015]. Therefore, restricting the energy consumption analysis to individual features does not provide a comprehensive landscape of a feature consumption, and additional analysis that consider feature interactions must be performed to obtain additional details about the energy consumption. By analyzing how the consumption of a feature evolves when this feature is combined with different features, it is thus possible to highlight feature interactions leading to positive or negative impact on the consumption of the product.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_m \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} c_1 & c_2 & \cdots & c_n \end{array} \\ \begin{bmatrix} E_{11} & E_{12} & \dots & E_{1n} \\ E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{m1} & E_{m2} & \dots & E_{mn} \end{bmatrix} \\ \begin{array}{cccc} \tilde{E}(c_1) & \tilde{E}(c_2) & \cdots & \tilde{E}(c_n) \end{array} \end{array} \tag{4.3}$$

A possible means to take pairwise feature interactions into account is by creating a new matrix with as many columns as there are valid pairs of features in the SPL. The consumption of each pair of features can be quantified by reporting on the energy consumption of each product in the columns of the pairs of features that this product contains, as illustrated in Matrix (4.3). c_1 to c_n are all the valid pairs of features, p_1 to p_m the sampled products, and E_{xy} is the energy

consumption of p_x if it contains c_y, or is left empty otherwise. To ensure a proper interaction coverage—*i.e.*, that all valid pairs of features are measured—the sampling of products must be performed by an algorithm ensuring such coverage.

Following the same methodology as in the *feature-wise analysis*, the consumption of pairwise feature interactions can be inferred by computing the median energy consumption $\tilde{E}(c)$ of each pair of features c. In the remainder of the chapter, we will refer to this method as the *pairwise analysis*. It is worth noting that this method is not only valid for pairwise interactions, but can also be used to deal with larger T-wise interactions of features.

Pairwise mitigation. Instead of replacing each feature by the substitute feature with the lowest energy consumption, this second approach iteratively picks the alternative features whose interactions with other features of the product results in a more energy-efficient product. At each iteration, the approach identifies a feature to remove from the product and, if required, replaces this feature.

To identify the feature f to be removed from a given product P, our approach relies on a *scoring* system: the interaction score $\mathcal{I}$. The interaction score of a feature f is computed by considering all pairs of feature containing f in the product P, and by summing the observed median energy consumption $\tilde{E}(c)$ of these pairs, as described in Formula (4.4).[1]

$$\mathcal{I}(f, P) = \sum_{g \in P \mid\ g \neq f} \tilde{E}(gf) \tag{4.4}$$

The iterations of this approach are realized as described by Algorithm 4. This algorithm iterates over the set of features until no more improvement can be performed. That is, each iteration removes or changes one feature in the product. At each iteration, the feature with the highest interaction score in the product must be removed in priority. The algorithm starts by sorting features by decreasing interaction score (line 9) and considers the first feature of the list (line 10) as a removal candidate. If this removal feature is a user-required feature and cannot be removed, it is skipped (line 14). If the removal candidate is not a required feature and must be replaced (line 18), the replacement feature is identified among all possible substitutes—*i.e.*, F_f—by computing the interaction score of alternative features with regards to all remaining features of the product—*i.e.*, all but the removal candidate (lines 19 to 24). The selected replacement feature is the one with the lowest interaction score among all alternative features. As a result of the iteration, a new product is created by including the replacement feature (line 27).

However, if the removal candidate has the lowest interaction score, the replacement is discarded and the algorithm skips the feature, which is kept in the product. If a feature is skipped—*i.e.*, it was either a requirement or already the best option, a new removal candidate is defined as the next feature in the ordered list (line 35 and 12). Other features of the product will be changed over the next iterations to accommodate this skipped feature. Once a modification has been applied, the algorithm proceeds to the next iteration, unless a stop criteria is met: if a same product appears twice over different iterations, or if all features were tested during an iteration and no optimization was found (line 37).

Once a stop criteria is met, the energy consumption of the product resulting from each iteration is measured in order to monitor the energy gain. As the different mutations of the product are based on empirical data, which may be subject to imprecision and noise, it is possible that a specific iteration worsens the performance of the product. For this reason, the last step of this algorithm measures the energy consumption of the products resulting from each iteration. The product finally returned by this algorithm is the one with the lowest energy consumption, which may be the initial product in the worst case scenario (lines 22).

[1] The interaction score can also be used during the configuration process, *e.g.*, to assist the user when selecting the most energy efficient features when dealing with a partial configuration.

Algorithm 4 Interaction mitigation

```
1: Input initialProduct : a product to improve
2: Output bestProduct : the best product from all iterations
3:
4: iterations.addProd(initialProduct)
5: improvable ← true
6: while improvable do
7:     currentProd ← iterations.lastItem()
8:     currentProd.removeNonRequiredOptionalFeatures()
9:     sortedFeat ← sortByInteractionScore(currentProd)
10:    remCandIndex ← 0
11:    noChangeFound ← true
12:    while (remCandIndex < currentProd.size) ∧ noChangeFound do
13:        remCandidate ← sortedFeat.get(remCandIndex)
14:        if ¬isRequirement(remCandidate) then
15:            prodCandidate ← copy(currentProd)
16:            prodCandidate.remove(remCandidate)
17:            subOptions ← allF_sub(remCandidate)
18:            if subOptions then
19:                currentBest ← remCandidate
20:                for subCandidate ∈ subOptions do
21:                    if ℐ(subCandidate, prodCandidate) < ℐ(currentBest, prodCandidate) then
22:                        currentBest ← subCandidate
23:                    end if
24:                end for
25:                if currentBest ≠ remCandidate then
26:                    prodCandidate.addFeat(currentBest)
27:                    iterations.addProd(prodCandidate)
28:                    noChangeFound ← false
29:                end if
30:            else
31:                iterations.addProd(prodCandidate)
32:                noChangeFound ← false
33:            end if
34:        end if
35:        remCandIndex++
36:    end while
37:    improvable ← (remCandIndex < currentProd.size) ∧ allDifferent(iterations)
38: end whilereturn lowestEc(iterations)
```

4.2.3 Empirical Validation

In the previous section, we introduced two approaches to reduce the energy consumption of a given product. In this section, we experimentally assess each of these approaches. In particular, we aim to answer the following research questions:

RQ 1: *Do our different analysis detect feature interactions impacting energy consumption?* By applying the two approaches on the same set of products, it should be possible to determine whether feature interactions have been detected as the two analysis methods should provide different results.

RQ 2: *How effective are our approaches to reduce the energy consumption of a product?* The two proposed approaches rely on different analysis methods to mitigate energy consumption of products. We propose two experiments to ensure both of them improve the consumption of the products given a set of required features and evaluate how they differ.

To assess the effectiveness of our solution when measuring energy consumption of a software product line, we performed our experiments on RobocodeSPL, a software product line designed to yield robots for Robocode [Martinez 2018]. Robocode is an environment in which community-developed robots fight against each other in battles. A battle is composed of several rounds, and rounds have a time granularity of turns. During a turn, each robot taking part in the battle computes its next action and sends it to the Robocode engine which executes them all and proceeds to the next turn. A round ends when only one robot survives, and the winner of a match is the robot which caused the most damages to its opponents through the different rounds.

The RobocodeSPL proposes several implementations for the 5 mandatory features a robot requires to run properly—*i.e.*, *radar*, *targeting*, *movement*, *enemy selection* and *gun*. For instance, a *movement* can follow linear or circular patterns, follow the walls, or ram the opponent, among others. There are also 3 optional features related to resource management (*e.g.*, not spending more in-game energy than the robots have), for a total of 92 features and 72 leaf features. The number of valid products is 1.3×10^6. Figure 4.3 depicts an excerpt of the feature model of RobocodeSPL.

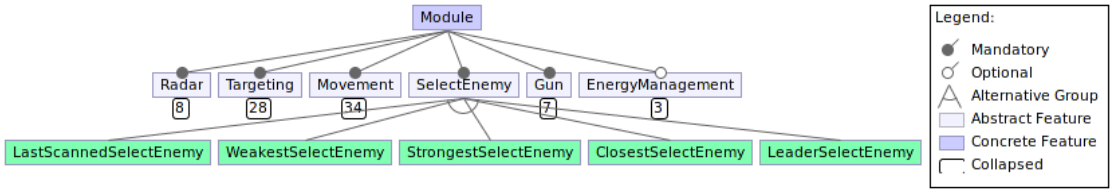
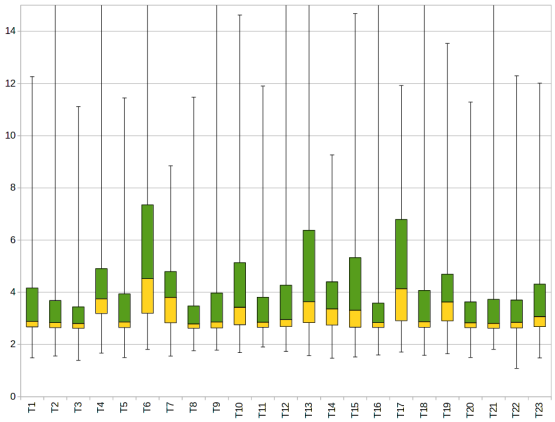


Figure 4.3: Excerpt of the Feature Model of RobocodeSPL.

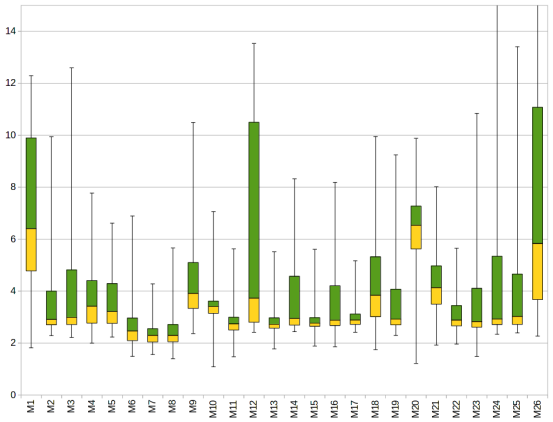
To evaluate our approach, we launched multiple robot matches and ran our mitigation techniques to minimize the energy consumption of such matches. In particular, we launched matches opposing a sampled robot and a reference robot, the `sample.Wall` robot, considered as the strongest robot provided by Robocode[2]. As the goal was to minimize the energy consumption, we were not interested in which robot wins or loses the match, but in the overall energy consumption of such a match. In order to fill the pairwise analysis matrix, the sample must contain several occurrences of each valid pair of features from the feature model. To ensure such a coverage, we relied on the T-wise algorithm ICPL [Johansen 2012] with $T = 2$ to sample the configuration space of RobocodeSPL. Another sampling technique may provide better uniform random samples [Kaltenecker 2020, Munoz 2019], but such techniques do not meet our coverage requirements. This algorithm generated 602 robots, hereafter referred to as the *training sample*. For each couple *(sampled robot, reference robot)*, we ran 10 matches to consolidate the performance data, resulting in a total of 6,020 matches of 1 round.

We used JJoules,[3] a Java tool using the RAPL device of Intel CPU, to measure the energy consumption of the matches. JJoules is also able to monitor the energy consumption of the DRAM, while other tools can monitor other components, such as Hard Disk Drives. As Robocode is mainly CPU-intensive, we decided to focus on the energy consumption of the CPU. The energy consumption was monitored from the start to the end of each match, thus including the energy consumed by both robots, but excluding the energy consumption of the startup and shutdown of Robocode. All measurements were obtained from a machine running the Manjaro Linux distribution with an Intel i5 CPU at 2.9GHz and 8GB of RAM. Results, input data and instructions to reproduce these experiments are available online.[4]

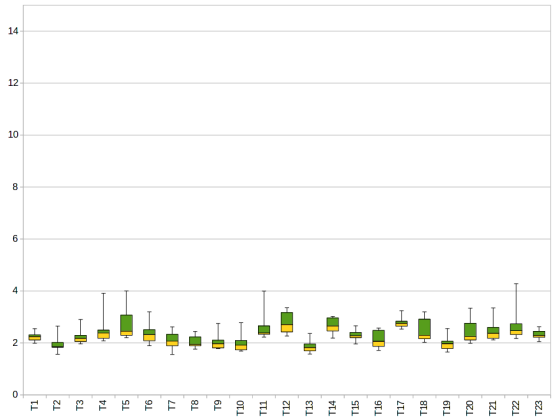
[2]According to the Robocode Wiki: `https://robowiki.net/`
[3]`https://github.com/powerapi-ng/j-joules`
[4]`https://doi.org/10.5281/zenodo.5048316`



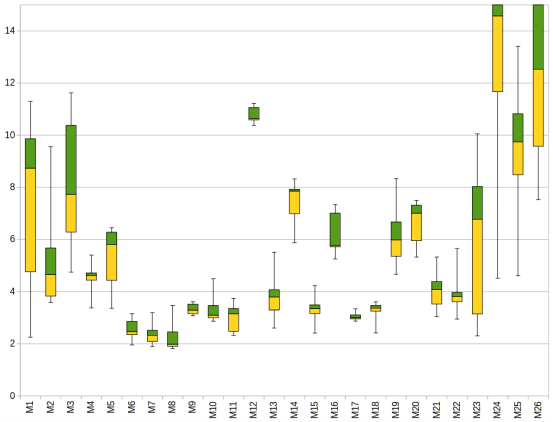
(a) Energy consumption per targeting feature.



(b) Energy consumption per movement feature.



(c) Energy consumption per targeting features when feature LinearRammingMovement (M7) is selected.



(d) Energy consumption per movement features when feature NoTargeting (T6) is selected.

Figure 4.4: Energy variations between the movement and targeting features and their potential interactions.

4.2.4 Results

Detecting interacting features. The pairwise analysis relies on feature interactions to mitigate energy consumption of products. The goal of this first experiment is therefore to ensure that the pairwise analysis is able to detect at least one occurrence of feature interaction. The first experiment thus compares the energy consumption of the Movement and Targeting features in different contexts—*i.e.*, in the presence of different sets of other features. Figure 4.4 depicts how the energy consumption of different features evolves depending on the analysis method.[5] Figures 4.4a and 4.4b report on the energy consumption (measured with the feature-wise analysis) of the products from our *training sample* containing respectively each targeting and movement feature. Among the targeting features, T4, T6 and T17 induce an higher energy consumption than the others, but most features show similar energy consumption, around 3 Joules. Among the movement features, M1, M20 and M26 impose the highest energy consumption, around 6 Joules, while M6, M7 and M8 report on the lowest one, slightly above 2 Joules.

These differences in energy consumption can partially be explained by the functional behavior of these features. For instance, T6 (NoTargeting) performs no particular operation and always makes the robot shoot forward—*i.e.*, in the direction it is aiming at. This is not a smart behavior and the energy consumed by matches involving this feature depends on how fast the opponent

[5]Mapping to real feature names available in the open data

is able to destroy this robot. By contrast, T13 (TargetAdvancingVelocitySegmentation) tries to anticipate the position of the opponent based on its speed and direction to ensure that the bullet and the opponent collide. Thus, a robot configured with T13 is able to win quickly, reducing the energy consumption despite the additional computations required to anticipate the position of the opponent.

Figure 4.4c presents the energy consumption of each targeting feature when the feature LinearRammingMovement is selected—*i.e.*, M7, the most energy efficient movement feature. This figure is obtained by selecting all measurements of M7 in Figure 4.4b, and breaking them down per targeting feature. M7 being the best movement feature, the consumption of products containing each targeting feature is either improved or unchanged when M7 is selected. However, when targeting features are sorted by median energy consumption, their rank change depending on the context. For instance, T21 is ranked 3rd by the feature-wise analysis, but becomes 16th when M7 is selected. T13 is ranked 19th out of 23 by the feature-wise analysis, but 1st in the pairwise analysis when M7 is selected. Furthermore, the median energy consumption of the couple of M7 and T13 is 1.8J, which is lower than the medians of both of these features alone, respectively 2.2 Joules and 3.6 Joules. Therefore, despite M7 being the best movement feature, its performance can still be improved by selecting a relevant targeting feature.

As shown by the feature-wise analysis in Figure 4.4b, products including M9 and M12 have similar median energy consumption—*i.e.*, respectively 3.9 Joules and 3.7 Joules. However, when paired with NoTargeting (T6), one of the worst targeting features, their consumption evolve differently, as depicted in Figure 4.4d. The energy consumption of products including M9 is reduced from 3.9 Joules to 3.3 Joules, while the one for products including M12 dramatically increases from 3.7 Joules to 10.6 Joules. Thus, despite being considered a sub-optimal choice by the feature-wise analysis, T6 becomes an efficient choice when paired with M9. The pair composed of T6 and M8 is another occurrence of pairwise interaction outperforming both of its members (2 Joules instead of 4.5 Joules and 2.2 Joules, respectively). This result can be explained by the behavior of the features: M8 is a *ramming* movement feature, meaning that it is always moving toward the opponent. In this context, the behavior of NoTargeting—*i.e.*, always shooting forward—is very efficient, as it always hits the opponent.

Such changes in the resulting energy consumption with couple of features outperforming both of their members alone show that the energy consumption of targeting and movement features changes depending on how they are paired. Therefore, it highlights feature interactions between the targeting and movement features in RobocodeSPL. This experiment thus unveiled occurrences of feature interactions allowing us to answer **RQ 1** positively: the pairwise analysis is able to detect interactions significantly impacting the energy consumption of products, and such interactions were not detected by the feature-wise analysis.

Behavior without required feature. The purpose of the second experiment is *(i)* to ensure the two mitigation approaches lead to a product different from the initial one, and *(ii)* to evaluate the energy consumption improvement resulting from these approaches. The first experiment showed that the feature-wise and pairwise analysis provide different results, due to their different granularity levels. It is yet to determine if the products resulting from their respective mitigation exhibit different energy consumption.

As explained in Section 4.2.2, the feature-wise analysis converges toward a specific product composed of no optional feature, and the features with the lowest energy consumption in each substitution set. In RobocodeSPL, considering our optimization goal, *i.e.*, reducing the energy consumption against `sample.Wall`, and without any required feature, this product is composed of the features TurnMultiplierLock, DistanceSegmentation, LinearRammingMovement, StrongestSelectEnemy, and NoFireGun. Whatever the initial product considered for improvement, the feature-wise analysis will always return this product, hereafter referred to as the *Best Theoretic* product, BT_0.

By applying the pairwise analysis on this product, we can determine how the pairwise analysis compares to the feature-wise analysis in the absence of required features. The result of this

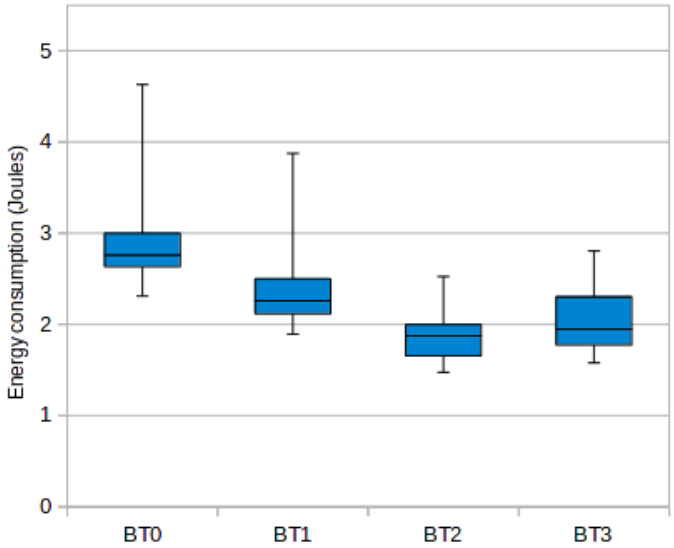


Figure 4.5: Improving the product resulting from the feature-wise analysis with the pairwise one.

experiment is depicted in Figure 4.5, where BT_0 consumes 2.8 Joules. The pairwise analysis performed three iterations before reaching a stop criteria and returning the best product of the different iterations (BT_2), whose energy consumption is 1.9 Joules, *i.e.*, 29% lower than BT_0.

This experiment provides a partial answer to our second research question **RQ 2**: the pairwise analysis outperforms the best product of the feature-wise analysis by 30%, when there is no required feature.

Behavior with required features. To complete this partial answer, the third experiment is a variant of the previous experiment that takes required features into account. The purpose of this experiment is *(i)* to ensure the changes our approaches perform on a product containing required features effectively reduce the energy consumption of such a product, and *(ii)* to evaluate these reductions. The products resulting from both approaches depend on the initial product, and on which features are required in this product. Therefore, by contrast to the previous experiment, it is not possible to assess our approaches with only one initial product. Thus, we used the FeatureIDE Product Generator to produce a sample of 520 random products—*i.e.*, 1 product tested for 2,500 products of the SPL—hereafter referred to as the *validation sample*. To mimic a real use case, we defined a random feature (based on the `java.util.Random` class) as a requirement in each of these products.

Relying on the consumption data measured on the *training sample* presented in Section 4.2.3, we applied our two energy mitigation approaches on each product from the *validation sample*. We evaluated how the products resulting from both approaches perform compared to their respective initial product. The feature-wise analysis generated 520 products, and the pairwise analysis generated 2,687 products—*i.e.*, a mean of 5 iterations per initial product. By design, the result of the first iteration of the pairwise analysis is the initial product, thus all initial products are included in these 3,207 products. We computed the performance of a product as its median energy consumption over 10 matches, for a total of 32,070 matches. Figure 4.6 presents the energy consumption of the products resulting from each analysis (on the vertical axis) depending on the energy consumption of the initial product (on the horizontal axis). Products that are on the improvement threshold line ($x = y$, identity line) performed the same as the initial product, meaning that the corresponding approach failed to reduce its consumption and returned the initial product. Products that are strictly below the improvement threshold line performed better than their respective initial products.

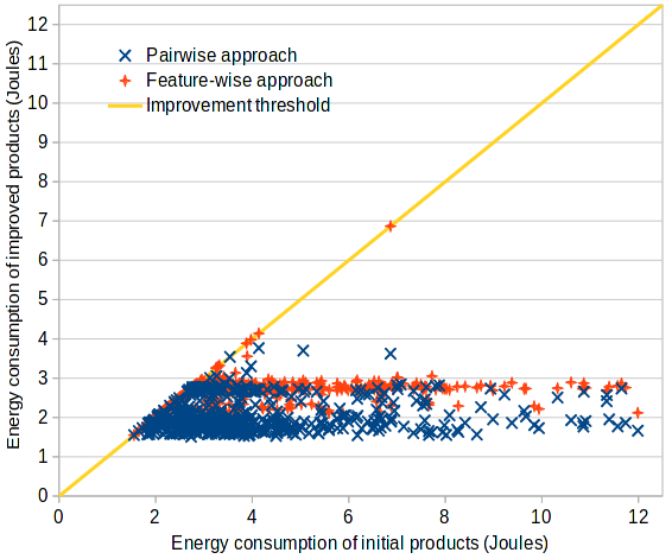


Figure 4.6: Energy consumption of the products resulting from both analysis.

The feature-wise analysis improved the performance of 375 products from the *validation sample* (72%), while the pairwise analysis improved the performance of 501 products (96%). For 127 products (24%), the pairwise analysis found improvement when the feature-wise analysis failed. For 1 product (0.2%), the feature-wise analysis found improvement while the pairwise did not. Additional analysis on this specific product tend to exclude noise or measurement error as a cause for this exception.

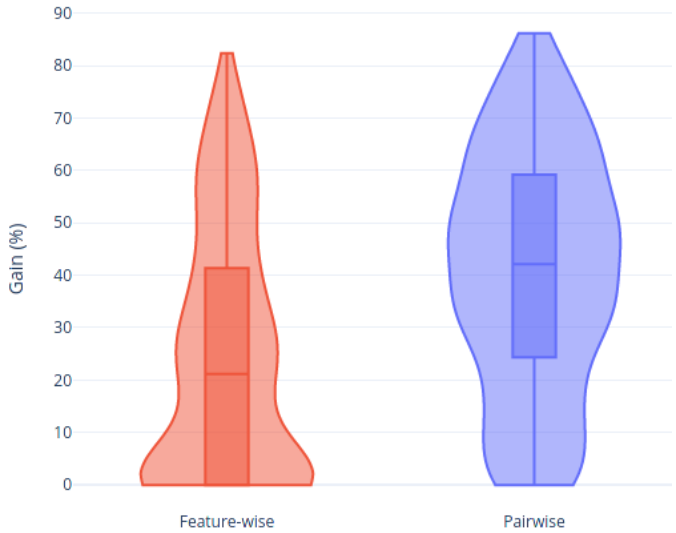
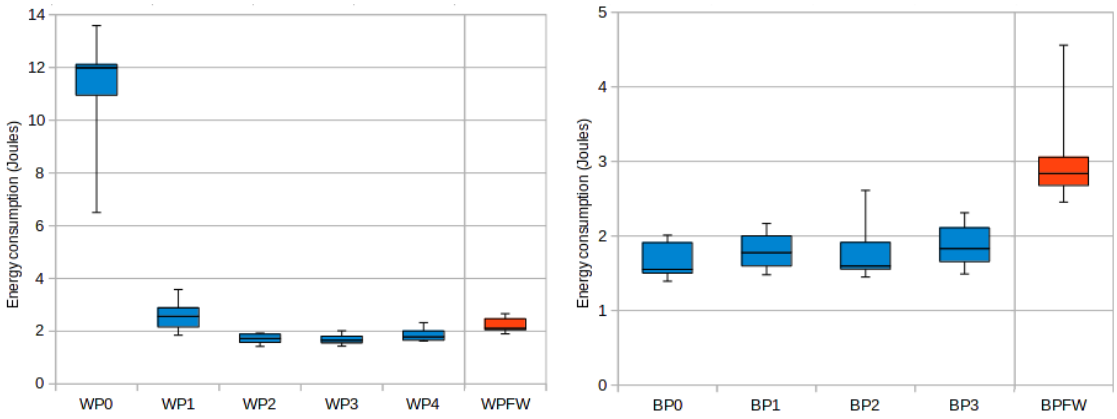


Figure 4.7: Relative gains of the pairwise and feature-wise analysis.

To get a better view on the efficiency differences between the two approaches, Figure 4.7 depicts their respective relative gains—*i.e.*, by how much they reduced the energy consumption of the initial products. In the feature-wise analysis, the end of the first quartile is still at 0%, as it improved 72% of the products, whereas with the pairwise analysis the end of the first quartile is already near a 24% gain. The median gain of the feature-wise analysis is 20%. Regarding the pairwise analysis, such a gain is reached before the second quartile. Therefore, only half of

the products resulting from the feature-wise analysis obtained gains higher than 20%, while the pairwise analysis improved more than three quarters of products by such a gain. Similarly, half of products improved by the pairwise analysis were improved by 40% or more, while the feature-wise analysis had such a gain for only a quarter of products. The maximum gain is similar for both approaches: 82% and 86%, respectively.



(a) Worst initial product.

(b) Best initial product.

Figure 4.8: Focus on the best and worst initial products from the *validation sample*.

Figure 4.8 presents how both approaches performed on two products: those with the worst and best initial energy consumption in the *validation sample*. The initial product is designated with the subscript 0 (*e.g.*, WP_0 in Figure 4.8a). The different iterations of the pairwise analysis on the two products are designated with their respective index (WP_1 to WP_4 and BP_1 to BP_3), while the result of the feature-wise analysis used as comparison is designated with the subscript FW ($WPFW$ and $BPFW$, respectively). Figure 4.8a depicts the product WP_0 with the worst initial energy consumption, 12 Joules. The pairwise analysis performed 4 iterations before meeting a stop criteria. Most of the gains are obtained after the first iteration, with WP_1 reducing the energy consumption by 78%. WP_2 and WP_3 brought additional gains of 32% and 3% on their preceding iteration, respectively. However, WP_4 increased the energy consumption by 8%, resulting in WP_3 being returned by the pairwise analysis, with an energy consumption 86% lower than WP_0. The feature-wise analysis returned a product $WPFW$ with an energy consumption 82% lower than for WP_0. The energy consumption of the product WP_3 resulting from the pairwise analysis is 21% lower than the product $WPFW$ resulting from the feature-wise analysis. Figure 4.8b depicts the product BP_0 with the best initial energy consumption, 1.6 Joules. This product is more challenging for both of our approaches, as none of them found any optimization. The pairwise analysis performed 3 iterations with energy consumption 14%, 2% and 18% higher than BP_0, respectively. The energy consumption of the product $BPFW$ resulting from the feature-wise analysis is 83% higher than BP_0. As both approach fail to find optimization, they return the initial product BP_0.

These results complete the partial answer to our second research question **RQ 2**: Both approaches are able to improve products, with and without required features, and the pairwise analysis outperforms the features-wise approach. Overall, both of our approaches succeed in improving products from RoboCodeSPL, with or without constraints. The feature-wise and pairwise analysis thus provided useful input data about energy consumption of features and couple of features, that could then be used to improve products through the feature-wise and pairwise mitigation processes. The pairwise analysis improved more products than the feature-wise analysis, and led to higher gains. However, although less efficient than the pairwise analysis, the feature-wise analysis is more straightforward to setup, and can be used as a first intent to reduce energy consumption. It is especially relevant in the absence of feature interactions, or in systems where pairs of features are too numerous to be exhaustively measured.

4.2.5 Discussion

Validity Threats. To assess our approach, we ran our experiments on a specific SPL (RobocodeSPL) to measure and reduce the energy consumption of real-world products derived from this SPL. Results, such as the success rate or the relative gains, are thus only related to this single system, and cannot be generalized. Nonetheless, our contribution can be easily applied to any SPL. The improvements resulting from applying our approaches to other SPL will depend on the initial energy consumption of products and the impact of feature interactions on these products. A second threat to validity lies in the *training sample* considered. To avoid measuring all products of the SPL, we sampled the configuration space and measured 602 products, which is only 0.05% of all valid products. Such a small sample may prevent the detection of some feature interactions and therefore, energy optimization hotspots. Still, it is worth noting that despite the low number of analyzed products, significant gains were obtained on the vast majority of products using our approaches.

Limitations. During the pairwise mitigation process, products are changed over several iterations. However, it might be possible that the optimal change in a given iteration prevents further improvements in the next iterations, *e.g.*, a sub-optimal change in that iteration might lead to further and greater improvements in the long term. Furthermore, this algorithm removes all non-required optional features, without taking into account their hypothetical positive interactions in the product. It does not either consider the possibility to add an optional feature to improve the energy consumption of the product.

Extensions to feature models have been developed to convey information about features, *e.g.*, attributes. Extended feature models could thus be used to assign consumption data on features, in order to automatically apply optimizations. However, the adoption of such extensions may raise some challenges when dealing with consumption metrics associated to pairs of features.

In the green computing domain, a commonly-used means to reduce energy consumption of software is by refactoring inefficient code—*i.e.*, making it more efficient without changing its functional behavior. Although our analysis methods highlight features or pairs of features with high energy consumption, they do not provide fine grained feedback nor means to identify what causes such non energy-inefficient products at low-level, *e.g.*, inefficient code or unexpected behavior.

The energy consumed to obtain the measurements for our experiments (*i.e.*, the *training sample* of 602 products) amounts to $24,328$ Joules. In comparison, the highest energy saving among the 520 products of the validation sample is 10 Joules per match. Hence, we can consider that our approach is profitable after $2,433$ matches in the best case scenario where energy savings are high. This might seem a significant number at first, but this result must be considered keeping in mind the 1.3×10^6 products of the SPL that can benefit from these measurements. In addition, it cannot be generalized to others SPL since this profitability threshold tightly depends on the number of features and products of the analyzed SPL.

Finally, the pairwise analysis method relies on a sample of products containing all pairs of features. As the number of features in the SPL grows, the number of pairs had a quadratic growth. For larger feature models, the use of heuristics to identify interactions between pairs of feature may proves necessary.

4.3 Configuration Optimization with Limited Functional Impact

Through the research presented in the previous section, we have established the groundwork for measuring performance related to (pair of) features. Subsequently, we explored ways to leverage these measurements to suggest better-performing configurations. The Iterative Configuration Optimization (ICO) approach discussed in this section thus builds upon the energy consumption reduction method presented in the previous section. In particular, ICO addresses its limitations, discussed above. We thus extended the method explained in Section 4.2 with a support for multi-objective optimization while handling cross-tree constraints in its optimization process. The core idea is as follows: From an initial configuration, ICO explores the remaining configuration space in search of configurations that *(i)* are neighbors of the initial configuration, *(ii)* comply with the user's functional requirements (*i.e.*, features that have to be selected or excluded) and *(iii)* optimize given performance indicators. It then provides optimization suggestions to the developer.

4.3.1 Optimizing Configurations

To perform the optimization process, ICO relies on the performance of each feature regarding all the considered metrics. That is, as shown by Equation 4.5, the overall performance P of a feature f with respect to n metrics is the sum, for each metric, of p_{if} the normalized performance of the feature regarding this metric, multiplied by w_i the weight associated to this metric and by d_i the objective optimization for this metric, *i.e.*, 1 or -1, respectively to maximize or minimize.

$$P_f = \sum_{i=1}^{n} d_i w_i p_{if} \tag{4.5}$$

As interactions between features impact performance [Siegmund 2012], ICO is able to optimize configurations *w.r.t* tuples of features of any size, in which case f defines a tuple of features instead of a single one. The performance of a configuration is then computed as the average performance of features - or tuples of features in interaction-wise optimization - contained in this configuration.

The ICO approach is realized by Algorithm 5, which takes the set of features, the list of constraints and the initial configuration as input to compute a set of improvement suggestions. The algorithm starts by creating a set of candidate configurations for the configuration to optimize (lines 7– 15). Candidate configurations are the set of configurations that are one change away from the initial configuration, *i.e.*, *neighbor* configurations, since they differ by the selection/deselection of one feature. For instance, a GPL-FH configuration for a `Weighted` graph is a neighbor of the same configuration where `Unweighted` graph is selected since both features are mutually exclusive. In a general way, each unselected feature leads to a candidate configuration where this feature is selected (lines 7– 9), each selected feature leads to a candidate configuration where this feature is unselected (lines 10– 12), and each exclusive relationship of both a selected and unselected features leads to a candidate configuration where the selected feature is deselected and the unselected one is selected (lines 13– 15). Candidate configurations are then ordered by performance gain (line 16), and finally filtered regarding their validity and performance (line 18), to ensure that the returned suggestions *(i)* cannot turn a valid configuration into an invalid one and *(ii)* can only improve the performance of the configuration, according to the performance model[6].

For each candidate configuration, the algorithm then computes the difference between this candidate configuration and the initial one (line 19). This difference takes the form of a feature to add or a feature to remove – or both, and its estimated performance gain. As a result, the algorithm provides a set of improvement suggestions, ordered by potential performance gains.

[6]The computation of the performance model is out of the scope of this chapter. Yet, we discuss this particular point in Section 4.3.4.

Algorithm 5 ICO optimization algorithm

Input: features, constraints, $conf_{init}$;
Output: suggestions
$candidates \leftarrow \emptyset$
$suggestions \leftarrow \emptyset$
$addable \leftarrow (features \backslash conf_{init}) \backslash constraints_{exclude}$
$removable \leftarrow conf_{init} \backslash constraints_{include}$
for $rem \in removable$ **do**
    $candidates \leftarrow candidates \cup$ `newConfig`$(conf_{init} \backslash rem)$
end for
for $add \in addable$ **do**
    $candidates \leftarrow candidates \cup$ `newConfig`$(add \cup conf_{init})$
end for
for $add \in addable, rem \in removable$ **do**
    $candidates \leftarrow candidates \cup$ `newConfig`$(addable \cup conf_{init} \backslash removable)$
end for
$candidates \leftarrow$ `sortByPerfGain`$(candidates)$
for $c \in candidates$ **do**
    if `isValid`$(c,$ `constraints`$) \wedge$ `perf`$(c) >$ `perf`$(conf_{init})$ **then**
        $suggestions \leftarrow suggestions \cup$ `diff`$(c, conf_{init})$
    end if
end for
return suggestions

For instance, a possible suggestion for a GPL-FH configuration is to replace the `Undirected` feature by the `Directed` feature which offers better performances, while other features remain unchanged. The approach can thus be entirely automated by applying, while new suggestions are provided, the one providing the highest performance gain. ICO also offers an interactive mode, where developers select the suggestion to apply according to their functional requirements and domain knowledge.

4.3.2 Empirical Validation

The ICO approach has been implemented in the ICO tool suite. This tool suite has been built with the objectives of *(i)* providing developers with feedback about the performances of a given configuration and *(ii)* providing suggestions to optimize its performance by adding or removing a feature. ICO is composed of three software components: (1) ICOLIB, a Java implementation of the proposed approach; (2) ICOCLI, a command-line interface; and (3) ICOPLUGIN, an Eclipse plugin. The ICO tool suite takes as input a configuration, a feature model and performance files, and then returns optimized configurations based on the suggestions provided by Algorithm 5. Fig-

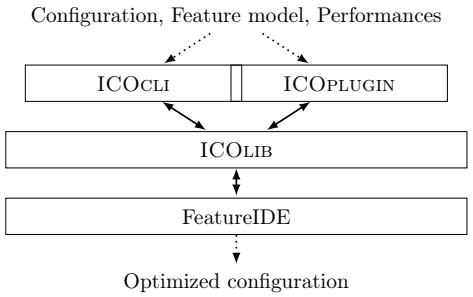


Figure 4.9: The architecture of the ICO tool suite.

ure 4.9 provides an overview of the architecture of ICO: through either ICOCLI or ICOPLUGIN, user instructions are sent to ICOLIB which then performs various operations.

The tool suite is centered on ICOLIB, a Java library that exposes the API managing all operations that can be performed with ICO: loading a project, displaying current performances, managing constraints (*i.e.*, the lists of features required or excluded by the developer), listing or applying improvement suggestions and saving the new configurations. In particular, ICOLIB delegates to the FeatureIDE [Thüm 2014] library the responsibility to load, update, validate and save the configurations. Taken as a standalone component, ICOLIB can be integrated as a Java dependency into any tool requiring an implementation of Algorithm 5. ICOPLUGIN is an Eclipse Plugin developed to interact with ICOLIB and implemented as an Eclipse view. It thus provides a GUI that assists developers when seeking optimized configurations, in particular by proposing visual feedback on suggested optimizations. ICOCLI is a command-line interface to interact with ICOLIB, enabling an in-depth exploration of the variability of the software and its performances. It can be used directly by the developer or integrated into automated processes such as CI/CD. The source code of ICO is publicly available[7], and [Guégain 2023] covers the specifics of its implementation.

Our goal is to assess the validity and effectiveness of our approach. In particular, we aim to answer the following research questions:

RQ 1: *Can any configuration be optimized?* Considering a configuration space, we investigate whether or not any configuration from that space can be optimized using our approach.

RQ 2: *How effective is the* ICO *optimization approach?* When the ICO approach provides a better configuration, we measure the performance discrepancy between that configuration and the initial one.

RQ 3: *How many iterations does it take to optimize a configuration?* We evaluate the number of iterations of ICO required to converge from an initial configuration to its respective optimal one.

We evaluated our approach on the real-world configurable system GPL-FH presented in Section 4.1. This system was selected for several reasons. First, both its source code and feature model are publicly available, and they seamlessly integrate as GPL-FH can be run from the command line. Second, its feature model (presented in Figure 4.2) exhibits 156 configurations, thus providing a large-enough configuration space for the optimization process to be significant.

The experiments consist in optimizing all 156 configurations regarding a pair of performance indicators, namely the execution time (`time`) and the number of lines of code (`LoC`). This exhaustive optimization highlights how the approach navigates through the configuration space. To not interfere with the `time` measurements, the logging functionality that comes as a default option of the GPL-FH system was disabled, as it might misrepresent the actual execution time. The GPL-FH default number of vertices was changed from 10 to 3500 to yield a larger graph and be able to properly measure the `time`, thus getting meaningful readings. The building time of the graph itself is excluded from the `time` measurement, since constant across configurations. In order to consolidate the measure of the `time` of each configuration, the experiment was repeated 20 times. Beyond that point, the average execution time converges.

The performance of each feature *w.r.t* `LoC` and `time` is computed according to the method proposed in Section 4.1, *i.e.*, the performance of a feature *w.r.t* a metric is the average performance in this metric of configurations containing this feature. The global performance of each feature (*i.e.*, the performance taking all metrics into consideration) is then calculated using Formula 4.5. Both metrics were given the same weight, while the optimization goal was set to a minimization of both performance indicators. The optimization algorithm has then been applied on each of the 156 configurations of GPL-FH: for each initial configuration, it seeks for a better neighbor configuration that minimizes `LoC` and `time`. All measurements were performed on a machine with an Intel Core i5 CPU at 2.9GHz and 8GB of RAM.

[7] `https://gitlab.inria.fr/ico`

4.3.3 Results

The configuration space of GPL-FH has been exhaustively measured, providing insight into the performance of each of the 156 configurations *w.r.t* `LoC` and `time`. Figure 4.10 presents such performances. The best and worst `time` are respectively 0.09 and 23.4 seconds, while `LoC` ranges from 282 to 632. The optimization of a configuration should thus provide higher variations in `time` than in `LoC`, as the ratio between the worst and best readings for `time` (260) is orders of magnitude higher than the one for `LoC` (2.2).

Investigating RQ1: Can any configuration be optimized? Applying the best suggestion (if any) provided by Algorithm 5 to a given configuration results in either one of the following situations: (S_1) the configuration improved regarding both performance indicators; (S_2) the configuration improved regarding one performance indicator and worsened regarding the other; (S_3) the configuration did not improve nor worsen, *i.e.*, ICO returned no suggestion; (S_4) the configuration worsened on both indicators[8].

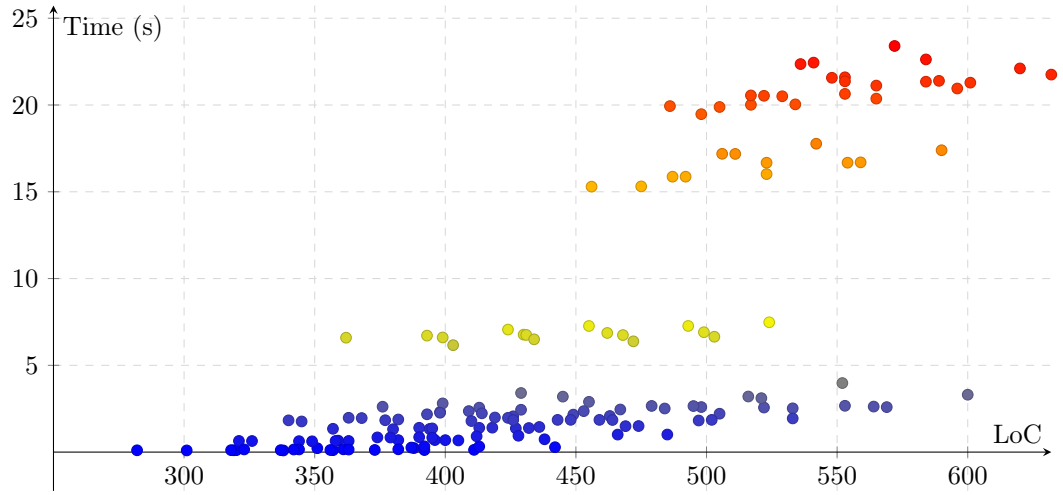


Figure 4.10: Performance of each GPL-FH configuration *w.r.t* `LoC` and `time` (lower left corner is better).

Table 4.1 summarizes the performance gains resulting from applying Algorithm 5 on the GPL-FH configuration space regarding the four situations discussed above. Out of the 156 configurations, 138 were modified while 18 remained unchanged. Among the 138 modified configurations, 110 were improved regarding both performance indicators, and 16 regarding only one.

[8]Due to inaccuracies in the performance model. See Section 4.3.4 for further analysis.

Performance change	Configurations		Removed LoC			Saved Time (s)		
w.r.t indicators (*Situation*)	Count	%	worst	med.	best	worst	med.	best
Optimized - both indicators (S_1)	110	70	5	69	129	∼0	0,78	20,21
Optimized - one indicator (S_2)	16	10	5	5	76	-1,35	-0,01	∼0
Unchanged (S_3)	18	12	-	-	-	-	-	-
Worsened - both indicators (S_4)	12	8	-69	-31	-31	-0,99	-0,74	-0,15

Table 4.1: The effect of ICO on the GPL-FH configuration space.

As a matter of fact, all these 16 single-indicator optimizations relate to an improvement of `LoC` at the expense of `time`. The remaining 12 configurations worsened on both performance indicators.

> **RQ 1**: These results show the efficiency of ICO: only 8% of the configuration space could not be improved by our approach. 12% remained unchanged as there was no way to further optimize them, and 80% were successfully optimized.

Investigating RQ2: How effective is the ICO optimization approach? Figure 4.11 shows the performance gains when running ICO on the GPL-FH configuration space. As anticipated above, variations in `time` were more significant than the `Loc`-related ones, *i.e.*, ranging from +96,6% to -133,6% regarding `time` and from +26,5% to -17,7% regarding `Loc`. The 12 configurations discussed in Table 4.1 worsen both performance indicators (situation S_4) thus represent a negative gain and as depicted below the horizontal axis and the left side of the vertical axis. The figure highlights that the performance loss on such features is very limited when compared to the performance gains in other situations.

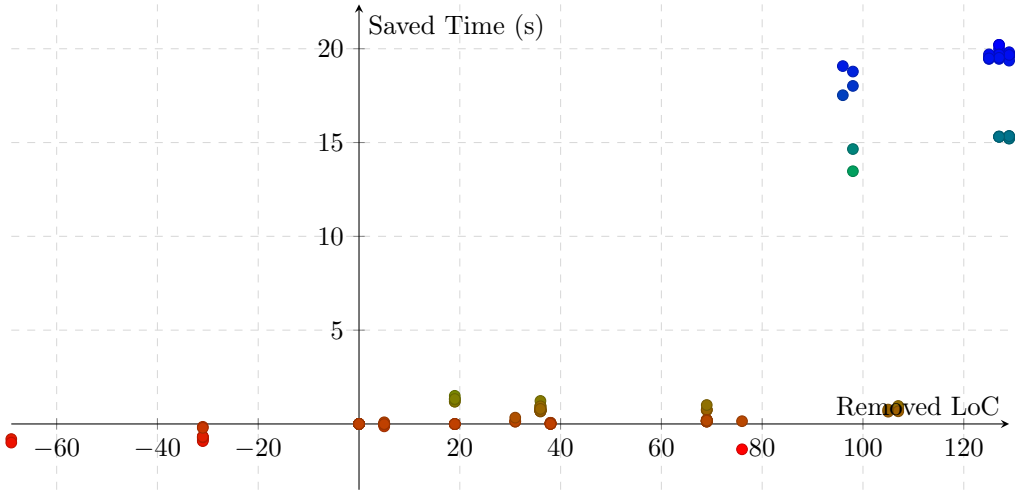


Figure 4.11: Performance gains for each GPL-FH configuration *w.r.t* `LoC` and `time` (top right corner is better).

> **RQ 2**: ICO provides efficient optimizations, especially for poorly performing configurations, but can sometimes worsen configurations' performance. Nevertheless, although worsened, these configurations remain in the top-tier performance ranking.

Investigating RQ3: How many iterations does it take to optimize a configuration? Since an initial configuration cannot be turned into an invalid one by Algorithm 5 (see line 18), running the algorithm on each configuration of the configuration space thus results in a set of optimized configurations which are a subset of the initial configurations. These optimized configurations cannot be further optimized, as they have no neighbor configuration with better performances. Based on this inclusion, it is then possible to build a directed graph representing all successive iterations of the algorithm.

Figure 4.12 depicts such a graph for the GPL-FH case study, where each node represents a configuration. For the sake of readability, nodes are placed on a relative logarithmic scale representing their related configuration's `time` and `LoC`, respectively on the vertical axis and

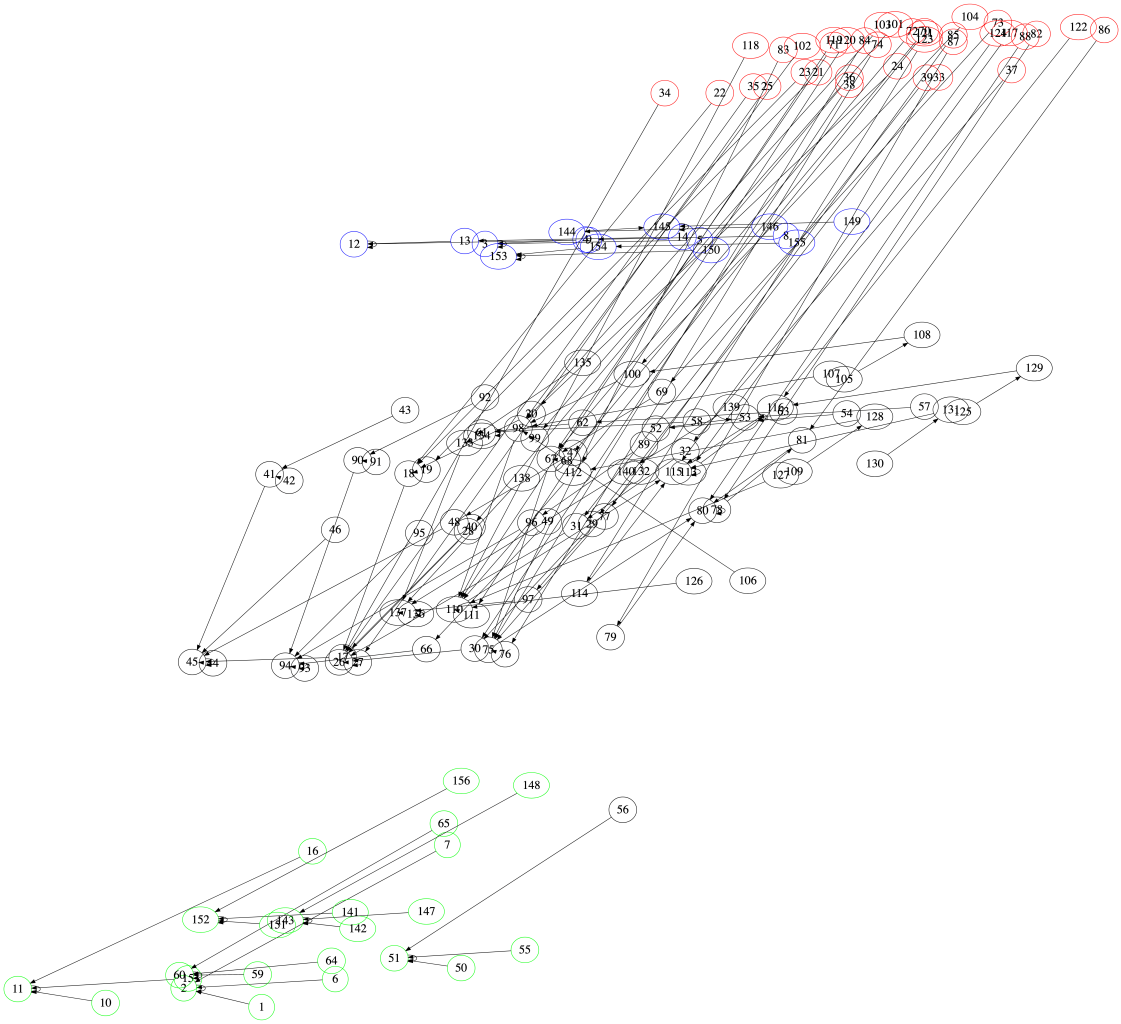


Figure 4.12: ICO transition graph between configurations of GPL-FH. Configurations on a relative logarithmic scale for readability, `time` on the vertical axis, `LoC` on the horizontal axis, lower left is better.

on the horizontal axis. Each edge represents the application of the first suggestion returned by Algorithm 5: the initial configuration is the source node for that edge, while the optimized configuration resulting from applying this first suggestion is the target node. Thus, an edge represents the removal of a feature, the addition of a feature, or the substitution of a feature by another one. This graph is composed of 18 disconnected sub-graphs. Each sub-graph converges towards one of the 18 configurations that could not be optimized and remained unchanged (see Table 4.1, situation S_3). These 18 configurations are thus local optima, and one of them is the global optimum.

Table 4.2 shows the number of iterations of Algorithm 5 required by all configurations to converge towards their related optimized configuration. As explained before, 18 configurations remain unchanged and therefore do not need any iteration of the ICO algorithm to reach their convergence point. Regarding the 138 other configurations, a single iteration drives 61 of them (44.2%) toward their convergence point. That is, after one iteration, 79 configurations (more than half the configuration space) have already converged. After a second iteration, 81% of configurations have reached their convergence point. Up to five iterations are required to optimize the whole set of configurations, but the last two iterations only apply to 3.8% of the configurations.

Nb Iterations	0	1	2	3	4	5
Nb Configurations	18	61	48	22	6	1
Nb Configurations, cumulative	18	79	127	149	155	156
% remaining configurations	11.5	44.2	62.3	75.8	85.7	100
% total configurations	11.5	39.1	30.7	14.1	3.8	0.6
% total configurations, cumulative	11.5	51	81	95	99	100

Table 4.2: Applying ICO on the GPL-FH configuration space.

RQ 3: The number of iterations required by ICO to optimize a configuration is very limited, as *(i)* half of configurations are optimized after a single iteration and *(ii)* the number of configurations yet to be optimized decreases dramatically after each iteration. In this experiment, only 1 configuration required the maximum number of five iterations to be optimized.

4.3.4 Discussion

Performance Model. To perform its optimization process, ICO relies on a performance model. This model provides an estimated performance for each feature, measured based on the method proposed in Section 4.2. As the performance model is estimated, it may contain measurement inaccuracies which in turn may impact the efficiency of the approach. For instance, we observed that, while optimizing GPL-FH, the performance of twelve configurations worsened after an iteration. When analyzing the initial and "optimized" configurations, we found out that the twelve performance regressions were caused by the addition of either the feature `Number` or `Cycle`. Both of these features happen to be present in all the configurations from C_1, the cluster of best-performing configurations. However, such good performances are actually not related to `Number` or `Cycle` alone, but to the presence of other features in combination. The performance model seems thus biased toward `Number` and `Cycle`, which causes inaccuracies during the execution of ICO.

Validity Threats. To assess our approach, we ran our experiments on a specific configurable system (GPL-FH) and optimized it based on specific metrics, *i.e.*, minimizing the execution time and the number of lines of code of configurations from this system. Results such as the performance gains or the number of iterations are thus only related to this single system, and cannot be generalized. Nonetheless, our contribution can be easily applied to any configurable system as long as a feature model is provided. The optimization gains resulting from applying our approach to other configurable systems will depend on the initial performance of each configuration for such systems.

We ran the ICO optimization algorithm on the whole configuration space of GPL-FH. While relying on an exhaustive performance model was convenient, we acknowledge that this may not be practical for any case study, in particular regarding performance models of software systems exhibiting larger configuration spaces. Yet, it is still possible to use our approach by sampling or predicting performance models for such larger spaces, using approaches such as [Nair 2020, Guégain 2021, Acher 2022].

4.4 Summary

In this chapter, we first proposed a method to measure and reduce the energy consumption of multiple products at once by sampling and analyzing a minimal set of products. In particular, our method distinguishes two approaches: one that considers the energy consumption of individual features and one that takes pairs of features into account. The latter takes feature interactions into consideration when measuring the energy consumed by a product to highlight pairs of features that may cooperate or obstruct each other at a behavioral level, while altering the energy spent to complete a task. This approach also suggests candidate features whose interaction with user-required features exhibits lower energy footprint than the one produced by the initial interaction. Based on this method, we then proposed an approach that optimizes a configuration regarding multiple performance objectives. Contrarily to prior work that samples or predicts performance models seeking for the best configuration of the whole configuration space [Švogor 2019, Nair 2020, Kaltenecker 2020], our approach optimizes existing configurations by maximizing performance gains while minimizing changes to such configurations. The objective is to provide the developer with the best-performing configuration by altering as little as possible the initial one, in order to remain as close as possible to the developer's functional requirements.

Material related to this chapter

- **[TODO: JSS]**. Edouard Guégain, Alexandre Bonvoisin, Clément Quinton. Submitted to Journal of Systems and Software on October 2023.
- **ICO : A Platform for Optimizing Highly Configurable Systems**. Edouard Guégain, Amir Taherkordi, Clément Quinton. International Workshop on Automated and verifiable Software sYstem Development, ASYDE 2023.
- **Configuration Optimization with Limited Functional Impact**. Édouard Guégain, Amir Taherkordi, Clément Quinton. International Conference on Advanced Information Systems Engineering, CAiSE 2023. *Core rank: A.*
- **On Reducing the Energy Consumption of Software Product Lines**. Édouard Guégain, Clément Quinton, Romain Rouvoy. International Conference on Systems and Software Product Line Conference, SPLC 2021.
- **On Reducing the Energy Consumption of Software Product Lines**. Reusable artifacts, datasets and experimental results available at `https://zenodo.org/record/5048316`.
- **The ICO Tool Suite**. Source code available at `https://gitlab.inria.fr/ico`.

Chapter 5

Conclusion & Perspectives

Contributions

My research revolves around software variability. In this manuscript, I presented the research I conducted in this domain and the related contributions. Said contributions were grouped into three chapters.

In **Chapter 2**, we present tangible examples illustrating the impact of evolution on the stability of a DSPL. Subsequently, we introduce a flexible approach that leverages a reference architecture to incorporate evolution support into a DSPL. We provide two distinct implementations of this reference architecture, each catering to a different DSPL in separate domains. One implementation targets a cyber-physical system, while the other addresses a runtime monitoring system. These implementations utilize diverse means for managing variability. To assess the feasibility and performance of our approach, we conduct a comprehensive evaluation by simulating typical evolution scenarios for both DSPLs. The results demonstrate that both implementations effectively identify inconsistencies introduced in a DSPL during runtime. Moreover, we successfully apply our approach to a real-world automation software system DSPL in the injection molding domain, thus highlighting its industrial applicability.

Moving on to **Chapter 3**, our research delves into the realm of variability-driven reinforcement learning methodologies for achieving self-adaptation in presence of design time uncertainty. We specifically tackle two issues concerning the exploration of adaptation actions in existing solutions. Firstly, these solutions tend to explore adaptation actions randomly, resulting in slow learning when numerous possible actions are available. Secondly, they lack awareness of evolution, leading to delayed exploration of newly introduced adaptation actions during evolution. To address these challenges, we propose innovative exploration strategies that employ feature models. These strategies leverage the semantics of the feature model to effectively guide exploration through the configuration space. Through our evaluation, we demonstrate that our proposed strategies accelerate the learning process when confronted with a multitude of adaptation actions and in the context of service evolution.

Finally, **Chapter 4** investigates performance variations within configurable systems. Specifically, our investigation centers on the influence of feature interactions on system performance, with a specific emphasis on energy consumption. We introduce a method designed to measure and mitigate the energy consumption of multiple variants at once, achieved through the sampling and analysis of a minimal set of these variants. This approach enables the estimation of individual feature energy consumption, shows the impact of feature interactions on variant energy consumption, and suggests variants with reduced energy usage. Building upon these insights, we propose an approach that guides the optimization of software performance across multiple objectives. This optimization strategy aims to recommend better performing configurations while minimizing alterations to the initial configuration, thereby ensuring alignment with the initial functional requirements.

Perspectives

In this section, I quickly describe some of the perspectives to the work I presented in this manuscript. These perspectives build on much of the work we've done on variability optimization, and especially center on sustainable software development and evolution. I have already begun looking towards green software product lines and the configuration of sustainable cloud-native stacks in an attempt to better understand the role played by software variability to reduce energy consumption of software. As of more recently, I have also begun to take an interest in the growing field of large language models (LLM) for software engineering, to investigate how such models can be leveraged and combined with variability management to better optimize software.

On the Impact of Cloud Services. Cloud computing, by facilitating resource sharing in large data centers, has enabled the optimization of energy consumption. However, it does not prevent the proliferation of data centers, posing the long-term risk of increased energy consumption and carbon footprint. We address this observation from two perspectives.

- *Cloud-native Software.* Most efforts to improve energy efficiency of cloud computing have focused on hardware and infrastructure. This includes works on dynamic server consolidation, hardware design with better power/performance trade-offs, and energy-aware scheduling algorithms, among others. Through the ANR Distiller project, we are currently investigating the design of microservices deployed in layered software stacks on top of cloud infrastructures to minimize resource waste and maximize efficiency. While existing approaches focus on reducing the energy consumption of programming languages and libraries, there is still a lack of knowledge on the energy consumed by each layer of the stack. As part of Alexandre Bonvoisin thesis, we started focusing on the energy consumption of the data access layer, in particular by studying the performances of different Object Relational Mapping (ORM) libraries. Our current study shows significant differences between various ORM provider configurations, impacting the energy consumption of the entire stack. Through several empirical studies, we aim at exploring other layers of the stack (*e.g.*, HTTP layer and REST API) to provide developers with a recommender system which will guide them to apply energy-efficient architectural patterns systematically for their microservices to be deployed in the cloud.

- *Computing with Boundaries.* Together with a geophysicist from IRIS Instrument, France, we recently started working on the materiality of our digital practices. Yes, our digital practices are deeply material. The worldwide rising demand for computing services indeed led to a proliferation of digital devices. However, this demand for digital devices also entails a significant dependence on the mineral resources required to manufacture them as hardware components found in a wide range of devices, such as workstations, tablets, smartphones, PCs, laptops, and even supercomputers, are made from critical minerals. Typically, a smartphone requires 75 natural minerals, involving elements covering almost two-thirds of the periodic table [Emsbo 2021, Pathak 2022]. In light of these considerations, we need dedicated tools and methods to properly assess the mineral impact of digital devices. While *Life Cycle Assessment* (LCA) are dedicated tools to "*study of the environmental impacts' contribution of a product or service across its entire life cycle*"[1], we observe that thorough analyses are still publicly scarce and that the resulting data quality greatly varies. When openly available, their scope and system boundaries are not always explicitly stated, and the uncertainty of the results is almost never quantified. We aim at exploring what metrics and indicators could be used to capture the assessment of mineral impacts, and how to share these metrics among several domains to understand and collectively act upon them.

[1] ISO 14040

Leveraging Large Language Models. We also want to explore emerging models of development assistance, such as the so-called no-code, low-code approaches, or even code assistants. These models, precisely Large Language Models (LLM), have recently increased in popularity, especially with the advent of ChatGPT. They provide support for accelerating initial development phases, and can be used to improve code reliability.

- *LLM Performances.* However, LLM are not specifically designed with sustainability in mind and the environmental footprint of the generated code is still to be studied. As a first step towards this research goal, we started looking as part of Tristan Coignion thesis at the performance of the code generated by such LLM. In particular, we investigated temperature variations in LLM. The temperature of an LLM determines how creative the model is and can affect the correctness of the code it generates. We thus explored how changing the temperature of a LLM may also impact the performance of the generated code, and showed that higher temperatures lead to an increased likelihood of generating inefficient and slow code. As future work, we also plan to investigate the energy impact of the generated code or the one of the coding assistants themselves. That is, measuring the energy consumption of the code generated by the LLM, especially through the comparison of multiple generations (which produce different generated code). Also, we would like to evaluate the energy consumption of the LLM itself when used as a coding assistant, in particular to assess whether it is worth relying on such an assistant regarding the energy consumed by the generated code.

- *LLM for Variability.* The generative nature of LLM makes them good candidates to produce various software variants. Several research topics are thus of interest in the field of LLM for configurable systems. LLM can thus be used to synthesize software variations based on requirements provided as prompts or high-level feature descriptions. When designing a recommender system, LLM can leverage such requirements to suggest influential and interpretable configuration options, helping developers make well-informed configuration decisions. Similarly, LLM can be leveraged to build predictive models that estimate software performance based on various configuration settings. These models will provide insights into the complex relationships between configurations and system performance. Finally, we will investigate the use of LLM for automatic feature identification and modeling within software systems. We aim to develop techniques that can identify and represent configurable units (features) effectively. All of these research directions need further inquiries to propose valid approaches, as variations in prompts or temperature may introduce significant variability-related issues and lead to incorrect generations.

Automated Browser Testing. Browsers are some of the most complex software ever built. There are layers and layers of abstractions, optimizations and features that have taken thousands of man years to build. Browsers and their features are rapidly developed with little interest or caution to privacy issues, opening the door to fingerprinting (*i.e.*, browser's configuration recognition) and other side-channel attacks. They are thus highly-configurable software and their different configurations can introduce many issues. Furthermore, browsers are extensible through extensions, and extensions can also introduce privacy bugs of their own that make users identifiable. However, despite the importance of privacy and the risks that your browser's configuration may be used to identify you, almost all of the privacy improvements have relied on purely manual efforts. Building on the Am I Unique project[2] and as part of Maxime Huyghe thesis, we would like to address this issues by providing an extensible and automated browser testing platform. We thus aim at proposing a platform that automatically explores browser configurations, executes browsers and generates browser fingerprints that are analyzed to find privacy issues and the components at fault, at development-time.

[2]The Am I Unique project https://amiunique.org

Variability Modeling and Knowledge Compilation. This line of work continues the research conducted within the CPER Data Commode project and revolves around two research directions. The first one is sampling large configuration spaces. When the configuration space of a software becomes very large (and it can quickly reach millions of configurations), it becomes impossible to exhaustively verify and test all configurations. Therefore, sampling is necessary, and to ensure that it is representative of the entire configuration space, it is desirable for it to be as random and uniform as possible. Existing approaches propose random solutions that tend towards optimal uniformity but cannot guarantee it. In contrast, we would like to propose a technique for random uniform sampling where uniformity is guaranteed by constructing the representation of the configuration space (in this case, a d-DNNF). The second research direction is about ordering configurations regarding a certain objective function. While existing approaches only allows for randomly enumerating the configuration space (also known as the all-SAT problem), we aim at providing a method for ranked enumeration of the configuration space. Configurations would thus be ranked according to an objective function, *e.g.*, *ranking the three less energy consuming configurations*.

Bibliography

[Abou Khalil 2019] Zeinab Abou Khalil, Eleni Constantinou, Tom Mens, Laurence Duchien and Clément Quinton. *A Longitudinal Analysis of Bug Handling Across Eclipse Releases.* In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 1–12, 2019.

[Acher 2011] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien and Philippe Lahire. *Reverse Engineering Architectural Feature Models.* In Proceedings of the 5th European Conference on Software Architecture (ECSA 2011), pages 220–235, Essen, Germany, 2011. Springer-Verlag.

[Acher 2012] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire and Philippe Merle. *Feature Model Differences.* In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper and Stanislaw Wrycza, editors, Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings, volume 7328 of *Lecture Notes in Computer Science*, pages 629–645. Springer, 2012.

[Acher 2022] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais and Juliana Alves Pereira. *Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size.* In Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A, SPLC '22, page 85–96, New York, NY, USA, 2022. Association for Computing Machinery.

[Angelov 2009] Samuil Angelov, Paul W. P. J. Grefen and Danny Greefhorst. *A classification of software reference architectures: Analyzing their success and effectiveness.* In Joint Working IEEE/IFIP Conf. on Software Architecture, pages 141–150, 2009.

[Apel 2009] Sven Apel and Christian Kästner. *An Overview of Feature-Oriented Software Development.* Journal of Object Technology, vol. 8, no. 5, pages 49–84, 2009.

[Apel 2010] Sven Apel, Wolfgang Scholz, Christian Lengauer and Christian Kastner. *Detecting Dependences and Interactions in Feature-Oriented Design.* In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering, ISSRE '10, page 161–170, 2010.

[Apel 2011] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein and Dirk Beyer. *Detection of Feature Interactions Using Feature-Aware Verification.* In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, page 372–375, 2011.

[Apel 2013] Sven Apel, Alexander Von Rhein, Thomas Thüm and Christian Kästner. *Feature-Interaction Detection Based on Feature-Based Specifications.* Comput. Netw., vol. 57, no. 12, page 2399–2409, August 2013.

[Arabnejad 2017] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi and Giovani Estrada. *A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling.* In 17th Intl Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, pages 64–73, 2017.

[Barrett 2013] Enda Barrett, Enda Howley and Jim Duggan. *Applying reinforcement learning towards automating resource allocation and application scalability in the cloud.* Concurrency and Computation: Practice and Experience, vol. 25, no. 12, pages 1656–1674, 2013.

[Bass 1998] Len Bass, Paul Clement and Rick Kazman. Software architecture in practice (2nd edition). Addison-Wesley Professional, 1998.

[Batory 2011] Don Batory, Peter Höfner and Jongwook Kim. *Feature Interactions, Products, and Composition.* SIGPLAN Not., vol. 47, no. 3, page 13–22, October 2011.

[Benavides 2010] David Benavides, Sergio Segura and Antonio Ruiz-Cortés. *Automated analysis of feature models 20 years later: A literature review.* Information Systems, vol. 35, no. 6, pages 615–636, 2010.

[Bencomo 2010] Nelly Bencomo, Jaejoon Lee and Svein O. Hallsteinsen. *How Dynamic is your Dynamic Software Product Line?* In Proceedings of the 14th International Software Product Line Conference (SPLC 2010) - Volume 2, pages 61–68, Jeju Island, Korea, 2010.

[Berg 2005] Kathrin Berg, Judith Bishop and Dirk Muthig. *Tracing Software Product Line Variability: From Problem to Solution Space.* In Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries, SAICSIT '05, pages 182–191, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[Berger 2013] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski and Krzysztof Czarnecki. *A Study of Variability Models and Languages in the Systems Software Domain.* IEEE Transactions on Software Engineering, vol. 39, no. 12, pages 1611–1640, 2013.

[Borba 2012] Paulo Borba, Leopoldo Teixeira and Rohit Gheyi. *A Theory of Software Product Line Refinement.* Theor. Comput. Sci., vol. 455, pages 2–30, 2012.

[Botterweck 2014] Goetz Botterweck and Andreas Pleuss. Evolution of software product lines, pages 265–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Bourhis 2022] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis and Clément Quinton. *Pseudo Polynomial-Time Top-k Algorithms for d-DNNF Circuits*, 2022.

[Bourhis 2023] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis and Clément Quinton. *Reasoning on Feature Models: Compilation-Based vs. Direct Approaches*, 2023.

[Bu 2013] Xiangping Bu, Jia Rao and Cheng-Zhong Xu. *Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach.* IEEE Trans. Parallel Distrib. Syst., vol. 24, no. 4, pages 681–690, 2013.

[Bürdek 2016] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter and Andy Schürr. *Reasoning about Product-Line Evolution Using Complex Feature Model Differences.* Automated Softw. Eng., vol. 23, no. 4, pages 687–733, December 2016.

[Bąk 2011] Kacper Bąk, Krzysztof Czarnecki and Andrzej Wąsowski. *Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled.* In Brian Malloy, Steffen Staab and Mark van den Brand, editors, Proceedings of the 4th International Conference on Software Language Engineering (SLE 2011), pages 102–122. Springer Berlin Heidelberg, Braga, Portugal, 2011.

[Calinescu 2020] Radu Calinescu, Raffaela Mirandola, Diego Perez-Palacin and Danny Weyns. *Understanding Uncertainty in Self-adaptive Systems.* In IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020, pages 242–251. IEEE, 2020.

[Capilla 2014] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés and Mike Hinchey. *An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry.* Journal of Systems and Software, vol. 91, pages 3–23, May 2014.

[Caporuscio 2016] Mauro Caporuscio, Mirko D'Angelo, Vincenzo Grassi and Raffaela Mirandola. *Reinforcement Learning Techniques for Decentralized Self-adaptive Service Assembly.* In 5th Eur. Conference on Service-Oriented and Cloud Computing, ESOCC'16, volume 9846, pages 53–68, 2016.

[Chen 2014] Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh and Wenyun Zhao. *Self-adaptation through incremental generative model transformations at runtime.* In 36th Intl Conf on Softw. Eng., ICSE '14, pages 676–687, 2014.

[Czarnecki 2012] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid and Andrzej Wąsowski. *Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches.* In Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012), pages 173–182, Leipzig, Germany, 2012. ACM.

[Czarnecki 2013] Krzysztof Czarnecki. *Variability in Software: State of the Art and Future Directions.* In Vittorio Cortellessa and Dániel Varró, editors, Fundamental Approaches to Software Engineering, pages 1–5, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[de la Iglesia 2015] Didac Gil de la Iglesia and Danny Weyns. *MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems.* TAAS, vol. 10, no. 3, pages 15:1–15:31, 2015.

[De Lemos 2013] Rogério De Lemos*et al.* *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap.* In Softw. Eng. for Self-Adaptive Systems II, volume 7475 of *LNCS*, pages 1–32. Springer, 2013.

[Dhungana 2011] Deepak Dhungana, Paul Grünbacher and Rick Rabiser. *The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study.* Automated Software Engineering, vol. 18, no. 1, pages 77–114, 2011.

[Egyed 2006] Alexander Egyed. *Instant consistency checking for the UML.* In Proceedings of the 28th Int'l Conference on Software Engineering (ICSE 2006), pages 381–390, Shanghai, China, 2006. ACM.

[Emsbo 2021] Poul Emsbo, Christopher Lawley and Karol Czarnota. *Geological surveys unite to improve critical mineral security.* Eos Science News, 2021.

[Filho 2017] Roberto Vito Rodrigues Filho and Barry Porter. *Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning.* TAAS, vol. 12, no. 3, pages 16:1–16:25, 2017.

[Font 2015] Jaime Font, Manuel Ballarín, Øystein Haugen and Carlos Cetina. *Automating the Variability Formalization of a Model Family by Means of Common Variability Language.* In Proceedings of the 19th International Software Product Line Conference (SPLC 2015), pages 411–418, Nashville, Tennessee, USA, 2015.

[Galster 2011] Matthias Galster and Paris Avgeriou. *Empirically-grounded Reference Architectures: A Proposal.* In Joint ACM SIGSOFT Conf. – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems, pages 153–158. ACM, 2011.

[Gamez 2011] Nadia Gamez and Lidia Fuentes. *Software Product Line Evolution with Cardinality-Based Feature Models.* In Proceedings of the 12th International Conference on Software Reuse (ICSR 2011), pages 102–118, Pohang, South Korea, 2011. Springer Berlin Heidelberg.

[Ghezzi 2017] Carlo Ghezzi. *Of software and change.* Journal of Software: Evolution and Process, vol. 29, no. 9, 2017.

[Guégain 2021] Édouard Guégain, Clément Quinton and Romain Rouvoy. *On Reducing the Energy Consumption of Software Product Lines.* In Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A, SPLC '21, page 89–99, 2021.

[Guégain 2023] Edouard Guégain, Amir Taherkordi and Clément Quinton. *ICO: A Platform for Optimizing Highly Configurable Systems.* In 5th International Workshop on Automated and verifiable Software sYstem DEvelopment, Kirchberg, Luxembourg, September 2023.

[Guo 2012] Jianmei Guo, Yinglin Wang, Pablo Trinidad and David Benavides. *Consistency Maintenance for Evolving Feature Models.* Expert System Applications, vol. 39, no. 5, pages 4987–4998, April 2012.

[Hallsteinsen 2008] S. Hallsteinsen, M. Hinchey, Sooyong Park and K. Schmid. *Dynamic Software Product Lines.* Computer, vol. 41, no. 4, pages 93–95, April 2008.

[Hinchey 2012] M. Hinchey, Sooyong Park and K. Schmid. *Building Dynamic Software Product Lines.* Computer, vol. 45, no. 10, pages 22–26, Oct 2012.

[Islam 2016] Syed Islam, Adel Noureddine and Rabih Bashroush. *Measuring energy footprint of software features.* In 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pages 1–4, 2016.

[Jagroep 2016] Erik A. Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom and Rob van Vliet. *Software Energy Profiling: Comparing Releases of a Software Product.* In Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, page 523–532, 2016.

[Jin 2014] Dongpu Jin, Xiao Qu, Myra B. Cohen and Brian Robinson. *Configurations Everywhere: Implications for Testing and Debugging in Practice.* In Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, page 215–224, New York, NY, USA, 2014. Association for Computing Machinery.

[Johansen 2012] Martin Fagereng Johansen, Øystein Haugen and Franck Fleurey. *An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models.* In Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, page 46–55, 2012.

[Kaltenecker 2020] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund and Sven Apel. *The interplay of sampling and machine learning for software performance prediction.* IEEE Software, vol. 37, no. 4, pages 58–66, 2020.

[Kephart 2003] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing.* IEEE Computer, vol. 36, no. 1, pages 41–50, 2003.

[Kinneer 2018] Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan and Claire Le Goues. *Managing uncertainty in self-adaptive systems with plan reuse and stochastic search.* In 13th Intl Symposium on Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS'18, pages 40–50, 2018.

[Mann 2016] Zoltán Ádám Mann. *Interplay of Virtual Machine Selection and Virtual Machine Placement.* In 5th European Conf. on Service-Oriented and Cloud Computing, ESOCC'16, volume 9846, pages 137–151, 2016.

[Mann 2018] Zoltán Ádám Mann. *Resource optimization across the cloud stack.* IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 1, pages 169–182, 2018.

[Marques 2019] Maira Marques, Jocelyn Simmonds, Pedro O Rossel and María Cecilia Bastarrica. *Software product line evolution: A systematic literature review.* Information and Software Technology, vol. 105, pages 190–208, 2019.

[Martinez 2018] Jabier Martinez, Xhevahire Tërnava and Tewfik Ziadi. *Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study.* In Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC '18, page 132–142, 2018.

[McGregor 2003] John McGregor. *The Evolution of Product Line Assets.* Technical report CMU/SEI-2003-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2003.

[Metzger 2014] Andreas Metzger and Klaus Pohl. *Software Product Line Engineering and Variability Management: Achievements and Challenges.* In Future of Software Engineering, FOSE'14, pages 70–84, 2014.

[Metzger 2020] Andreas Metzger, Clément Quinton, Zoltán Ádám Mann, Luciano Baresi and Klaus Pohl. *Feature Model-Guided Online Reinforcement Learning for Self-Adaptive Services.* In ICSOC, volume 12571 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2020.

[Moustafa 2014] Ahmed Moustafa and Minjie Zhang. *Learning Efficient Compositions for QoS-Aware Service Provisioning.* In IEEE Intl Conf. on Web Services, ICWS'14, pages 185–192, 2014.

[Moustafa 2018] Ahmed Moustafa and Takayuki Ito. *A deep reinforcement learning approach for large-scale service composition.* In Intl. Conf. on Principles and Practice of Multi-Agent Systems, pages 296–311, 2018.

[Munoz 2019] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes and Don Batory. *Uniform random sampling product configurations of feature models that have numerical features.* In Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A, pages 289–301, 2019.

[Nachum 2017] Ofir Nachum, Mohammad Norouzi, Kelvin Xu and Dale Schuurmans. *Bridging the Gap Between Value and Policy Based Reinforcement Learning.* In Advances in Neural Inform. Proc. Systems 12 (NIPS 2017), pages 2772–2782, 2017.

[Nair 2020] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund and Sven Apel. *Finding Faster Configurations Using FLASH.* IEEE Transactions on Software Engineering, vol. 46, no. 7, pages 794–811, 2020.

[Nakagawa 2012] Elisa Yumi Nakagawa, Flávio Oquendo and Martin Becker. *RAModel: A Reference Model for Reference Architectures.* In Joint Working IEEE/IFIP Conf. on Software Architecture and European Conference on Software Architecture, pages 297–301, 2012.

[Neves 2011] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa and Paulo Borba. *Investigating the Safe Evolution of Software Product Lines.* In Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE 2011), pages 33–42, Portland, Oregon, USA, 2011. ACM.

[Neves 2015] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena and Uirá Kulesza. *Safe Evolution Templates for Software Product Lines.* Journal of Systems and Software, vol. 106, no. C, pages 42–58, August 2015.

[Noureddine 2013] Adel Noureddine, Romain Rouvoy and Lionel Seinturier. *A Review of Energy Measurement Approaches.* SIGOPS Oper. Syst. Rev., vol. 47, no. 3, page 42–49, November 2013.

[Noureddine 2015] Adel Noureddine, Romain Rouvoy and Lionel Seinturier. *Monitoring energy hotspots in software.* Automated Software Engineering, vol. 22, 9 2015.

[Palm 2020] Alexander Palm, Andreas Metzger and Klaus Pohl. *Online Reinforcement Learning for Self-Adaptive Information Systems.* In Eric Yu and Schahram Dustdar, editors, Int'l Conference on Advanced Information Systems Engineering, CAiSE'20, 2020.

[Pang 2016] Candy Pang, Abram Hindle, Bram Adams and Ahmed E. Hassan. *What Do Programmers Know about Software Energy Consumption?* IEEE Software, vol. 33, no. 3, pages 83–89, 2016.

[Passos 2013] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski and Paulo Borba. *Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel.* In Proceedings of the 17th International Software Product Line Conference (SPLC 2013), pages 91–100, Tokyo, Japan, 2013. ACM.

[Passos 2015] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba and Jianmei Guo. *Coevolution of variability models and related software artifacts.* Empirical Software Engineering, pages 1–50, 2015.

[Pathak 2022] M. Pathak, R. Slade, P.R. Shukla, J. Skea, R. Pichs-Madruga and D. Ürge-Vorsatz. *Technical Summary.* In P.R. Shukla, J. Skea, R. Slade, A. Al Khourdajie, R. van Diemen, D. McCollum, M. Pathak, S. Some, P. Vyas, R. Fradera, M. Belkacemi, A. Hasija, G. Lisboa, S. Luz and J. Malley, editors, Climate Change 2022: Mitigation of Climate Change. Contribution of Working Group III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change. Cambridge University Press, Cambridge, UK and New York, NY, USA, 2022.

[Pereira 2020] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes and João Saraiva. *SPELLing out energy leaks: Aiding developers locate energy inefficient code.* Journal of Systems and Software, vol. 161, page 110463, 3 2020.

[Pereira 2021] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck and Anthony Ventresque. *Learning software configuration spaces: A systematic literature review.* Journal of Systems and Software, vol. 182, page 111044, 2021.

[Plappert 2018] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel and Marcin Andrychowicz. *Parameter Space Noise for Exploration.* In 6th Intl Conf. on Learning Representations, ICLR 2018. OpenReview.net, 2018.

[Pleuss 2012] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer and Stefan Kowalewski. *Model-driven Support for Product Line Evolution on Feature Level.* Journal of Systems and Software, vol. 85, no. 10, pages 2261–2274, October 2012.

[Prud'homme 2014] Charles Prud'homme, Jean-Guillaume Fages and Xavier Lorca. *Choco3 Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[Quinton 2013] Clément Quinton, Nicolas Haderer, Romain Rouvoy and Laurence Duchien. *Towards Multi-cloud Configurations Using Feature Models and Ontologies.* In Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds (MultiCloud 2013), pages 21–26, Prague, Czech Republic, 2013. ACM.

[Quinton 2014] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien and Goetz Botterweck. *Consistency Checking for the Evolution of Cardinality-based Feature Models.* In Proceedings of the 18th International Software Product Line Conference (SPLC 2014), pages 122–131, Florence, Italy, 2014. ACM.

[Quinton 2015] Clément Quinton, Rick Rabiser, Michael Vierhauser, Paul Grünbacher and Luciano Baresi. *Evolution in dynamic software product lines: challenges and perspectives.* In SPLC, pages 126–130. ACM, 2015.

[Quinton 2016] Clement Quinton, Daniel Romero and Laurence Duchien. *SALOON: a Platform for Selecting and Configuring Cloud Environments.* Software: Practice and Experience, vol. 46, no. 1, pages 55–78, 2016.

[Rabiser 2015] Rick Rabiser, Michael Vierhauser and Paul Grünbacher. *Variability Management for a Runtime Monitoring Infrastructure.* In Proceedings of the 9th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2015), pages 35–42, Hildesheim, Germany, 2015. ACM.

[Ramirez 2010] Andres J. Ramirez, Betty H. C. Cheng, Philip K. McKinley and Benjamin E. Beckmann. *Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration.* In 7th Intl Conf. on Autonomic Computing, ICAC'10, pages 225–234, 2010.

[Romero 2015] Daniel Romero, Clément Quinton, Laurence Duchien, Lionel Seinturier and Carolina Valdez. *SmartyCo: Managing Cyber-Physical Systems for Smart Environments.* In Danny Weyns, Raffaela Mirandola and Ivica Crnkovic, editors, 9th European Conference, ECSA 2015, volume 9278 of *Lecture Notes in Computer Science, Software Architecture*, pages 294–302, Dubrovnik/Cavtat, Croatia, September 2015.

[Salehie 2009] Mazeiar Salehie and Ladan Tahvildari. *Self-adaptive software: Landscape and research challenges.* TAAS, vol. 4, no. 2, 2009.

[Sayagh 2020] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams and Fabio Petrillo. *Software Configuration Engineering in Practice Interviews, Survey, and Systematic Literature Review.* IEEE Transactions on Software Engineering, vol. 46, no. 6, pages 646–673, 2020.

[Seidl 2012] Christoph Seidl, Florian Heidenreich and Uwe Aßmann. *Co-evolution of Models and Feature Mapping in Software Product Lines.* In Proceedings of the 16th International Software Product Line Conference (SPLC 2012), pages 76–85, Salvador, Brazil, 2012. ACM.

[Shaw 2022] Rachael Shaw, Enda Howley and Enda Barrett. *Applying Reinforcement Learning towards Automating Energy Efficient Virtual Machine Consolidation in Cloud Data Centers.* Inf. Syst., vol. 107, no. C, jul 2022.

[She 2011] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski and Krzysztof Czarnecki. *Reverse Engineering Feature Models.* In Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pages 461–470, Waikiki, Honolulu, HI, USA, 2011. ACM.

[Siegmund 2012] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller and Gunter Saake. *Predicting Performance via Automated Feature-Interaction Detection.* In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, page 167–177, Zurich, Switzerland, 2012.

[Siegmund 2015] Norbert Siegmund, Alexander Grebhahn, Sven Apel and Christian Kästner. *Performance-Influence Models for Highly Configurable Systems.* In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, page 284–294, 2015.

[Siegmund 2020] Norbert Siegmund, Nicolai Ruckel and Janet Siegmund. *Dimensions of Software Configuration: On the Configuration Context in Modern Software Development.* In

Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 338–349, New York, NY, USA, 2020. Association for Computing Machinery.

[Steinberg 2009] David Steinberg, Frank Budinsky, Marcelo Paternostro and Ed Merks. Emf: Eclipse modeling framework 2.0. Addison-Wesley Professional, 2nd édition, 2009.

[Sutton 2018] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. MIT Press, Cambridge, MA, USA, 2nd édition, 2018.

[Tartler 2011] Reinhard Tartler, Daniel Lohmann, Julio Sincero and Wolfgang Schröder-Preikschat. *Feature consistency in compile-time-configurable system software: facing the linux 10, 000 feature problem.* In Proceedings of the Sixth European conference on Computer systems (EuroSys 2011), pages 47–60, Salzburg, Austria, 2011.

[Taylor 2009] Matthew E. Taylor and Peter Stone. *Transfer Learning for Reinforcement Learning Domains: A Survey.* J. Mach. Learn. Res., vol. 10, pages 1633–1685, 2009.

[Thüm 2009] Thomas Thüm, Don Batory and Christian Kastner. *Reasoning About Edits to Feature Models.* In Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pages 254–264, Vancouver, British Columbia, Canada, 2009. IEEE Computer Society.

[Thüm 2011] Thomas Thüm, Christian Kästner, Sebastian Erdweg and Norbert Siegmund. *Abstract Features in Feature Modeling.* In 15th Intl Conf. on Software Product Lines, SPLC'11, pages 191–200, 2011.

[Thüm 2014] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake and Thomas Leich. *FeatureIDE: An extensible framework for feature-oriented software development.* Science of Computer Programming, vol. 79, pages 70–85, 1 2014.

[Vierhauser 2012] Michael Vierhauser, Paul Grünbacher, Wolfgang Heider, Gerald Holl and Daniela Lettner. *Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines.* In Proceedings of the 15th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2012), pages 531–545, Innsbruck, Austria, 2012.

[Vierhauser 2015] Michael Vierhauser, Rick Rabiser, Paul Grünbacher and Alexander Egyed. *Developing a DSL-Based Approach for Event-Based Monitoring of Systems of Systems: Experiences and Lessons Learned.* In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), pages 715–725, Lincoln, Nebraska, USA, 2015. ACM.

[Vierhauser 2016] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner and Helmut Zeisel. *ReMinds: A Flexible Runtime Monitoring Framework for Systems of Systems.* Journal of Systems and Software, vol. 112, pages 123–136, 2016.

[Wang 2017] Hongbing Wang, Mingzhu Gu, Qi Yu, Huanhuan Fei, Jiajie Li and Yong Tao. *Large-Scale and Adaptive Service Composition Using Deep Reinforcement Learning.* In 15th Intl Conference on Service-Oriented Computing (ICSOC'17), pages 383–391, 2017.

[Wang 2019] Hongbing Wang, Mingzhu Gu, Qi Yu, Yong Tao, Jiajie Li, Huanhuan Fei, Jia Yan, Wei Zhao and Tianjing Hong. *Adaptive and large-scale service composition based on deep reinforcement learning.* Knowledge-Based Systems, vol. 180, pages 75–90, 2019.

[Weinreich 2014] Rainer Weinreich and Georg Buchgeher. *Automatic reference architecture conformance checking for soa-based software systems.* In Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture (WICSA 2014), pages 95–104, Sydney, Australia, 2014. IEEE.

[Weyns 2013] Danny Weyns and et al. *Perpetual Assurances for Self-Adaptive Systems.* In Rogério de Lemos, David Garlan, Carlo Ghezzi and Holger Giese, editors, Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers, volume 9640 of *Lecture Notes in Computer Science*, pages 31–63. Springer, 2013.

[Weyns 2021] Danny Weyns. Introduction to self-adaptive systems: A contemporary software engineering perspective. Wiley, 2021.

[Xu 2015] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy and Rukma Talwadker. *Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software.* In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, page 307–319, New York, NY, USA, 2015. Association for Computing Machinery.

[Zhao 2017] Tianqi Zhao, Wei Zhang, Haiyan Zhao and Zhi Jin. *A Reinforcement Learning-Based Framework for the Generation and Evolution of Adaptation Rules.* In Intl Conf. on Autonomic Computing, ICAC, pages 103–112, 2017.

[Švogor 2019] Ivan Švogor, Ivica Crnković and Neven Vrček. *An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study.* Information and Software Technology, vol. 105, pages 30–42, 2019.