SpringerBriefs in Computer Science

Alexander Felfernig · Andreas Falkner · David Benavides

Feature Models
AI-Driven Design, Analysis and Applications

OPEN ACCESS

Springer

SpringerBriefs in Computer Science

Alexander Felfernig • Andreas Falkner
David Benavides

Feature Models

AI-Driven Design, Analysis
and Applications

Springer

Alexander Felfernig
Institute of Software Technology
Graz University of Technology
Graz, Austria

Andreas Falkner
Corporate Technology
Siemens (Austria)
Wien, Austria

David Benavides
ETS de Ingeniería Informática
University of Seville
Sevilla, Spain

ISSN 2191-5768 ISSN 2191-5776 (electronic)
SpringerBriefs in Computer Science
ISBN 978-3-031-61873-4 ISBN 978-3-031-61874-1 (eBook)
https://doi.org/10.1007/978-3-031-61874-1

The work presented in this book was partially supported by: (1) the Austrian Research Promotion Agency (FFG) (www.ffg.at), (2) the FEDER/Ministry of Science and Innovation/Junta de Andalucia/State Research Agency (www.ciencia.gob.es), and (3) the company sponsors Siemens (www.siemens.com) and Lam Research (www.lamresearch.com).

Preface

Feature models (FMs) have become a fundamental means of representing variability knowledge related to software systems, services, and also physical products such as furniture, cars, and cyber–physical systems. They define all allowed combinations of the features representing possible variants of a product. FMs provide a language and a corresponding formal semantics that helps to support reasoning operations, for example, finding correct FM configurations and analyzing FMs. The ever–increasing amount of research on the integration of Artificial Intelligence (AI) methods into feature modelling related processes motivated us to write this book on *Feature Models: AI-driven Design, Analysis, and Applications*. Its purpose is to provide a basic introduction to feature modelling and analysis as well as the integration of AI methods with feature modelling. This book is intended as an introduction for persons new to the field and also as reference material for researchers, teachers, and practitioners. More specifically, while focusing on the AI perspective, the book covers the topics of feature modelling, FM analysis, and interacting with FM configurators. These topics are discussed along the AI areas of knowledge representation and reasoning (KRR), explainable AI (XAI), and machine learning (ML). Last not least a personal note: we decided to order the author names by the order of our first names.

Graz, Vienna, Sevilla
April 2024

Alexander Felfernig
Andreas Falkner
David Benavides

Acknowledgements

We want to thank the following colleagues for their valuable contributions in terms of providing feedback on the book contents that helped us to develop a broader view on the topic and also to significantly improve understandability: *Don Batory* (University of Texas), *José A. Galindo, David Romero, and José A. Zamudio* (University of Seville), *José M. Horcas* (University of Málaga), *Viet-Man Le, Sebastian Lubos, Thi Ngoc Trang Tran, Damian Garber, and Tamim Burgstaller* (Graz University of Technology), and *Mathias Uta* (Siemens Energy). Furthermore, special thanks goes to *Jean Marc Jezequel* who provided invaluable inspirations regarding an easy-to-understand complexity characterization of feature models. The work presented in this book was partially supported by: (1) the *Austrian Research Promotion Agency* (FFG) (www.ffg.at) within the scope of the research projects ParXCel (880657) and OpenSpace (FO999891127), (2) the *FEDER/Ministry of Science and Innovation/Junta de Andalucía/State Research Agency* (www.ciencia.gob.es) with the following grants: Data-pl(PID2022-138486OB-I00), Tasova Plus research network (RED2022-134337-T) and MIDAS (IDI-20230256), and (3) the company sponsors Siemens (www.siemens.com) and Lam Research (www.lamresearch.com) who made the open-access publishing of this book possible.

Contents

Chapter 1
Introduction

Abstract *Feature models* (FMs) are an established means for representing variability and commonality properties of software product lines and beyond (e.g., financial services and configurable products such as furniture, cars, and cyber-physical systems). *Artificial Intelligence* (AI) plays an increasingly important role in supporting feature modelling tasks, FM analysis, and FM configuration. In this chapter, we explain our major motivation for writing this book. We provide a short overview of the history of feature modelling specifically focusing on the relationship between feature modelling related tasks and AI methods. We discuss relevant benefits of applying FMs and refer to further topics with a relationship to feature modelling. We conclude this chapter with an overview of the major topics of in this book.

1.1 Motivation for the Book

Feature models (FMs) are a wide-spread means for representing variability properties of software product lines (SPL) [2, 6, 9, 10, 16, 47] as well as configurable products and services [9, 26, 45]. Major advantages of FMs are that (1) they are easy to understand and develop (only a few modelling concepts with a clear semantics are provided which are sufficient in many application scenarios), (2) they can be directly translated into a corresponding formal representation, for example, a constraint satisfaction problem (CSP) [53] or a Boolean satisfiability (SAT) problem [15] which allows for automated and efficient reasoning processes, and (3) there exists a plethora of tools for FM-based variability management (see Chapter 5).

On an informal level, FMs represent configuration spaces with a graphical representation in terms of (1) features which can be included in a configuration or excluded (i.e., are not part of the configuration) and (2) a set of constraints which restrict the combinations of individual features in a final configuration. An FM of a configurable *smartwatch* will be used as a working example throughout this book. Example features of a smartwatch are *payment* and *screen* (screentype of a smartwatch). A related constraint could specify that if a user is interested in a standard

https://doi.org/10.1007/978-3-031-61874-1_1

screen, no payment feature is available, i.e., the payment feature is incompatible with the standard feature. The basic concept is that a set of similar products can be described in terms of *features* and relationships among them. An FM represents allowed combinations of features (for details, see Chapter 2).

Our major motivation for writing this book is that Artificial Intelligence (AI) methods and techniques play an increasingly important role in variability management processes where FMs are a central element [7, 8]. On the basis of an analysis of existing AI approaches in software product lines and knowledge-based configuration, we discuss these approaches in the context of the identified categories of (1) feature modelling (Chapter 2), FM analysis (Chapter 3), interacting with FM configurators (Chapter 4), and related tools and applications (Chapter 5).

Another motivation was to move forward towards a more integrative view on the topics addressed in different communities (1) the SPL community, exemplified by the Software Product Line Conference (SPLC) and the Working Conference on Variability Modelling of Software Intensive Systems (VaMoS), and (2) the knowledge-based configuration community, exemplified by the Workshop on Configuration (ConfWS). With this, we expect to foster more intensive cooperation in related fields and also indicate relevant open research issues to these communities.

1.2 A Short History of Feature Models

FMs and software product lines are software engineering key technologies for producing highly configurable software products. The concept of FMs was invented in the early 1990's by Kang et al. in their 1990 seminal paper "Feature-oriented Domain Analysis (FODA) Feasibility Study" [39]. There were previous works on similar topics that settled the basis for modern software product line engineering approaches. McIlroy's paper in 1969 on "Mass Produced Software Components" is probably the most important seminal paper [42]. The key idea was that customized software should be industrialized as in other domains such as hardware.

Following basic FMs, different variants thereof were proposed in the late 1990's and 2000's [9, 29] ranging from cardinality-based FM representations [19] to FMs taking into account feature attributes [11, 39].

An increasing adoption of feature modelling and software product lines in industry could be observed starting in the early 2000's [9, 29]. From that time on, FMs became a central element of reuse-driven development processes for highly-configurable software systems [54]. This increasing industrial relevance was observed in industries such as electronic components and car manufacturing [13].

Following the adoption of FMs in industry, the quality of tool support increased from that time on until now (and is still continuing) resulting in various tools/frameworks and applications (see, e.g., Meinicke et al. [43] and Beuche [14]). Examples of related open source and commercial tools are discussed in Chapter 5.

Nowadays, FMs in the context of software product lines (SPLs) are in wide-spread use in industry as well as in academia with applications [35] ranging from operating systems [54], software systems for controlling trains on various hardware platforms and in different countries [1], automotive systems [20, 22, 64], synthetic biology [17], to software product lines for large telescope control software [32], just to mention a few. In the late 2010's and 2020's, the ever-increasing popularity of Artificial Intelligence methods – specifically, machine learning (ML) – also had enormous impacts on feature modelling and variability management research. Examples of related research are the application of ML to personalized FM configuration [27, 50, 52] and configuration space learning [31, 48]. The integration of AI with FMs is the central topic of this book and will be discussed throughout Chapters 2–5.

Similar to SPLs which help to customize software systems, *product configuration* is about customizing hardware (and more). Felfernig et al. [26] relate it to the mass customization paradigm [37] which is based on the idea of the customer-individual production of highly variant products under near mass production pricing conditions. Sabin and Weigel [55] define configuration as a *special case of design activity where the artifact being configured is assembled from instances of a fixed set of well-defined component types which can be composed conforming to a set of constraints*. Configuration has been one of the most successfully applied technologies of AI for several decades and in many application domains [26, 55, 58].

We want to emphasize that specifically in the context of highly-configurable products, configuration solutions were already developed throughout the 1970's and 1980's a.o. in the context of the R1/XCON computer configurator [5]. These systems focused on rule-based knowledge representation and reasoning resulting in serious efforts in configuration model development and maintenance. At the same time as initial versions of FM languages were developed [39], configuration knowledge representation and reasoning moved away from rule-based representations to so-called model-based knowledge representations allowing a clear separation of product domain and reasoning knowledge (e.g., in terms of search heuristics). Related emerging (model-based) reasoning techniques such as constraint solving [53] and SAT solving [15] became established in both fields of research, i.e., feature modelling [6, 39] and knowledge-based configuration [26, 45, 55, 58].

Although the research communities of feature modelling and knowledge-based configuration were established in parallel and in many cases work on similar topics, we can observe an increasing degree of cooperation which appears to be fruitful for both research communities [8]. Today, *SPLC* and *VaMoS* can be regarded as major scientific conferences for FM-related topics whereas the *Configuration Workshop* (*ConfWS*) is the platform for research on topics of knowledge-based configuration (and beyond). In 2022, *SPLC* and *ConfWS* were co-located the first time.[1]

[1] https://2022.splc.net/

1.3 The Role of AI in Feature Models

Artificial Intelligence (AI) plays an increasing important role in different FM-related tasks [27]. An overview is given in Table 1.1. We distinguish between the tasks of (1) *feature modelling*, (2) *FM analysis*, and (3) *FM configuration* (i.e., interacting with configurators). Those topics are discussed in Chapters 2–4. We now discuss the concepts of Table 1.1 in more detail.

Artificial Intelligence (AI) Aspects		**FM-related Tasks**		
AI Areas	**Example AI Techniques**	Feature modelling (**Chapter 2**)	FM analysis (**Chapter 3**)	Interacting with FM configurators (**Chapter 4**)
knowledge representation (KR)	knowledge graphs, answer set programs (ASP)	basic, attribute-, and cardinality-based FMs	analysis operations (with/without solver)	basic, attribute-, and cardinality-based FM configuration
reasoning (R)	CSP solving, SAT solving, rule-based reasoning	CSP-based, SAT-based, rule-based, text-based FM formalization	CSP-, SAT-, and rule-based analysis	configuration (CSP, SAT, and rule-based)
explainable AI (XAI)	argumentation, conflict detection, model-based diagnosis	argumentation- and consistency-based explanations	FM inconsistencies, FM redundancies	explaining configurations and inconsistencies
machine learning (ML)	prediction (e.g., regression, factorization), classification (e.g., neural networks, LLMs, decision trees)	configuration space learning, knowledge extraction from data	predicting faulty FM model elements	recommending features and (re)configurations

Table 1.1: Artificial Intelligence (AI) aspects covered in this book and relationships to *feature modelling*, *analysis*, and *configuration* (*interacting with configurators*).

AI Aspects. Our categorization of different AI areas relevant in the context of FMs is based on the following scheme. First, the role of *knowledge representation & reasoning* (KRR) [63] is to develop appropriate concepts and languages for representing variability properties and support efficient problem solving (reasoning) procedures. Examples of related AI techniques are (1) knowledge graphs [34] and answer set programs [46], and (2) SAT solving [15], constraint solving [53], and rule-based reasoning [30]. Second, following the idea of *explainable AI* (XAI) [21], solutions as well as problems (e.g., inconsistencies) need to be explained such that users understand why a specific configuration has been proposed or no solution could be found. Examples of AI techniques supporting such tasks are argumentation [12], conflict detection [38], and model-based diagnosis [51]. Finally, different types of *machine learning* (ML) [44] approaches can help to support *prediction* (e.g., what will be the maximum price accepted by the user) and *classification* tasks (e.g., will a specific feature be of interest for the user). Machine learning is applied to provide a personalized user experience in feature modelling (see, e.g., [27]).

Feature Modelling. To be applicable in FM configuration, FMs have to be designed (typically, this is performed on a graphical level) and then translated into a corresponding formal representation (FM formalization) that is a basis for the follow-up tasks of FM analysis and FM configuration (interacting with configurators). FM formalization can be based on different AI-based approaches such as SAT solving, constraint solving, and rule-based reasoning. For demonstration purposes, we will focus our discussions on constraint-based representations, however, most of the discussed concepts can as well be applied with the mentioned alternatives. FM design and FM formalization will be discussed in Chapter 2. FM design also depends on decisions regarding the inclusion of specific constraints, for example, regarding the combination of individual features and also on decisions regarding the inclusion or exclusion of features. An important task in this context is *product line scoping* which entails methods and techniques helping to figure out relevant features and corresponding constraints describing the envisioned (software) product line. Example AI techniques that can be used in this context are explanations (that help to understand inclusion and exclusion decisions), conflict detection (pointing out inconsistencies that need to be resolved), and diagnosis (in which way conflicts should be resolved to develop consensus regarding the final shape of an FM). Some explanation-related aspects of product line scoping will be discussed in Chapter 2. Finally, to assure efficient solution search, AI techniques can also be used to support developers in optimizing FM configurator search heuristics and in predicting the performance of FM configurations (see Chapter 2).

FM Analysis. This task covers different aspects of assuring the quality of FMs with regard to aspects such as model consistency and FM complexity (e.g., in terms of the number of supported solutions). Some of the related analysis operations can be performed without solver support (e.g., counting the number of features and constraints) and other operations are in the need of solver support (e.g., checking model satisfiability, counting or approximating the number of supported solutions (configurations), and checking if some of the features are dead, i.e., cannot be included in a configuration). Different types of FM analysis operations and their relevance in modelling contexts are discussed in Chapter 3. Besides the mentioned analysis operations, FMs can also be tested with regard to conformance with the underlying application domain. In this context, FM development and maintenance can be supported with different types of diagnosis and repair functions that help to locate the sources of inconsistent FM behaviors. FM inconsistencies could be predicted using ML approaches. Related aspects are also discussed in detail in Chapter 3.

Interacting with Configurators. FM configuration is typically supported by tools denoted as configurators. These tools are based on a formal knowledge representation such as constraint satisfaction problems (CSP) or Boolean satisfiability problems (SAT) supported by corresponding reasoning engines. In addition to the identification of a solution, some scenarios require optimization functionalities, for example, minimizing the overall price of a configuration. Concepts supporting interactive configuration scenarios are discussed in Chapter 4. In FM configuration, diagnosis algorithms are needed, for example, to support the minimization of configurations (only relevant components should be included) or the identification of repair actions to find ways out of the *no solution could be found* dilemma. Also in this context, alternative repairs can be ranked which may require the integration of machine learning, more specifically recommendation concepts, that help to identify the most relevant trade-offs, i.e., trade-offs with a high probability of being accepted by the user. Finally, in situations where users are not sure about the inclusion of specific features, recommendation techniques can help to support the user in terms of recommending reasonable inclusions or exclusions. Related techniques and detailed examples are provided in Chapter 4.

Finally, in Chapter 5 we discuss the practical relevance of the FM-related tasks summarized in Table 1.1 by providing and discussing links to different FM based tools and applications.

1.4 Topics Related to Feature Models

There are a couple of topics with a direct relation to feature modelling related research. These topics will be touched in upon in one way or another in this book.

Software Product Line Engineering focuses on the development of a software codebase (with an emphasis on reuse) that represents a family of related products with variabilities and commonalities [2, 3, 18, 49, 56].

Knowledge-based Configuration has overlaps with FMs [39] regarding research topics and research results [26, 45, 55, 58]. *Knowledge representation & reasoning* [63], *explainable AI* [21], and *machine learning* [44] are AI research fields that play an increasingly important role in configuration tasks. Our discussion focuses on the application of these research fields in various feature modelling related tasks.

Recommender Systems [23, 28, 52, 62] support the identification of user-relevant items from large assortments defined, for example, by product catalogs or configuration knowledge bases. These systems combine different AI techniques such as machine learning [44] and explanations [21] to provide a personalized user experience when being confronted with complex item spaces.

Mass Customization is the production of highly-variant products and services under mass production pricing conditions [24, 37]. Software product lines extend the application of the mass customization paradigm to the area of software engineering with similar related tasks and research issues. Intangibility, high complexity, and a higher degree of adaptability (compared to physical products) make related management and implementation tasks even more demanding. A phenomenon related to mass customization is *mass confusion* [36] referring to cognitive overloads of customers triggered by a high number of configuration choices. Different machine learning concepts such as recommender systems that can help to tackle the challenges induced by mass confusion are discussed in this book.

Human Decision Making. This aspect is highly relevant specifically in the context of FM configuration. Knowledge about how humans decide, which basic types of shortcuts are used in human decision making, and in which way humans prefer to state their preferences must be taken into account when developing configurator user interfaces [4, 60]. Importantly, decisions are often made in groups, for example, in the context of product line scoping [41]. Group members need to decide about which features and constraints need to be included in a feature model, i.e., decide about very specific variability properties. In this context, group decision support is needed to support the group in finding a good solution [25, 40, 59].

1.5 Benefits of Feature Models and Configuration

Feature models (FMs) are key enabling technologies for supporting variability management in software development (and other types of tasks such as the configuration of physical products). The challenges of variability management and related benefits of FMs and configuration technologies can be summarized as follows.

Efficient variability model development & configuration. FMs are in many cases based on a graphical representation understandable for technical experts (e.g., configurator developers) as well as domain experts (e.g., product development) which is of specific relevance in early stages of a software development process. For this reason (both parties are able to "speak" the same language), the so-called knowledge acquisition bottleneck can be reduced in terms of lower communication overheads between developers and domain experts. For the same reason, domain know-how can be increased resulting in a kind of *corporate variability knowledge memory* [33]. Due to the systematic representation of software variabilities (using FMs), corresponding configurations can be derived in an efficient fashion helping to reduce software development efforts and the corresponding time to market [16, 61].

Avoiding erroneous and suboptimal FM configurations. FMs represent the variability properties of the underlying software product line (and beyond). When reusing and integrating individual software components, it is extremely important to assure the correctness of configurations, i.e., we want to avoid situations where incompatible software features result in faulty or at least low-performance behavior when being installed on the customer site. Beyond promoting correctness, FMs can also help to reduce *lead-times*, since configuration processes and configuration completion can be automated, i.e., are not a manual and time-consuming process anymore.

Understanding the configuration space (and its set of possible solutions). Knowledge about the configuration space can be of help to figure out weaknesses in terms of configurations leading to low system performance and to configurations assuring stable runtime performance [48]. Furthermore, configuration space understanding can help to better understand potential impacts of the supported configuration space on corresponding sales and production processes – this holds for physical products as well as reusable software components.

Efficient testing. FMs represent the software configuration space of software product lines and can also be used to support the systematic generation of test cases, for example, to achieve specific test coverage criteria [57]. Since FMs can easily be translated into a corresponding formal representation, basic FM properties can be easily analyzed, for example, *if every feature is part of at least one configuration*.

Avoiding mass confusion. When configuring complex items, it cannot be guaranteed that users know in detail every offered feature. In some cases, for example, when selling software services online, a cognitive overload can lead to situations where users (customers) refrain from taking a purchase decision. In such contexts, FM configuration (often combined with corresponding personalization services) can support users in identifying and also explaining the most relevant configurations. For the company itself, FMs can be regarded as a kind of *corporate memory* assuring the explicit representation of the variability properties of the offered software (as well as products and services).

Using a common language in different domains. Although there are many different FM dialects, most of them share a common way of expressing commonalities and variabilities. The same language can be used in different application domains which facilitates engineering activities. There is a community effort to develop a *Universal Variability Language* (UVL) [10] (see Chapter 2) to encourage knowledge sharing while promoting open science principles.

1.6 Book Overview

The remainder of this book is organized as follows:

- Chapter 2 introduces FM modelling languages with corresponding semantics using constraint satisfaction problems (CSPs) and Boolean satisfiability (SAT) problems. Specifically, CSP semantics are used as a basis for the discussions in the follow-up Chapters 3 – 5.
- Chapter 3 focuses on an in-depth discussion of FM analysis operations and concepts that help to assure FM model quality thus providing support in different FM maintenance tasks.
- Chapter 4 presents personalization approaches in FM-related processes. In particular, we analyze possibilities of integrating machine learning (specifically, recommender systems) into FM configuration.
- Finally, Chapter 5 discusses different aspects of applying the techniques introduced in Chapters 2 – 4 in tools and configurator applications.

References

1. M. Abbas, R. Jongeling, C. Lindskog, E. Enoiu, M. Saadatmand, and D. Sundmark. Product Line Adoption in Industry: An Experience Report from the Railway Domain. In *24th ACM*

Conference on Systems and Software Product Line: Volume A - Volume A, pages 1–11. ACM, 2020.
2. S. Apel, D. Batory, C. Kästner, and G. Saake. *Software Product Lines*, pages 3–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
3. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented Software Product Lines*. Springer, 2016.
4. M. Atas, A. Felfernig, S. Polat-Erdeniz, A. Popescu, T. Tran, and M. Uta. Towards Psychology-Aware Preference Construction in Recommender Systems: Overview and Research Issues. *J. Intell. Inf. Syst.*, 57(3):467–489, 2021.
5. V. Barker, D. O'Connor, J. Bachant, and E. Soloway. Expert Systems for Configuration at Digital: XCON and Beyond. *Commun. ACM*, 32(3):298–318, mar 1989.
6. D. Batory. Feature Models, Grammars, and Propositional Formulas. In H. Obbink and K. Pohl, editors, *International Conference on Software Product Lines*, pages 7–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
7. D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006.
8. D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank. Automated analysis in feature modelling and product configuration. In J. Favaro and M. Morisio, editors, *Safe and Secure Software Reuse*, pages 160–175, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
9. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615 – 636, 2010.
10. D. Benavides, C. Sundermann, S. Vill, K. Feichtinger, , J. A. Galindo, R. Rabiser, and T. Thüm. UVL: Feature Modelling with the Universal Variability Language. Technical report, Elsevier, 2024.
11. D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.
12. T. Bench-Capon and P. E. Dunne. Argumentation in Artificial Intelligence. *Artificial Intelligence*, 171(10):619–641, 2007. Argumentation in Artificial Intelligence.
13. T. Berger, D. Nair, R. Rublack, J. Atlee, K. Czarnecki, and A. Wasowski. Three Cases of Feature-Based Variability Modeling in Industry. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 302–319, Cham, 2014. Springer International Publishing.
14. D. Beuche. Using pure: Variants across the product line lifecycle. In *20th International Systems and Software Product Line Conference*, pages 333–336, New York, NY, USA, 2016. Association for Computing Machinery.
15. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.
16. L. Brownsword and P. Clements. A Case Study in Successful Product Line Development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.
17. M. Cashman, J. Firestone, M. Cohen, T. Thianniwet, and W. Niu. DNA as Features: Organic Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A*, pages 108–118. ACM, 2019.
18. A. E. Chacón-Luna, A. M. Gutiérrez, J. A. Galindo, and D. Benavides. Empirical software product line engineering: a systematic literature review. *Information and Software Technology*, 128:106389, 2020.
19. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and Their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
20. C. Dumitrescu, R. Mazo, C. Salinesi, and A. Dauron. Bridging the Gap between Product Lines and Systems Engineering: An Experience in Variability Management for Automotive Model Based Systems Engineering. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, page 254–263, New York, NY, USA, 2013. Association for Computing Machinery.

21. R. Dwivedi, D. Dave, H. Naik, S. Singhal, R. Omer, P. Patel, B. Qian, Z. Wen, T. Shah, G. Morgan, and R. Ranjan. Explainable AI (XAI): Core Ideas, Techniques, and Solutions. *ACM Comput. Surv.*, 55(9), jan 2023.
22. M. Eggert, K. Günther, J. Maletschek, A. Maxiniuc, and A. Mann-Wahrenberg. In Three Steps to Software Product Lines: A Practical Example from the Automotive Industry. In *26th ACM International Systems and Software Product Line Conference - Volume A*, page 170–177. ACM, 2022.
23. A. Falkner, A. Felfernig, and A. Haag. Recommendation Technologies for Configurable Products. *AI Magazine*, 32(3):99–108, 2011.
24. A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Transactions on Engineering Management*, 54(1):41–56, 2007.
25. A. Felfernig, L. Boratto, M. Stettinger, and M. Tkalcic. *Group Recommender Systems: An Introduction*. Springer Publishing Company Inc., 2nd edition, 2024.
26. A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen. *Knowledge-based Configuration – From Research to Business Cases*. Morgan Kaufmann, 2014.
27. A. Felfernig, V. Le, A. Popescu, M. Uta, T. Tran, and M. Atas. An Overview of Recommender Systems and Machine Learning in Feature Modeling and Configuration. In *15th Intl. Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '21. ACM, 2021.
28. A. Felfernig, M. Wundara, T. Tran, V.-M. Le, S. Lubos, and S. Polat-Erdeniz. Sports recommender systems: Overview and research issues. *Journal of Intelligent Inf. Sys.*, 2024.
29. J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, 101(5):387–433, 2019.
30. F. Hayes-Roth. Rule-based systems. *Commun. ACM*, 28(9):921–932, 1985.
31. R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering*, 27(2):1–34, 2022.
32. J. Hofer and M. B. A. Schäfer. Behavioral Customization of State Machine Models at ESO. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 188–198, New York, NY, USA, 2022. ACM.
33. P. Hofman, T. Stenzel, T. Pohley, M. Kircher, and A. Bermann. Domain Specific Feature Modeling for Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, pages 229–238. ACM, 2012.
34. A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4), 2021.
35. J. Horcas, M. Pinto, and L. Fuentes. Empirical Analysis of the Tool Support for Software Product Lines. *Software and Systems Modeling*, 22:377–414, 2023.
36. C. Huffman and B. Kahn. Variety for Sale: Mass Customization or Mass Confusion? *Journal of Retailing*, 74(4):491–513, 1998.
37. B. J. P. II. *Mass Customization: The new frontier in business competition*. Harvard Business, 1993.
38. U. Junker. QuickXPlain: Preferred Explanations and Relaxations for over-Constrained Problems. In *AAAI'04*, page 167–172. AAAI Press, 2004.
39. K. Kang, C. Sholom, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
40. V. Le, T. Tran, and A. Felfernig. Consistency-Based Integration of Multi-Stakeholder Recommender Systems with Feature Model Configuration. In *26th ACM International Systems and Software Product Line Conference*, pages 178–182, New York, NY, USA, 2022. ACM.
41. L. Marchezan, E. Rodrigues, W. K. G. Assunção, M. Bernardino, F. Basso, and J. Carbonell. Software Product Line Scoping: A Systematic Literature Review. *Journal of Systems and Software*, 186:111189, 2022.

42. M. D. McIlroy, J. Buxton, P. Naur, and B. Randell. Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.
43. J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.
44. T. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997.
45. S. Mittal and F. Frayman. Towards a Generic Model of Configuraton Tasks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'89, page 1395–1401, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
46. V. Myllärniemi, J. Tiihonen, M. Raatikainen, and A. Felfernig. Using Answer Set Programming for Feature Model Representation and Configuration. In *ConfWS'14*, pages 1–8, 2014.
47. E. OliveiraJr and D. Benavides. Principles of software product lines. In *UML-Based Software Product Line Engineering with SMarty*, pages 3–26. Springer, 2022.
48. J. Pereira, M. Acher, H. Martin, J. Jézéquel, G. Botterweck, and A. Ventresque. Learning Software Configuration Spaces: A Systematic Literature Review. *Journal of Systems and Software*, 182:111044, 2021.
49. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering*, volume 10. Springer, 2005.
50. A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V. Le, K. Pilsl, M. Enzelsberger, and T. Tran. An Overview of Machine Learning Techniques in Constraint Solving. *Journal of Intelligent Inf. Sys.*, 58(1):91–118, 2022.
51. R. Reiter. A Theory of Diagnosis From First Principles. *AI Journal*, 32(1):57–95, 1987.
52. J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, and D. Benavides. Selection of Software Product Line Implementation Components Using Recommender Systems: An Application to Wordpress. *IEEE Access*, 7:69226–69245, 2019.
53. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. ISSN. Elsevier Science, 2006.
54. V. Rothberg, N. Dintzner, A. Ziegler, and D. Lohmann. Feature Models in Linux: From Symbols to Semantics. In *VaMoS '16*, pages 65–72. ACM, 2016.
55. D. Sabin and R. Weigel. Product Configuration Frameworks – A Survey. *IEEE Intelligent Systems and their Applications*, 13(4):42–49, 1998.
56. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14:477–495, 2012.
57. S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools: a test suite. *IET software*, 5(1):70–82, 2011.
58. M. Stumptner. An Overview of Knowledge-Based Configuration. *AICom*, 10(2):111–125, 1997.
59. T. Tran, A. Felfernig, and V. Le. An overview of consensus models for group decision-making and group recommender systems. *User Model User-Adap Inter*, 2023.
60. T. Tran, A. Felfernig, and N. Tintarev. Humanized Recommender Systems: State-of-the-Art and Research Issues. *ACM Trans. Interact. Intell. Syst.*, 11(2), 2021.
61. J. Trasobares, A. Domingo, L. Arcega, and C. Cetina. Evaluating the Benefits of Software Product Lines in Game Software Engineering. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 120–130. ACM, 2022.
62. M. Uta, A. Felfernig, V. Le, T. Tran, D. Garber, S. Lubos, and T. Burgstaller. Knowledge-based Recommender Systems: Overview and Research Directions. *Frontiers in Big Data*, 7:1–30, 2024.
63. F. van Harmelen, V. Lifschitz, and B. Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, CA, USA, 2007.
64. L. Wozniak and P. Clements. How Automotive Engineering is Taking Product Line Engineering to the Extreme. In *19th International Conference on Software Product Lines*, pages 327–336, New York, NY, USA, 2015. ACM.

Chapter 2
Feature Modelling

Abstract In this chapter, we describe the basis of *Feature Models* (FMs) using graphical as well as textual representations. We introduce a *smartwatch* FM that will be used as a working example for this and later chapters. Based on this example, we describe feature modelling extensions using cardinalities and attributes. In the following, we show how FMs can be translated into a formal representation (constraint satisfaction problems and SAT problems) and introduce corresponding definitions of a *FM configuration task* and a corresponding *FM configuration* (also known as configuration, product, or solution). Finally, we discuss example machine learning (ML) approaches that can be applied in the context of feature modelling tasks.

2.1 Features, Products, and Configurations

A natural way of describing any product is in terms of features. A *feature* is an increment in product functionality [9, 11, 10]. If I want a Chinese wok, I have to decide whether I want rice or noodles; duck or prawns or both; or, if I want a very spicy sauce. Similarly, if I want a smartwatch, I have to decide on the list of features I want. I may want to have sport tracking support or a concrete screen type or maybe I want that my watch allows me to pay in shops. Those are all examples of features. Similarly, in software engineering, a product is not described in terms of technical details about the way a product is developed (e.g., what specific object-oriented pattern was used to develop part of a package), it is described to the general audience in terms of features. In systems and software engineering, the size of features is arbitrary, i.e., a feature can be of any size depending on the level of abstraction [10]. For example, a feature can be a set of classes and methods but a feature can also be described at the level of architectural elements depending on the granularity of the scope of the given product line. In this book, we will consider a feature at any level of abstraction, i.e., the concepts, tools, and processes discussed in the book can be adapted to the needed abstraction level.

A. Felfernig et al., *Feature Models*, SpringerBriefs in Computer Science,
https://doi.org/10.1007/978-3-031-61874-1_2

There are several definitions of what a *feature* is [20] – from more abstract to more technical ones (see below). For example, a very general and abstract definition is given by Kang et al. [45] in the seminal work about feature modelling in 1990. In contrast, Apel et al. [5] provide a definition more focusing on technical aspects.

"***A prominent or distinctive user–visible aspect, quality, or characteristic of a software system or systems***" – Kang et al. [45].

"***A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option***" – Apel et al. [5].

In this book, we consider a feature as *an increment in (program/product) functionality*. A product could be software, hardware, or both.

A complete list of features describes a configuration of a product. There are some features that are *implicit* to a product, this is, some features that cannot be decided or selected by the user, while there are some features that can be decided by the user. For instance, I can decide if I want a sweet sauce in a Chinese wok but I may not be able to decide the concrete kind of rice flour in the case I select a wok with rice noodles. Similarly, I may be able to decide if I want to have a concrete screen size but not the specific sport tracking technology in the case I decide to have that feature. This differentiation is often referred as *internal* or *external* variability [65]. The external variability is the one of domain artefacts or assets that are visible for customers or stakeholders while the internal variability is the one that is handled internally by the organization and is not visible for the external stakeholders.

Some feature combinations are allowed while some others are forbidden. The allowed feature combinations are determined by a model representing features and constraints among them. For instance, I can decide to have a wok with rice but then I cannot have noodles. The ingredients constraints will be determined by the wok menu. Similarly, I can decide to have a standard screen in my smartwatch but then I will not be able to have the payment feature. The allowed feature combinations (a.k.a. configurations) are defined by an FM.

The process of selecting and deselecting features when customizing (configuring) a product is known as *configuration process*. The final result of a configuration process is a *configuration* that can be a *complete configuration* (a.k.a. *full configuration*) if all the decisions regarding feature inclusion or exclusion were made during the configuration process or a *partial configuration* in the case some decisions were not made and not all the features were selected or deselected. In the former example, one can decide to get a noodles wok but not really being sure to have fish or meat that would make a partial configuration of a wok product; or there can be a full description of the wok with all the features selected or deselected that would make a complete configuration of a wok product. Similarly, I can be sure that I want a smartwatch including sports tracking but unsure about the tracking type.

The number of allowed configurations (a.k.a. *configuration space*) grows with the number of features. If a model has n optional features with no constraints, it could have 2^n distinct configurations. Let's stop here for a moment to understand the complexity of the problem that we can face when configuring products with many options. It is estimated that the observable universe has around 10^{80} atoms. This is a very large number. If we take a highly complex configuration system example, the LINUX kernel, we find that the configuration options have been growing in the past and it is easy to imagine that they will keep on increasing in the future. Kernel versions 5.0, 4.0 and 3.0 have around 16,000, 14,000 and 11,000 configuration options respectively [47]. If those configuration options were only Boolean (i.e., the configuration option can be set or not but there are not more than these two options), which is not always the case, the number of potential configuration options of the LINUX kernel would be in the order of 10^{4816}, i.e., we would need around $10^{4,736}$ universes to store all the configurations of the LINUX kernel if a configuration could be stored in only one atom – these numbers are huge. In those cases, we often talk about *colossal configuration spaces* [42, 62].

An *FM configurator* is a tool that allows configuring an FM (during application engineering, c.f. Figure 2.2) such that a product can be produced. Such a tool has as input the FM, and permits stakeholders to define the product they want by selecting (including) or deselecting (excluding) features. A configurator verifies that the feature selection is legal or not. In our *wok* example, the real-world counterpart of the configurator is represented by a combination of wok menu, waiter, and the cook who will ensure that the selected wok features are allowed before starting the "production" process. A configurator for our *smartwatch* example FM (see Figure 2.3) can be a software tool that offers a web interface with options to select and deselect features and taking into account domain and application constraints. We will see a related example in the next sections.

Figure 2.1 provides an overview of feature modelling related activities discussed in this chapter. First, the goal of *FM design* is to build an FM as a basis for follow-up configuration activities. FM design can be supported by (1) *FM (product line) scoping* which helps to identify those features which can be regarded as relevant and should be taken into account in the product line, (2) *configuration space learning* which helps to identify basic solver search heuristics for making search processes efficient, and (3) *knowledge extraction from data* which helps to identify FMs or parts thereof in an automated or semi-automated fashion (e.g., the automated identification of features from requirements specifications). For the purpose of enabling FM configuration, FMs have to be translated into a corresponding logic-based representation, for example, as a constraint satisfaction problem (CSP) or SAT problem. All of these aspects will be discussed in detail in the follow-up sections.

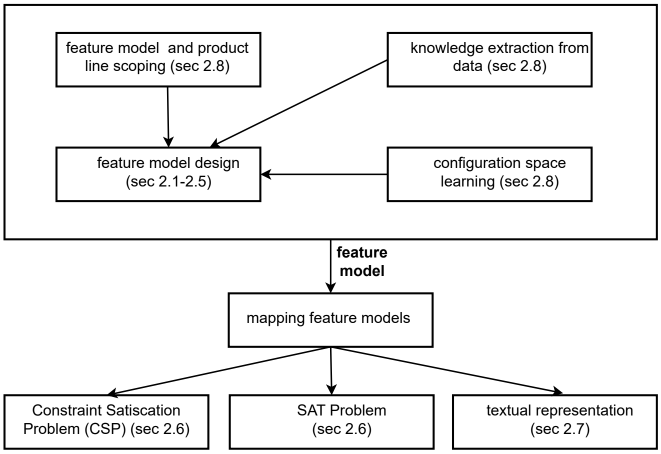

feature model and product line scoping (sec 2.8)
knowledge extraction from data (sec 2.8)
feature model design (sec 2.1-2.5)
configuration space learning (sec 2.8)
feature model
mapping feature models
Constraint Satiscation Problem (CSP) (sec 2.6)
SAT Problem (sec 2.6)
textual representation (sec 2.7)

Fig. 2.1: Feature modelling related activities and mapping to a corresponding logical representation (e.g., as a constraint satisfaction problem (CSP) or SAT problem) or textual representation for knowledge exchange purposes (ids in brackets refer to the corresponding subsection).

2.2 Feature Modelling in the Engineering Process

Feature modelling is used as a pivotal part in software product line engineering [4, 63] and can be applied to different contexts. There are several proposals for engineering software product lines. In this book, we follow a simplified and practical process proposal described by Apel et al. [4]. We distinguish four main activities that are shown in Figure 2.2. Software product line engineering activities are associated with two different dimensions. The vertical dimension distinguishes between *domain engineering* and *application engineering* (upper and lower part of Figure 2.2). The horizontal dimension distinguishes between *problem space* and *solution space* (left and right part of Figure 2.2).

Domain engineering develops reusable assets but not final products and has two different sub-processes: *domain analysis* (in the problem space) and *domain implementation* (in the solution space). Domain analysis identifies features in the scope of the product line and produces an FM that represents the allowed feature combinations. Domain implementation is the development of reusable assets to be used in application engineering.

Imagine a product line of wok dishes, during domain analysis, the first step would be to identify the ingredients and choices that we want to offer in the menu. Also, the constraints among these elements have to be identified. Similarly, during domain implementation, some pre-cooking of ingredients and preparation can be done to be reused later during application engineering. Components, platforms, APIs, libraries, documents, test cases, and in general any artefact that can be reused later in the production process are outputs of the domain engineering process. Most of the assets are common to all the products and this is why product line engineering is a good approach when there are commonalities among the products in a concrete domain. A central artefact in the domain engineering process is the FM.[1] There is a mapping between features in the FM and implemented artefacts in the solution space.

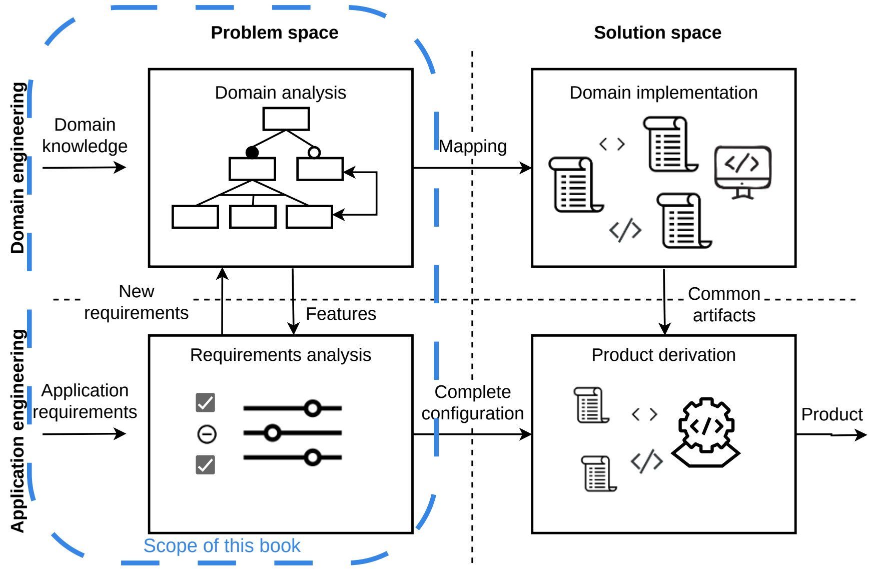


Fig. 2.2: Software product line engineering process based on [4]. This book focuses on *domain analysis* (feature modelling) and *requirements analysis* (configuration).

Application engineering produces a product based on a set of feature inclusions and exclusions (a.k.a. FM configuration) defined by an FM configurator. Reusable elements developed in domain engineering are used together with specific needs to produce a concrete product. An FM configurator is built using the FM that was designed during domain engineering and provides a user interface for interacting during the feature selection process. The output of this process is an FM configuration. During application engineering, there are two sub-processes: *requirement analysis* and *product derivation*. In this context, requirement analysis identifies the application requirements taking into account the user needs. For that, an FM configurator helps with the correct inclusion and exclusion of features in an step-wise process. Some new requirements can affect the domain analysis process when new features can be added, changed, or removed. Product derivation takes an FM configuration and corresponding (implemented) reusable artefacts as input and assembles a product that conforms with the application requirements and fulfills user needs.

[1] Other variability modelling approaches have been also proposed in the literature such as OVM, CVL, COVAMOF, decision modelling, and others [16, 28].

In the wok example, application engineering would be the process of selecting items with the help of the menu and the waiter as well as the process of preparing the dish and deliver it to the customer.

The *problem space* is distinguished from the *solution space* (left and right side of Figure 2.2). The problem space takes the perspective and vision of the external stakeholders, the context restrictions and, in general, the domain knowledge. In contrast, the solution space considers the perspective of internal stakeholders such as managers, developers and testers. The problem space is the *what*, while the solution space is the *how*. *What* features to offer and *what* products to build versus *how* features are implemented and *how* products are built.

In this book, *we concentrate on the left hand side of the process* (i.e., the problem space) where feature modelling plays a major role. There are different alternatives for implementing features and producing products from existing features that range from templates and `#if...#elsif...#endif` conditional compilation directives to modules of feature-oriented programming. For details on the solution space process, we refer to other books such as the one of Apel et al. [4] or Meinicke et al. [57]. In the problem space, features are the key concepts to organize the domain knowledge.

In the following, the main concepts of feature modelling are defined. In Chapter 3, FM analysis is explained (mostly used in domain analysis) and in Chapter 4, the FM configuration process will be discussed (mostly used in requirements analysis).

2.3 Feature Model Basics

The term "*feature model*" was coined by Kang et al. in the FODA report back in 1990 [45]. Feature modelling has been one of the main lines of research in software product lines since then [39]. There are different FM languages [13, 76]. We review the most well known dialects for those languages. In general, there is no FM language that will fit all scenarios and often, some concrete adaptations have to be done [3].

An FM is a compact representation of all possible configurations of a product line. FMs are widely used in software product line engineering but they can also be used in other domains such as video encoding [3], security information [56], biological information [18] or representing exam options [50] just to mention a few diverse examples. The holy grail of the SPL community is the Linux operating system FM which has thousands of modules and configuration options called options.

Figure 2.3 shows our running example of a *smartwatch* product line encoded using a common FM notation. An FM is composed of:

- A hierarchically arranged set of features (a.k.a. *feature diagram* or *feature tree*) that are encoded using relationships between a parent feature and its child features.
- A separate list of cross–tree constraints, typically inclusion or exclusion statements in the form of: *if feature X is included, then features Y and Z must also be included (or excluded)* but can include more complex constraints in the form of arbitrary propositional formulae.

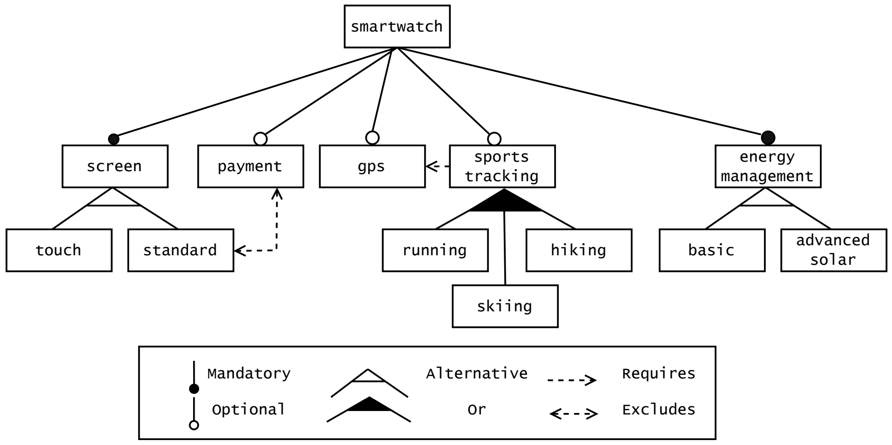


Fig. 2.3: Example *smartwatch* FM used in the book.

Classical FMs. A feature diagram of a classical FM declares four relationships:

- **Mandatory**. A child feature has a *mandatory* relationship with its parent if the child is part of all configurations in which its parent feature is included. In Figure 2.3, every *smartwatch* must provide a *screen*.
- **Optional**. A child feature has an *optional* relationship with its parent if the child can optionally be part of a configuration in which its parent feature is included. A *smartwatch* may optionally include *sportstracking*.
- **Alternative**. Child features have an *alternative* relationship with their parent feature if exactly one child is part of a configuration in which its parent feature is included. The *screen* of the *smartwatch* must be either *touch* or *standard* (but not both) in every configuration; and
- **Or**. Child features have an *or* relationship with their parent if one or more children are part of all the configurations that include the parent. Whenever *sportstracking* is selected, *at least one* (possibly all or any combination) of *running*, *skiing*, *hiking* are selected.

The root feature is included in all configurations. A feature can only be included in a configuration if the parent feature is included. In addition to the tree–like relationships between features described above, an FM can also contain cross–tree constraints between features – basic ones are the following:

- **Requires**. If a feature *A requires* a feature *B*, the selection of *A* in a configuration implies the selection of B. A *smartwatch* that includes *sportstracking* must also include the *gps* feature.
- **Excludes**. If a feature *A excludes* a feature *B*, both cannot be included in the same configuration. The *payment* feature cannot be combined with a *standard* screen, i.e., *payment* and *standard* are incompatible.

More complex cross-tree relationships have been proposed later in the literature allowing constraints in the form of generic propositional formulas, e.g., "*A and B implies not C*" [9, 34].

Abstract and concrete features. In some cases, there is a distinction between *concrete* and *abstract* features. Concrete features have a relationship with domain implementation artifacts in the solution space (c.f. Figure 2.2) while abstract features are only used for organization purposes and do not have any direct mapping to any artifact in the solution space. It is often recommended to only define the leaves of the tree as concrete features and let all the other intermediate features to be abstract ones [10]. For simplicity but without loss of generality, in this book, we will not distinguish between concrete and abstract features but will consider all as equal.

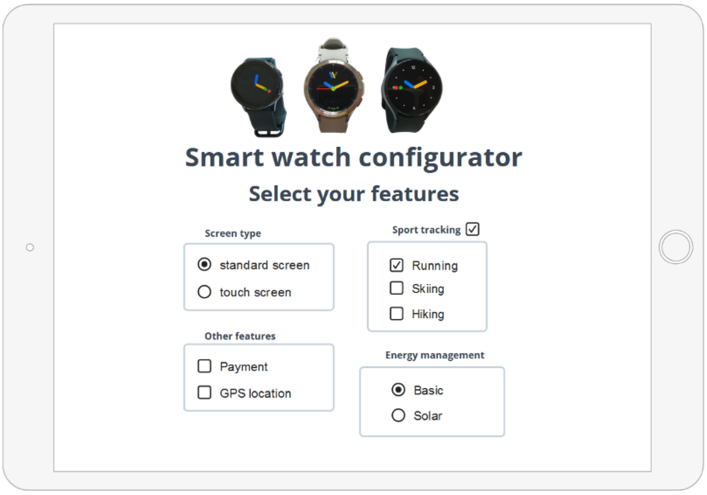
Smartwatch example. In the example of Figure 2.3, all smartwatches must include a *screen* (either *touch* or *standard*), and an *energy management* system (either *basic* or *advanced–solar*). Optionally, *payment*, *gps* and *sports tracking* features can be included. Furthermore, any combination of at least one out of the features *running*, *skiing*, and *hiking* can be included when the *sports tracking* feature is selected.

Table 2.1: Examples of *satisfiable* ($conf_1$) and *non-satisfiable* ($conf_2$) FM configurations (in $conf_2$, *payment* and *standard* cannot be included in the same configuration).

configuration	features included in configuration $conf_i$
$conf_1$	{smartwatch=true, screen=true, standard=true, energymanagement=true, basic=true}
$conf_2$	{smartwatch=true, screen=true, **standard=true**, **payment=true**, energymanagement=true, basic=true}

Table 2.1 shows satisfiable and non-satisfiable configurations $conf_i$ for our example FM (see Figure 2.3). Configuration $conf_1$ is satisfiable because it does not violate any of the FM constraints. In contrast, $conf_2$ is non-satisfiable because the *payment* feature is included and also the *standard* screen is. These features are incompatible due to the excludes constraint. Therefore, $conf_2$ is non-satisfiable.

From FMs, configuration tools (a.k.a. *FM configurators*) are constructed (see Chapter 5). An FM configurator is a tool to select and deselect features interactively while checking the consistency, i.e., either not allowing non-satisfiable configurations or alerting about the potential inconsistency. A possible user interface for configuring smartwatches is the one of Figure 2.4. There are groups of features that can be configured with selection and deselection of features. The FM configurator has to take care to only allow satisfiable configurations and advice the user in the case any misconfiguration is produced. For that, analysis and interaction with FMs are needed and those are the topics that will be addressed in Chapters 3 and 4.



Smart watch configurator

Select your features

Screen type
- ● standard screen
- ○ touch screen

Sport tracking ☑
- ☑ Running
- ☐ Skiing
- ☐ Hiking

Other features
- ☐ Payment
- ☐ GPS location

Energy management
- ● Basic
- ○ Solar

Fig. 2.4: Example *smartwatch* FM configurator.

2.4 Feature Model Extensions

There are different proposals in the literature to extend or modify feature modelling with different FM constructs. The most well known families of extensions are *cardinality–based* and *attribute–based* FMs. These extensions include a discussion that has been in the community for a while regarding what are the semantics of feature cardinalities, cloning or attributes. In this book, we will not address those problems in detail and refer the reader to related work [21, 46, 58, 59, 67, 75]. In any case, all techniques presented in this book are agnostic with respect to the way FMs are defined. These techniques can be equally used to analyse or configure classical, cardinality–based or attribute–based FMs. In the following, we provide a short discussion of these extensions.

2.4.1 Cardinality–based Feature Models

Cardinality–based FMs incorporate *cardinalities*, which resemble those found in the *Unified Modelling Language* (UML) (see [23, 69]). The relationships introduced in cardinality–based feature modelling are the following [12, 13]:

- **Feature cardinality.** A feature cardinality is a sequence of intervals $[n..m]$ with n as lower bound and m as upper bound ($n \leq m$). Intervals describe the number of instances of the feature that can be part of a configuration. This relationship may be used as a generalization of the original mandatory ($[1, 1]$) and optional ($[0, 1]$) relationships defined in FODA (Section 2.3).

- **Group cardinality.** A group cardinality is an interval $\langle n..m \rangle$, with n being the lower and m the upper bound ($n \leq m$) limiting the number of child features that can be included in a configuration. An alternative relationship is equivalent to a $\langle 1..1 \rangle$ group cardinality. An or–relationship is equivalent to $\langle 1..N \rangle$, being N the number of features in the relationship.

Figure 2.5 shows an example of the smartwatch FM using a cardinality–based notation. This FM represents the same *configuration space* (i.e., it represents exactly the same set of configurations) as the one in Figure 2.3.

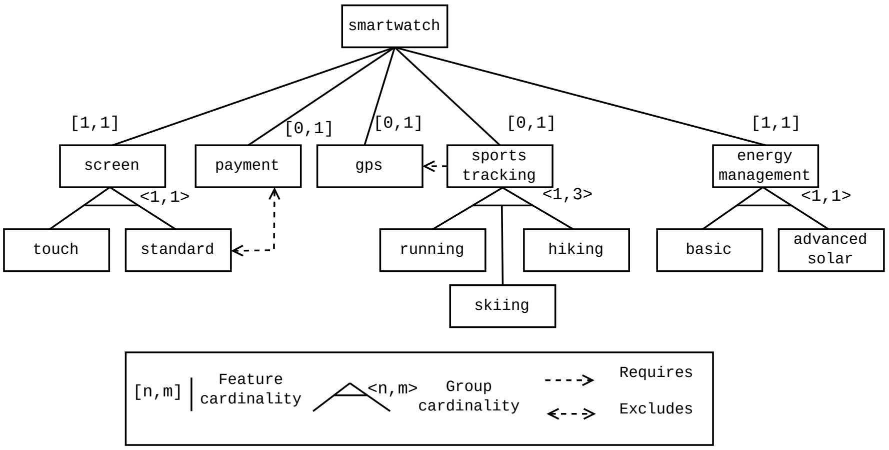


Fig. 2.5: Cardinality–based FM example.

2.4.2 Attribute–based Feature Models

To determine the cost or memory usage of a particular feature in a smartwatch configuration, *feature attributes* are needed. When FMs are expanded by including feature attributes, they are referred to as *extended, advanced, or attribute-based FMs*.

FODA [45], the seminal report on FMs, had a forward-thinking approach in considering the incorporation of more data into FMs. This involved introducing connections between features and their attributes, in addition to features and their relationships. Later, Kang et al. [44] made an explicit reference to what they call "non–functional" features related to feature attributes. There is no consensus on a graphical notation for attributes. However, most proposals agree that an attribute should consist at least of a *name*, a *domain*, and a *value*. Figure 2.6 depicts a sample FM including attributes using a notation inspired by Benavides et al. in [14]. As illustrated, attributes can be used to specify the price of a feature or the size of a concrete screen. Attribute–based FMs can also include complex constraints among attributes and features like: "*If attribute price of feature advanced solar is lower than a value X, then feature touch cannot be part of the configuration*". For instance, there can be a global constraint that specifies that the price of a smartwatch is calculated using the sum of the prices of the selected features. Similarly, there can also be customer constraints that specify an upper bound on the price of a smartwatch.

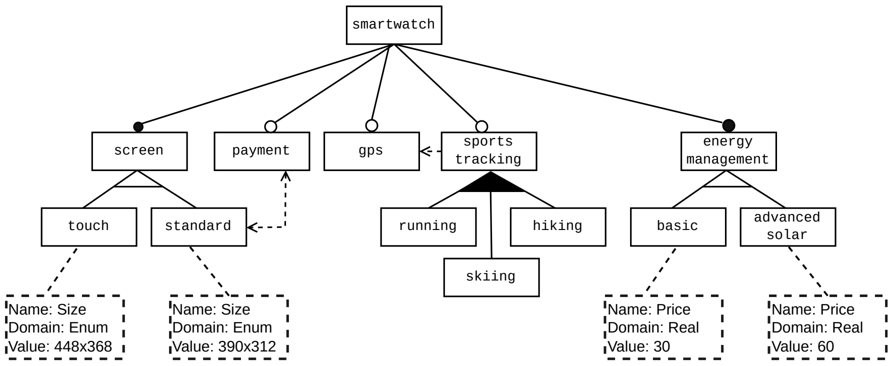


Fig. 2.6: Attribute–based FM example.

More advanced configurators can be built when using attributes, cardinalities, and complex constraints. In this book, for the sake of simplicity, we only use classical FMs as described in Section 2.3. However, all the described techniques can also be applied to other FM types.

2.5 Feature Model Semantics

To provide a semantics for FMs, the main concepts of previous sections are now defined formally. We use propositional logic in the form of a CSP (Constraint Satisfaction Problem) [6, 72]. An FM is composed of two main elements:

- A non-empty set of features that can be combined to form FM configurations
- A constraint model which determines the combinations of features that are satisfiable configurations of the FM

Definition 2.1 (Feature). A feature is the basic element of an FM and it is assigned a value in an *FM configuration*. Boolean features that are true ($\top$) are included; false ($\bot$) are excluded. Non-boolean features are possible (e.g., integers) but, for the sake of simplicity, we do not define them here.

Definition 2.2 (Set of all features). The set of all features in an FM is $F = \{f_1, f_2, ..., f_n\}$. Only the features in F can be part of the constraints of the *constraint model*.

Definition 2.3 (Constraint model). A constraint model CF of an FM is a set of constraints $CF = \mathcal{R} \cup \Pi$, where:

- $\mathcal{R}$ is the finite set of decompositional relationships between features that are mapped as a set of constraints over the set of features F, i.e., $\mathcal{R} \subseteq \mathbb{B}(F)$[2]
- Π is a set of cross-tree constraints defined as arbitrary propositional formulas over the set of features F, i.e., $\Pi \subseteq \mathbb{B}(F)$.

Definition 2.4 (Feature Model). A feature model (FM) is a tuple $(F{,}CF)$, where F is the set of all features and CF a constraint model that uses only the features in F. The semantic domain of the FM is determined by the constraints in CF and represents all the FM configurations (the FM configuration space).

Definition 2.5 (Application requirement). An FM *application requirement* is a set of constraints CR specifying specific preferences[3] of a stakeholder that have to be considered in an FM configuration, i.e., $CR = \{c_1..c_m\}$.

Definition 2.6 (FM Configuration). An *FM configuration* is an assignment $A = \{f_1 = v_{f1}..f_n = v_{fn}\}$ $(v_{fi} \in dom(f_i))$ on the features of an FM represented as variables $f_i \in F$. A is *satisfiable* if it does not violate any constraint in the FM and application requirements (i.e., it does not violate the set $CF \cup CR$ - the *consistency* property). An FM configuration is *complete* (a.k.a. full configuration), if every feature has an assignment describing an inclusion or exclusion and it is *partial* otherwise.

Definition 2.7 (FM configuration space). The set of all the complete and satisfiable FM configurations of an FM represents the *FM configuration space*. Therefore, all the configurations of the FM configuration space are satisfiable (i.e., they do not violate the set $CF \cup CR$).

Definition 2.8 (FM Configuration Task). An FM configuration task is a tuple (F, D, FMC) defined by a set $F = \{f_1, f_2, ..., f_n\}$ of features; corresponding domains for the features $D = \{dom(f_1), dom(f_2), ..., dom(f_n)\}$ (e.g., for classical FMs , $dom(f_i)$ =){true ($\top$), false ($\bot$)}); and a set of constraints $FMC = CF \cup CR$ restricting the set of possible solutions (CF) and a set of application requirements (CR) as defined previously. In this context, CF= $\{c_1..c_k\}$ and $CR = \{c_{k+1}..c_m\}$.

For an FM configuration task, a constraint solver can be activated to find a corresponding solution (FM configuration). More details on the inclusion of solvers are provided in Section 2.6 and Chapters 3 and 4.

The terms used in different research fields are similar but can lead to confusion. In this book, we defined an *FM configuration* in Definition 2.6 and an *FM Configuration Task* in Definition 2.8. In the literature, one can find related terms such as *product configuration*, *configuration*, *feature selection*, *feature combination*, *product*

[2] Let $\mathbb{B}$ denote the boolean domain, $\mathbb{B} = \{\text{true } (\top), \text{false } (\bot)\}$ and $\mathbb{B}(F)$ a function denoting all possible boolean constraints on the set of features F. Classical FMs use only boolean features.

[3] Also known as user, stakeholder, or customer requirements.

description, *product specification*, or *staged configuration* just to mention a few. To clarify these terms, we differentiate between *process* and *process result*.

FM configuration process: In this process, features are selected or deselected with the goal to find a satisfiable FM configuration for a given FM configuration task. This process can be initiated if the FM is stable and is performed during application engineering in the problem space dimension (see Figure 2.2). The FM configuration process is also referred as *product configuration* [57], *configuration process* [4, 57], *feature selection process* or less common *configuration setting*. During this process, the possibilities that are available to the user are often referred as *configuration options*, *configuration alternatives*, or *configuration attributes* [4, 29]. If the FM configuration process is performed in several stages, it is sometimes called *staged configuration* [24].

FM configuration: This is the result of an FM configuration process and – following Definition 2.6 – is the result of a selection and deselection of features. Alternative terms may be used for FM configuration, such as *product*, *feature selection*, *feature combination*, *product description*, *product specification*, *product configuration* or simply *configuration*. A product is often considered a complete FM configuration with a consistent feature selection. In this book, we will use the term *FM configuration* but also *configuration* for simplification purposes. Following Figure 2.2, we propose to use the term FM configuration because using the term *product* can be confusing since a product is produced after the product derivation process from a complete FM configuration.

There are other related terms that should not be confused. When an FM configuration has been created as a satisfiable set of included and excluded features (in the application engineering process), the product has finally to be produced in the solution space. This "production" is also denoted as *product derivation*, *product configuration* (in some software engineering contexts), *product generation* or *product assembly* [4]. The techniques used for product derivation are different to the ones described in this book. Among those techniques, there is one that can cause confusion: *configuration parameters* [4]. Among the most common options, configuration parameters are passed through command line, global variables, and values in a properties or requirements file. In this book, we will not use this term that can be controversial with *feature attributes* (see Section 2.4.2). Therefore, techniques, tools, and studies about configuration parameters are out of the scope of this book.

2.6 Mapping Feature Models to Logic

Up to now, we have introduced formal definitions and indicated that an FM has a constraint model. Depending on the FM constructs, different constraint models can be built. In the following, we define the mapping from FMs to logic using constraint programming and SAT solving.

2.6.1 Constraint programming mapping

A *Constraint Satisfaction Problem* (CSP) [6] consists of a set of variables, a set of finite domains for those variables, and a set of constraints restricting the values of the variables. *Constraint programming* is the set of techniques such as algorithms or heuristics that deal with CSPs. A CSP is solved by finding values for variables (a.k.a. states) in which all constraints are satisfied. CSP solvers can deal not only with binary values (true or false) but also with numerical values such as integers, intervals, and symbolic domains (e.g., smart watch price).

A CSP solver is a software package that takes a problem modelled as a CSP and determines whether there exists a solution for the problem. From a modelling point of view, CSP solvers provide a richer set of modelling elements in terms of variables (e.g. sets, finite integer domains, etc.) and constraints (not only propositional connectives) as it is the case with SAT solvers.

The mapping of an FM into a CSP can vary depending on the concrete solver. In general, the following steps are performed: (1) each feature of the FM maps to a variable of the CSP with a domain of 0..1 (or *false, true*), depending on the kind of variable supported by the solver, (2) each relationship of the model is mapped into a constraint depending on the type of relationship (in this step, some auxiliary variables can appear), (3) the resulting CSP is the one defined by the variables of steps (1) and (2) with the corresponding domains and an additional constraint assigning true to the variable that represents the root, i.e., $root \Leftrightarrow true$ or $root == 1$, depending on the variables' domains.

Concrete rules for translating an FM into a CSP using propositional logic are listed in Figure 2.7 (see also the original proposal of Benavides et al. [14]). Also, the mapping of our running example of Figure 2.3 is presented. A *propositional formula* consists of a set of primitive symbols or variables and a set of logical connectives constraining the values of the variables, e.g. $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. It is important to remark that the root feature is part of any product and that is why an extra constraint is added to represent this property. Note that $\oplus$ is used to denote that only one feature of the set can be selected. Depending on the solver, this operator may not be available and then a formula with other basic operators has to be built [12].

2.6.2 SAT based mapping

A *SAT solver* is a software tool that works with a propositional formula in order to determine if the formula is satisfiable, i.e., there is a variable assignment that makes the formula evaluate to true. Input formulas are usually specified in *Conjunctive Normal Form* (CNF) using formats such as DIMACS [17]. CNF is a standard form to represent propositional formulas that is used by most SAT solvers where only three connectives are allowed: $\neg, \wedge, \vee$, this is, the logical negation, logical conjunction and logical disjunction of formulas. It is well known, and was proved time ago, that every propositional formula can be encoded into an equivalent CNF formula.

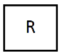
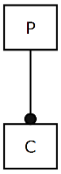
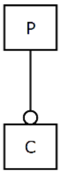
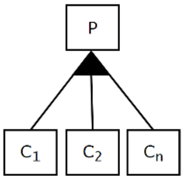
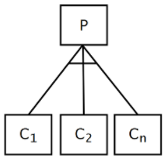
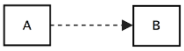
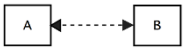
	Relationship	PL Mapping	Smartwatch Example
ROOT		R ⇔ TRUE	smartwatch ⇔ TRUE
MANDATORY		$P \Leftrightarrow C$	smartwatch ⇔ screen smartwatch ⇔ energy management
OPTIONAL		$C \Rightarrow P$	payment ⇒ smartwatch gps ⇒ smartwatch sportstracking ⇒ smartwatch
OR		$P \Leftrightarrow (C_1 \vee C_2 \vee ... \vee C_n)$	sportstracking ⇔ (running ∨ skiing ∨ hiking)
ALTERNATIVE		$P \Leftrightarrow \oplus(C_1, C_2, ..., C_n)$	energy management ⇔ basic ⊕ advanced solar screen ⇔ touch ⊕ standard
IMPLIES		$A \Rightarrow B$	sports tracking ⇒ gps
EXCLUDES		$\neg(A \wedge B)$	¬(payment ∧ standard)

Fig. 2.7: Mapping from an FM to a constraint satisfaction problem (CSP). In this context, the semantics of $\oplus(C_1, C_2, ..., C_n)$ is that exactly one of the features $C_1, C_2, ..., C_n$ must be included. For the sake of simplicity, we use the infix notation when only two features are involved.

Similarly, SAT is a well known NP-complete problem. Nevertheless, due to extensive research in the SAT solving area [37], there have been big advances in research and practice of SAT solving which makes it possible to address many practical problems with efficient computing resource usage [17].

The mapping of an FM into a propositional formula can change depending on the used solver. In general, the mapping is performed in the following steps: (1) each feature of the FM maps to a variable of the propositional formula, (2) each relationship of the model is mapped into one or more small formulas depending on the type of relationship, in this step some auxiliary variables can appear, (3) the resulting formula is the conjunction of all the resulting formulas of step (2) plus an additional constraint assigning true to the variable that represents the root. Rules for translating an FM into a propositional formula are listed in Figure 2.8. Further related work on specific – and potentially more efficient – SAT encodings can be found, for example, in Nguyen et al. [61] and Sinz [77].

There are other tools that also work with propositional formulas. One of those tools that is also used in FM analysis and configuration is BDD. A *Binary Decision Diagram* (BDD) solver is a software package that takes a propositional formula as input (it can be in CNF or not) and translates it into a graph representation (the BDD itself). With this data structure, it is very easy to determine whether the formula is satisfiable and there are efficient algorithms for counting the number of possible solutions [17]. The size of the BDD is crucial because it can grow exponentially in the worst case. Although it is possible to find a good variable ordering that reduces the size of the BDD, the problem of finding the best variable ordering remains NP-complete [42].

2.6.3 CSP example mapping

The FM example of Figure 2.3 can be represented as a CSP. Table 2.2 shows a CSP representing an FM configuration task (see Definition 2.8): F as a set of features, D as the corresponding domains, FMC as the union of the set of constraints CF of the FM and the constraints representing the application requirements CR of the smartwatch FM example (in this example representing that only configurations with gps are desired, but of course other combinations could be introduced). Note that $c_0 : smartwatch$ is a *root constraint* which prevents the generation of empty configurations, i.e., configurations where no feature is selected. Following the already introduced formalizations, we apply the logical operators of $\Rightarrow$ denoting an implication, $\Leftrightarrow$ denoting equivalence, $\vee$ denoting a logical *or*, $\wedge$ denoting a conjunction (logical *and*), and $\oplus$ denoting a logical *xor* indicating that only one feature can be selected from the given set, for example, $basic \oplus advancedsolar$ represents $\neg basic \wedge advancedsolar \vee basic \wedge \neg advancedsolar$.

Note that throughout the book, we follow the formatting rule that (1) CF, i.e., the set of constraints and relationships of the FM, is defined without the explicit usage of $\{true, false\}$, for example, we write $c_0 : smartwatch$ also meaning

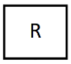
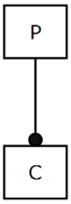
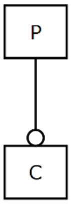
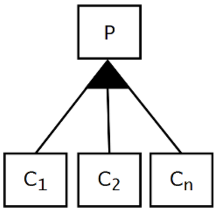
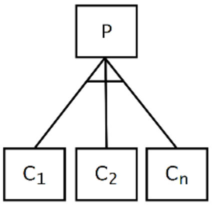
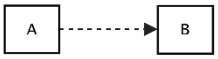
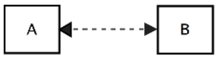
	Relationship	CNF Mapping
ROOT		R
MANDATORY		$(\neg C \vee P) \wedge (\neg P \vee C)$
OPTIONAL		$\neg C \vee P$
OR		$(\neg C_1 \vee P) \wedge (\neg C_2 \vee P) \wedge ... \wedge (\neg C_n \vee P) \wedge$ $(\neg P \vee C_1 \vee C_2 \vee ... \vee C_n)$
ALTERNATIVE		$(C_1 \vee .. \vee C_n \vee \neg P) \wedge$ $(\neg C_1 \vee \neg C_2) \wedge (\neg C_1 \vee \neg C_3) \wedge ... \wedge (\neg C_{n-1} \vee \neg C_n) \wedge$ $(\neg C_1 \vee P) \wedge ...(\neg C_n \vee P)$
IMPLIES		$\neg A \vee B$
EXCLUDES		$(\neg A \vee \neg B)$

Fig. 2.8: Mapping from FM to CNF (SAT solving context).

F =	{smartwatch, screen, touch, standard, payment, gps, sportstracking, running, skiing, hiking, energymanagement, basic, advancedsolar}
D =	{dom(smartwatch)={true, false}, dom(screen)={true, false}, dom(touch)={true, false}, .., dom(advancedsolar)={true, false}}
CF =	{ c_0: smartwatch, c_1: smartwatch ⇔ screen, c_2: payment ⇒ smartwatch, c_3: gps ⇒ smartwatch, c_4: sportstracking ⇒ smartwatch, c_5: smartwatch ⇔ energymanagent, c_6: sportstracking ⇔ (running ∨ skiing ∨ hiking), c_7: screen ⇔ touch ⊕ standard, c_8: energymanagement ⇔ basic ⊕ advancedsolar, c_9:¬(payment ∧ standard), c_{10}:sportstracking ⇒ gps }
CR =	{ c_{11}: gps=true }

Table 2.2: CSP mapping example.

$c_0 : smartwatch = true$. In contrast, for understandability reasons, the values $\{true, false\}$ are explicitly included when specifying customer requirements and FM configurations, for example, we write $c_{11} : gps = true$ also meaning $c_{11} : gps$ (an example of concrete FM configurations is given in Table 2.1).

2.7 Textual Languages for Feature Models

Representing FMs as diagrams has always been possible. Indeed, the original FODA report provided a first graphical representation of FMs that has little evolved in general. Graphical representations of FMs are usually known as *feature diagrams*. Most of the representations look similar to the ones presented in this chapter (see, e.g., Figure 2.3). There are others with different visual representations but basically, all express the same concepts.

In parallel, there has been a tendency to propose different textual representations of FMs [26]. There are different motivations to propose a textual variability model language [15]. Among those, exchanging models for allowing interoperability among tools as well as sharing among researchers and practitioners; teaching and learning using a common language that can be produced by programmers and displayed in any text editor; or allowing common analysis over the models with different tools.

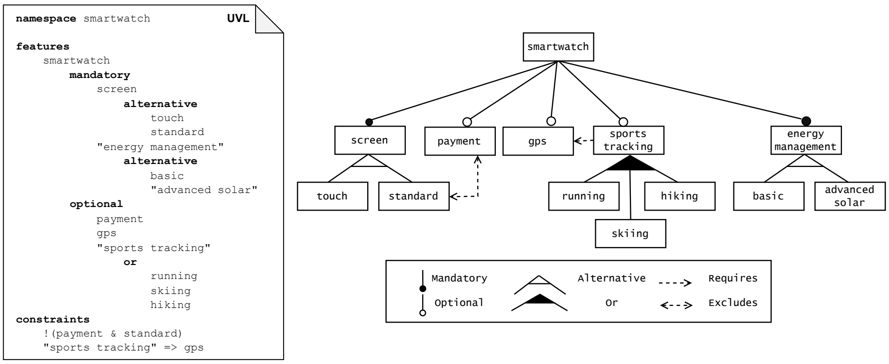
UVL. The goal of the MODEVAR initiative[4] is to create a common language for variability modelling. A proposal towards an Universal Variability Language (UVL[5]) is being pushed forward [13, 79]. UVL is a textual variability language that can express

[4] https://modevar.github.io/

[5] https://github.com/Universal-Variability-Language

basic FMs and has extension mechanisms to provide enriched versions to include, for instance, cardinalities, attributes, and types.

UVL utilizes a tree–like structure to represent the hierarchical nature of FMs and a tabular based identation to separate concepts. To illustrate this, Figure 2.9 shows the UVL representation of our running example of the smartwatch product line. UVL includes several key concepts for specifying constraints, including mandatory and optional as well as the “or” and “alternative” relationships. Finally, cross-tree constraints are supported, allowing any arbitrary propositional constraint involving various features.



```
namespace smartwatch                    UVL

features
    smartwatch
        mandatory
            screen
                alternative
                    touch
                    standard
            "energy management"
                alternative
                    basic
                    "advanced solar"
        optional
            payment
            gps
            "sports tracking"
                or
                    running
                    skiing
                    hiking
constraints
    !(payment & standard)
    "sports tracking" => gps
```

Fig. 2.9: UVL [13] FM example.

There exist other textual variability modelling languages [26]. Some of these are based on XML and others have their own syntax such as TVL [19]. There was even an attempt to standardize a variability modelling approach at the OMG level. The approach was called Common Variability Language (CVL) [41] but did not materialize as a real standard.

Recently, and still in the umbrella of the MODEVAR initiative, a repository of UVL models was released [13, 71].[6] This repository is designed using open science principles and allows the upload, search, and download of FM datasets. It is a central point to share FMs among practitioners and researchers using UVL.

Other textual constraint languages. In the constraint solving and configuration communities, there have been also efforts to propose textual languages for representing constraints or configuration problems. The motivations are similar. Among the most relevant proposals are DIMACS [38], Minizinc [86], or XCSP [8].

The DIMACS format [38] is commonly used to represent SAT instances that can be interpreted by different SAT solvers. This format uses plain text and includes a collection of clauses that are represented as a sequence of literals, which can be variables or negations. The DIMACS format is widely adopted for benchmarking SAT solvers and sharing SAT problems in research studies. DIMACS has a compact

[6] https://www.uvlhub.io/

syntax and it is not that easy to understand for humans since it is a plain text file comprising only numbers for Boolean variables which represent the features. Each row forms a disjunction of possibly negated variables which represents a structural or cross-tree constraint (see Section 2.6.2). Those simple syntactical rules make it easy to process DIMACS files by SAT solvers.

MiniZinc [86] is a textual constraint modelling language that is used to specify CSPs. MiniZinc is open-source and allows to describe a CSP in a textual fashion – CSPs formulated this way can be solved by different constraint solvers. It is designed to be solver–independent, enabling the user to switch between solvers easily. MiniZinc is used in various fields such as operations research, scheduling, planning and can be also used in product configuration. The syntax of MiniZinc is similar to a programming language and its intention is that it can be produced and edited by humans. An example screenshot of the MiniZinc IDE including a constraint-based representation of our example *smartwatch* FM is shown in Chapter 5.

XCSP (XML Constraint Satisfaction Problems) [8] is a textual language used for specifying instances of combinatorial problems. XCSP provides a unified representation of various types of problems, including CSPs, combinatorial optimization problems, and scheduling problems. The format is based on XML, making it possible to parse and process using standard software tools. XCSP supports a wide range of constraints, including global constraints, soft constraints, and constraints over finite domains or real numbers. The format has been adopted by some academic and industrial tools, and it is used for benchmarking, sharing, and comparing different constraint solvers. Being XML, the syntax is closer to a machine than a human user. Nevertheless, it is readable by humans, too. In the remainder of this book, for understandability reasons, we will use mostly the graphical representation of FMs – all examples can be easily translated to UVL and processed by UVL-compatible tools.

2.8 Further Feature Modelling Aspects

Up to now, we have focused on knowledge representation and formalization aspects in the context of feature modelling. Further related issues will be discussed in the following subsections. *First*, product line scoping [55] is related to the task of deciding which features should finally be included in the FM and – as a consequence – presented as an option to a configurator user. In this context, we will focus on different *explanation* aspects which play a major role in such decision contexts. *Second*, *configuration space learning* [64] is directly related to the task of feature modelling: FMs can be formalized, for example, as a CSP [14]. In order to assure search efficiency of the constraint solver, machine learning (ML) techniques can be used to learn solver search heuristics in a way that a configurator shows an acceptable runtime in most of the cases. *Third*, also in the context of designing FMs, machine learning techniques, for example, *knowledge extraction from data*, can help to automatically determine features or even constraints from textual requirements.

2.8.1 Product Line Scoping

Deciding which features and constraints to include in an FM can be regarded as a basic task in the context of different product (line) scoping scenarios [43, 52, 55, 70]. In such scenarios, basic machine learning and decision support techniques can be used to support stakeholders such as product owners, sales managers, and domain experts in their decision regarding the inclusion and exclusion of features in new versions of a product line. Specifically, recommendations need to be explained which is a task related to the field of explainable AI (XAI) [25].

Table 2.3 includes a simplified example of a decision scenario regarding the inclusion and exclusion of our example smartwatch features. In this example, individual stakeholders $s_i \in \{s_1..s_3\}$ vote for or against the inclusion of a specific feature where 1 indicates inclusion and 0 indicates exclusion. Such decisions about inclusion and exclusion of features can be interpreted as a basic optimization problem with the goal to minimize the number of adaptations needed such that overall consensus can be achieved regarding each individual feature [30]. Such an optimization could also take into account aspects such as fairness and unequal weights of individual stakeholders (e.g., experts vs. non-experts regarding a specific feature [7]).

Table 2.3: Simplified example of a recommendation support for the inclusion (=1) or exclusion (=0) of individual features in a product line (rec = recommendation with majority voting).

stakeholder	touch	standard	payment	gps	running	skiing	hiking	basic	advancedsolar
s_1	1	0	1	1	1	0	1	0	1
s_2	0	1	0	0	0	1	0	0	1
s_3	1	0	0	1	1	0	1	0	1
rec	1	0	0	1	1	0	1	0	1

In Table 2.3, the recommendation (*rec*) shown to the group $\{s_1, s_2, s_3\}$ is based on *majority voting* [31]. When analyzing the preferences of the individual stakeholders regarding feature inclusion, we can observe that the stakeholders s_1 and s_3 have a basic consensus regarding the inclusion and exclusion of individual features. There is one exception since s_3 does not seem to support the inclusion of the *payment* feature. There might be different reasons for this preference ranging from not understanding the feature to a more basic reason of not being aware of the feature importance. In any case, the recommendation system should not just recommend to exclude the feature but also recommend, for example, discussion between s_1 and s_3. Another related observation is that stakeholder s_2 has preferences which differ completely from those of s_1 and s_3 with one exception. One reason behind might be that s_2 has more expertise regarding the preferences of the underlying customer community and the market potential of individual features. Anyway, discussions have to be triggered among the individual stakeholders in order to achieve a consensus at the end [31, 49, 82].

2.8.2 Configuration Space Learning

An important issue specifically with large and complex variability models is to provide a means for assuring acceptable runtime performance of the underlying constraint solver or SAT solver [66]. Just translating the FM into corresponding sets of variables and constraints is not enough. We have to take care of selecting appropriate *search heuristics* that help to improve solver performance [84]. Recent developments in the fields of constraint solving and SAT solving aim to integrate machine learning for recommending search heuristics that will help to solve a problem instance (configuration task) efficiently – for an overview see Popescu et al. [66].

Similar to other complex domains, learning search heuristics requires the availability of corresponding datasets that can be used as an input for a (supervised) machine learning process. As such, the problem of learning search heuristics can be seen as a specific instance of *configuration space learning* [64] where different data synthesis approaches are used to generate relevant example problem instances which can then be used for optimizing a corresponding machine learning model [84]. In the following, we discuss a simplified example of a nearest neighbor (NN) based approach for recommending solver search heuristics. Table 2.4 depicts an example of a synthesized dataset. The underlying assumption is that *users only specify their preferences with regard to the inclusion of different* sportstracking *features*. The remaining features are directly selected by the constraint solver where *l* denotes the *lowest value first* search heuristic (i.e., *false* is selected before *true*) and *h* denotes the *highest value first* search heuristic (i.e., the solver selects *true* before *false*).

Table 2.4: Simplified example of machine learning based search heuristics recommendation. In this context, *h* and *l* denote search heuristics: *h=highest value first* and *l=lowest value first*. Furthermore, *id* represents the identifier of the corresponding dataset entry and *c* represents the preferences defined by the current user. For simplicity, we assume that users only specify preferences regarding the *sportstracking* features – configuration completion is then performed by the constraint solver. With *ms* we denote the *runtime in milliseconds* needed for configuration completion.

id	touch	standard	payment	gps	sportstracking	running	skiing	hiking	basic	advancedsolar	*ms*
1	l	h	l	l	true	false	true	false	l	h	12.2
2	l	h	l	l	true	false	true	false	h	l	44.2
3	h	l	h	h	true	true	true	true	h	l	34.5
c	?	?	?	?	true	?	true	false	?	?	?

The dataset includes three satisfiable configurations with the identifiers $\{1,2,3\}$. Furthermore, in our scenario the current customer c specifies his/her requirements regarding a smartwatch which are $sportstracking = true, skiing = true, hiking = false$. If we compare these preferences with the entries $\{1,2,3\}$ in Table 2.4, the entries with the most similar preferences, i.e., the nearest neighbors are $\{1,2\}$. Since the nearest neighbor with the id 1 has the better performance (in *ms*), the search heuristics of 1 would be recommended for the efficient determination (completion) of an FM configuration for the current user c.

Another application of machine learning is when a set of configurations are sampled from an FM to perform measurement of a given performance function (e.g. runtime) [62].[7] From a sample, machine learning techniques can be applied to predict the performance of a configuration without having to run all the FM configurations which is impractical in most of the cases. Recent advances [62] show that uniform random sampling is better finding *near–optimal* FM configurations than existing machine learning proposals.

2.8.3 Knowledge Extraction from Data

FMs can become quite large – see, for example, the Linux operating system FM [2, 81]. In such a context, it can be helpful to automatically identify feature candidates and also related constraints (see, e.g., [40, 51, 80]).

Basic Machine Learning Approaches. Feature candidates can be determined, for example, on the basis of content-based machine learning techniques (e.g., clustering) that allow an intelligent grouping of software requirements – see Li et al. [51]. Terms extracted from these requirements and associated with specific clusters (requirement groups) can be regarded as representatives of features. If features have already been determined, machine learning methods can be applied to determine related constraints [80]. The idea is to randomly generate configurations out of an FM (assuming that features are already available) and then to use an oracle (e.g., a software that tests the generated configuration) to figure out if the configuration is satisfiable (e.g., the software is operable). Following this approach, a dataset such as the one shown in Table 2.5 can be generated. With such a dataset, machine learning (e.g., decision tree learning) can be used to infer potential (need to be evaluated) FM constraints (see, e.g., Temple et al. [80]). If decision trees are used, FM constraints can be derived by interpreting "faulty" paths as negated constraints.

Table 2.5: Abstract dataset as a basis for machine learning based constraint extraction from configuration data. The oracle feedback can be 0 (faulty/non satisfiable configuration) and 1 (satisfiable configuration).

configuration	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	oracle feedback
$conf_1$	1	1	1	0	1	0	1	1	0
$conf_2$	1	0	1	0	1	0	0	1	1
$conf_3$	0	0	1	0	1	0	0	0	1
...	...	...	...	...	...	...	...	...	...
$conf_n$	1	1	0	0	1	1	0	0	0

[7] Predicting FM configuration performance can also be regarded as an analysis task (see Chapter 3).

Use of Large Language Models (LLMs). Large language models can help to increase the efficiency of FM development, for example, by automatically extracting features from given requirements specifications [40]. Table 2.6 depicts an example that shows in a simplified fashion how requirements can be generated and related features be extracted using an LLM.[8]

Table 2.6: An example of LLM-generated requirements and features of a group decision support software ("Company Name Decision Application") using the LLM prompts "generate 5 requirements for an application supporting group decisions about company names" and "create an FM with 5 functional features".

requirement	feature
The application should facilitate collaborative and creative brainstorming sessions for company names. It should allow multiple team members to contribute ideas, comment on suggestions, and collectively build a list of potential names	Collaborative Brainstorming (Real-time Editing, Commenting on Ideas)
The application must include a robust voting mechanism to enable team members to express their preferences for specific company names. This can involve a rating system, likes/dislikes, or a voting poll to help identify the most favored options within the group	Voting Mechanism (Rating System, Voting Polls)
The application should integrate a feature to check the availability of selected company names in relevant domains and social media platforms. Additionally, it should provide tools for conducting preliminary trademark searches to ensure legal feasibility and avoid potential conflicts	Availability Check (Domain Availability Check, Social Media Platforms Check)
A comprehensive feedback and commenting system should be in place to allow team members to provide constructive feedback on proposed company names. This feature promotes communication and helps refine name choices based on input from various perspectives	Feedback System (Commenting on Company Name Proposals, Anonymous Feedback Option)
The application should maintain a version history of proposed company names, changes, and discussions. This ensures transparency and allows team members to trace the evolution of naming decisions over time. Additionally, the ability to export or document the final decisions can be valuable for record-keeping purposes	Version History (Tracking Changes Over Time, Rollback to Previous Versions)

Large language models can be applied for the generation of potential constraints regarding a set of identified features. Table 2.7 depicts two LLM-generated example constraints that could be of relevance for the generated set of features. The first constraint in Table 2.7 expresses the idea that rollback only makes sense if a brainstorming history is available. The idea of the second example constraint is that voting only makes sense of the corresponding (internet) domain is available.

[8] The requirements, features, and associated constraints have been generated with ChatGPT 3.5 – see `https://openai.com/`.

Table 2.7: Excerpt of LLM-generated FM constraints (on the features of Table 2.6).

constraint ID	description
1	The "Rollback to Previous Versions" feature requires the selection of the "Collaborative Brainstorming" feature
2	The "Rating System" feature requires the selection of the "Domain Availability Check" feature.

For sure, both, generated features and constraints have to be evaluated by domain experts, however, following this LLM-based approach has the potential to reduce time efforts for FM development and related quality assurance tasks [40, 54].

2.9 Discussion

In this chapter, we have presented different FM knowledge representations and corresponding formalizations in terms of constraint solving and SAT solving. We have discussed extensions to basic feature modelling concepts in terms of attribute- and cardinality-based FMs – these concepts can be regarded as sufficient for representing variability properties in various application domains. Furthermore, we have introduced definitions of an FM configuration task and a corresponding FM configuration which will be used as a basis for the discussions in the following chapters. Finally, we have included scenarios that show how data-driven AI techniques can help in the context of feature modelling. In the context of the topic of feature modelling, we regard the following as major open research issues.

Further Extensions of FM Knowledge Representations. As discussed in this chapter, there are different research streams regarding the extension of basic FMs (e.g., in terms of attributes and cardinalities) and also regarding the standardization of FM representations, specifically on a textual level. Further work on FM standardization could take into account the support of other constraint types [22]. For example, resource constraints are a widely used concept in the context of knowledge-based configuration [33, 29, 74, 78]. When configuring, for example, computer systems, a resource (producer) could be the maximum acceptable price defined by the user (customer) and the corresponding consuming resources would be the hardware components integrated into the computer configuration. A related resource constraint would specify the overall price of the included components must be below the price limit specified by the user. Approaches to configuration knowledge representation in UML/OCL and corresponding logical representations are discussed in Felfernig et al. [29, 32] – taking into account these representations might also be a way to further extend the expressivity of FMs when applied in product configuration. Finally, answer set programming (ASP) is established as an expressive configuration knowledge representation focusing on an object-oriented modelling approach – an application in the context of FM representations is worth further investigations [27, 60, 73].

Cognitive Aspects of FM Development and Maintenance. The development of graphical models is supported by different types of graphical user interfaces (see also Chapter 5). In this context, model understandability is a major criterion for assuring maintainability and consistency of complex models in the long run. Further research is needed to figure out in more detail which types of knowledge structures are more understandable compared to others. Such studies can be performed, for example, on the basis of eye tracking equipment which can help to estimate the cognitive overload of knowledge engineers in their FM development and maintenance activities. Assuring model understandability can also be supported on the basis of basic machine learning methods, for example, different feature and constraint grouping strategies could result in different levels of model understandability [35].

Decision Support in Product Line Scoping. In this chapter, we have provided a simplified example of integrating decision support systems in the process of product line scoping. A related decision support needs to provide specific predefined decision goals such as to maximize the revenue of the offered items but also goals such as maximizing sustainability of the offered solutions and minimize the CO_2 footprint [36]. In this context, also the feasibility of the selected features needs to be taken into account, for example, in terms of available development resources, development risk, and market-related risks [53]. A similar situation occurs in the context of knowledge-based configuration scenarios where a configuration model has to be tailored (also in a scoping process) in such a way that it supports only configurations which can be produced by the existing production infrastructure [53].

Variability Mining. As already mentioned, the increasing size and complexity of FMs triggers a need for the automated support of variability knowledge extraction/mining. Similar to *process mining* where processes are discovered from existing logs [68, 85], we envision techniques and tools inspired by Artificial Intelligence for variability mining. Example areas for future research are the inclusion of techniques for a *user-centered knowledge acquisition* based on the ideas of *human computation* [48, 83] and the application of large language models (LLMs) for the (semi-) automated generation and maintenance of FMs (and beyond) [1, 40].

References

1. M. Acher, J. G. Duarte, and J.-M. Jézéquel. On programming variability with large language model-based assistant. In *27th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '23, pages 8–14, New York, NY, USA, 2023. ACM.
2. M. Acher, H. Martin, L. Lesoil, A. Blouin, J. Jézéquel, D. Khelladi, E. Djamel, O. Barais, and J. Pereira. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 85–96. ACM, 2022.
3. M. Alférez, M. Acher, J. A. Galindo, B. Baudry, and D. Benavides. Modeling Variability in the Video Domain: Language and Experience Report. *Software Quality Journal*, 27(1):307–347, 2019.

4. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented Software Product Lines*. Springer, 2016.
5. S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *International Conference on Algebraic Methodology and Software Technology*, pages 36–50. Springer, 2008.
6. K. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.
7. M. Atas, T. Tran, R. Samer, A. Felfernig, and M. Stettinger. Liquid Democracy in Group-based Configuration. In *20th International Configuration Workshop*, pages 93–98, 2018.
8. G. Audemard, F. Boussemart, C. Lecoutre, C. Piette, and O. Roussel. Xcsp 3 and its ecosystem. *Constraints*, 25:47–69, 2020.
9. D. Batory. Feature Models, Grammars, and Propositional Formulas. In H. Obbink and K. Pohl, editors, *International Conference on Software Product Lines*, pages 7–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
10. D. Batory. *Automated Software Design Volume 1*. Lulu Press, 2020.
11. D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006.
12. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615 – 636, 2010.
13. D. Benavides, C. Sundermann, S. Vill, K. Feichtinger, , J. A. Galindo, R. Rabiser, and T. Thüm. UVL: Feature Modelling with the Universal Variability Language. Technical report, Elsevier, 2024.
14. D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.
15. T. Berger and P. Collet. Usage Scenarios for a Common Feature Modeling Language. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, pages 174–181, 2019.
16. T. Berger, R. Rublack, D. Nair, J. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A Survey of Variability Modeling in Industrial Practice. In *7th international workshop on Variability Modelling of Software-intensive Systems*, pages 1–8, 2013.
17. A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*, volume 185. IOS press, 2009.
18. M. Cashman, J. Firestone, M. Cohen, T. Thianniwet, and W. Niu. DNA as Features: Organic Software Product Lines. In *23rd International Systems and Software Product Line Conference-Volume A*, pages 108–118, 2019.
19. A. Classen, Q. Boucher, and P. Heymans. A text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011.
20. A. Classen, P. Heymans, and P. Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *International Conference on Fundamental Approaches to Software Engineering*, pages 16–30. Springer, 2008.
21. M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 472–481. IEEE, 2013.
22. K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *6th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 173–182. ACM, 2012.
23. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and Their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
24. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software process: improvement and practice*, 10(2):143–169, 2005.
25. R. Dwivedi, D. Dave, H. Naik, S. Singhal, R. Omer, P. Patel, B. Qian, Z. Wen, T. Shah, G. Morgan, and R. Ranjan. Explainable AI (XAI): Core Ideas, Techniques, and Solutions. *ACM Comput. Surv.*, 55(9), jan 2023.

26. H. Eichelberger and K. Schmid. A Systematic Analysis of Textual Variability Modeling Languages. In *17th International Software Product Line Conference*, pages 12–21, 2013.
27. A. Falkner, A. Ryabokon, G. Schenner, and K. Shchekotykhin. OOASP: Connecting Object-Oriented and Logic Programming. In F. Calimeri, G. Ianni, and M. Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 332–345, Cham, 2015. Springer.
28. K. Feichtinger and R. Rabiser. Towards Transforming Variability Models: Usage Scenarios, Required Capabilities and Challenges. In *24th ACM International Systems and Software Product Line Conference - Volume B*, pages 44–51. ACM, 2020.
29. A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Transactions on Engineering Management*, 54(1):41–56, 2007.
30. A. Felfernig. *AI Techniques for Software Requirements Prioritization*, chapter 2, pages 29–47. World Scientific, 2021.
31. A. Felfernig, L. Boratto, M. Stettinger, and M. Tkalcic. *Group Recommender Systems: An Introduction*. Springer Publishing Company Inc., 2nd edition, 2024.
32. A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker. Configuration Knowledge Representations for Semantic Web Applications. *Artif. Intell. Eng. Des. Anal. Manuf.*, 17(1):31–50, 2003.
33. A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen. *Knowledge-based Configuration – From Research to Business Cases*. Morgan Kaufmann, 2014.
34. A. Felfernig, B. Ortner, and V. Le. Table-Based Knowledge Representations for Industrial Feature Models. In *26th ACM International Systems and Software Product Line Conference - Volume B*, pages 245–248. ACM, 2022.
35. A. Felfernig, S. Reiterer, M. Stettinger, F. Reinfrank, M. Jeran, and G. Ninaus. Recommender Systems for Configuration Knowledge Engineering. In *Workshop on Configuration*, ConfWS'13, pages 51–54, 2013.
36. A. Felfernig, M. Wundara, T. Tran, S. Polat-Erdeniz, S. Lubos, M. E. Mansi, and D. G. V. Le. Recommender systems for sustainability: overview and research issues. *Frontiers in Big Data*, 6, 2023.
37. J. K. Fichte, D. L. Berre, M. Hecher, and S. Szeider. The Silent (R)evolution of SAT. *Communications of the ACM*, 66:64–72, 6 2023.
38. N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda. Sat competition 2020. *Artificial Intelligence*, 301:103572, 2021.
39. J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, 101(5):387–433, 2019.
40. J. A. Galindo, A. J. Dominguez, J. White, and D. Benavides. Large language models to generate meaningful feature model instances. In *27th ACM International Systems and Software Product Line Conference-Volume 1*, 2023.
41. Ø. Haugen, A. Wasowski, and K. Czarnecki. CVL: Common Variability Language. In *16th International Software Product Line Conference-Volume 2*, pages 266–267, 2012.
42. R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering*, 27(2):1–34, 2022.
43. P. Hofman, T. Stenzel, T. Pohley, M. Kircher, and A. Bermann. Domain Specific Feature Modeling for Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, pages 229–238. ACM, 2012.
44. K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature–oriented Reuse Method with Domain–specific Reference Architectures. *Annals of software engineering*, 5(1):143–168, 1998.
45. K. Kang, C. Sholom, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
46. A. Karataş, H. Oğuztüzün, and A. Doğru. From Extended Feature Models to Constraint Logic Programming. *Science of Computer Programming*, 78(12):2295–2312, 2013.

47. H. Kuo, J. Chen, S. Mohan, and T. Xu. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Communications of the ACM*, 65(5):101–109, 2022.
48. E. Law and L. von Ahn. *Human Computation*. Morgan & Claypool Publishers, 1st edition, 2011.
49. V. Le, T. Tran, and A. Felfernig. Consistency-Based Integration of Multi-Stakeholder Recommender Systems with Feature Model Configuration. In *26th ACM International Systems and Software Product Line Conference*, pages 178–182, New York, NY, USA, 2022. ACM.
50. V.-M. Le, T. N. T. Tran, M. Stettinger, L. Weißl, A. Felfernig, M. Atas, S. P. Erdeniz, and A. Popescu. Counteracting Exam Cheating by Leveraging Configuration and Recommendation Techniques. In *ConfWS*, pages 73–80, 2021.
51. Y. Li, S. Schulze, H. Scherrebeck, and T. Fogdal. Automated Extraction of Domain Knowledge in Practice: The Case of Feature Extraction from Requirements at Danfoss. In *24th ACM International Systems and Software Product Line Conference*, pages 1–11, New York, NY, USA, 2020. ACM.
52. R. Lindohf, J. Krüger, E. Herzog, and T. Berger. Software Product-Line Evaluation in the Large. *Empirical Softw. Eng.*, 26(2), 2021.
53. S. Lubos, A. Felfernig, V.-M. Le, T. N. T. Tran, D. Benavides, J. A. Zamudio, and D. Garber. Analysis operations on the run: Feature model analysis in constraint-based recommender systems. In *27th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '23, pages 111–116, New York, NY, USA, 2023. Association for Computing Machinery.
54. S. Lubos, A. Felfernig, T. N. T. Tran, D. Garber, M. E. Mansi, S. P. Erdeniz, and V.-M. Le. Leveraging LLMs for the Quality Assurance of Software Requirements. In *32nd IEEE International Requirements Engineering 2024*. IEEE, 2024.
55. L. Marchezan, E. Rodrigues, W. K. G. Assunção, M. Bernardino, F. Basso, and J. Carbonell. Software Product Line Scoping: A Systematic Literature Review. *Journal of Systems and Software*, 186:111189, 2022.
56. A. G. Márquez, Á. J. Varela-Vaca, M. T. G. López, J. A. Galindo, and D. Benavides. Vulnerability impact analysis in software project dependencies based on satisfiability modulo theories (smt). *Computers & Security*, 139:103669, 2024.
57. J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.
58. R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 82–89, 2011.
59. D.-J. Munoz, J. Oh, M. Pinto, L. Fuentes, and D. Batory. A tool to transform feature models with numerical features and arithmetic constraints. In *International Conference on Software and Software Reuse*, pages 59–75. Springer, 2022.
60. V. Myllärniemi, J. Tiihonen, M. Raatikainen, and A. Felfernig. Using Answer Set Programming for Feature Model Representation and Configuration. In *ConfWS'14*, pages 1–8, 2014.
61. V.-H. Nguyen, V.-Q. Nguyen, K. Kim, and P. Barahona. Empirical Study on SAT-Encodings of the At-Most-One Constraint. In *9th International Conference on Smart Media and Applications (SMA 2020)*, SMA 2020, pages 470–475. ACM, 2021.
62. J. Oh, D. Batory, and R. Heradio. Finding near-optimal configurations in colossal spaces with statistical guarantees. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–36, 2023.
63. E. OliveiraJr and D. Benavides. Principles of software product lines. In *UML-Based Software Product Line Engineering with SMarty*, pages 3–26. Springer, 2022.
64. J. Pereira, M. Acher, H. Martin, J. Jézéquel, G. Botterweck, and A. Ventresque. Learning Software Configuration Spaces: A Systematic Literature Review. *Journal of Systems and Software*, 182:111044, 2021.
65. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering*, volume 10. Springer, 2005.

66. A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V. Le, K. Pilsl, M. Enzelsberger, and T. Tran. An Overview of Machine Learning Techniques in Constraint Solving. *Journal of Intelligent Inf. Sys.*, 58(1):91–118, 2022.
67. C. Quinton, D. Romero, and L. Duchien. Cardinality-Based Feature Models with Constraints: A Pragmatic Approach. In *17th ACM International Systems and Software Product Line Conference*, SPLC '13, pages 162–166, New York, NY, USA, 2013. ACM.
68. B. Ramos-Gutiérrez, Á. J. Varela-Vaca, J. A. Galindo, M. T. Gómez-López, and D. Benavides. Discovering configuration workflows from existing logs using process mining. *Empirical Software Engineering*, 26:1–41, 2021.
69. M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, June 2002.
70. D. Romero-Organvidez, D. Benavides, J.-M. Horcas, and M. T. Gómez-López. Variability in data transformation: towards data migration product lines. In *18th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 83–92, 2024.
71. D. Romero-Organvidez, J. Galindo, C. Sundermann, J. Horcas, and D. Benavides. UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles. Technical report, Elsevier, 2023.
72. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. ISSN. Elsevier Science, 2006.
73. N. Rühling, T. Schaub, and T. Stolzmann. Towards a formalization of configuration problems for ASP-based reasoning: Preliminary report. In *ConfWS-2023: 25th International Configuration Workshop (2023)*, volume 3509, pages 85–94. CEUR, 2023.
74. D. Sabin and R. Weigel. Product Configuration Frameworks – A Survey. *IEEE Intelligent Systems and their Applications*, 13(4):42–49, 1998.
75. T. Schnabel, M. Weckesser, R. Kluge, M. Lochau, and A. Schürr. Cardygan: Tool Support for Cardinality-based Feature Models. In *10th International Workshop on Variability Modelling of Software-intensive Systems*, pages 33–40, 2016.
76. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic Semantics of Feature Diagrams. *Computer networks*, 51(2):456–479, 2007.
77. C. Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
78. M. Stumptner. An Overview of Knowledge-Based Configuration. *AICom*, 10(2):111–125, 1997.
79. C. Sundermann, S. Vill, T. Thüm, K. Feichtinger, P. Agarwal, R. Rabiser, J. A. Galindo, and D. Benavides. Uvlparser: Extending uvl with language levels and conversion strategies. In *27th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '23, pages 39–42, New York, NY, USA, 2023. Association for Computing Machinery.
80. P. Temple, J. Galindo, M. Acher, and J. Jézéquel. Using Machine Learning to Infer Constraints for Product Lines. In *20th International Systems and Software Product Line Conference*, pages 209–218, New York, NY, USA, 2016. Association for Computing Machinery.
81. T. Thüm. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. ACM, 2020.
82. T. Tran, A. Felfernig, and V. Le. An overview of consensus models for group decision-making and group recommender systems. *User Model User-Adap Inter*, 2023.
83. T. Ulz, M. Schwarz, A. Felfernig, S. Haas, A. Shehadeh, S. Reiterer, and M. Stettinger. Human Computation for Constraint-Based Recommenders. *J. Intell. Inf. Syst.*, 49(1):37–57, 2017.
84. M. Uta, A. Felfernig, D. Helic, and V. Le. Accuracy- and Consistency-Aware Recommendation of Configurations. In *26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, page 79–84. ACM, 2022.
85. Á. J. Varela-Vaca, J. A. Galindo, B. Ramos-Gutiérrez, M. T. Gómez-López, and D. Benavides. Process mining to unleash variability management: discovering configuration workflows using logs. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 265–276, 2019.
86. M. Wallace. *Building Decision Support Systems: Using MiniZinc*. Springer, 2020.

Chapter 3
Analysis of Feature Models

Abstract Developing and maintaining Feature Models (FMs) can become an error-prone activity. In this chapter, we focus on different aspects of analyzing relevant properties of FMs. Such an analysis helps to increase the maintainability and correctness of FMs and also makes them better manageable in industrial settings. Analysis operations are discussed in detail and also presented formally. In addition to analysis operations, we also show how to automatically determine erroneous elements of an FM that have to be adapted or deleted in order to restore the intended FM semantics.

3.1 Feature Model Analysis Process

As explained in Chapter 2, FMs are a central technique for engineering software product lines. Developing and maintaining FMs can be an error-prone activity due to missing domain knowledge, cognitive overloads of persons in charge of FM development, and outdated knowledge parts in existing FMs [4, 26]. In order to tackle this challenge, intelligent techniques and tools are needed which help to identify anomalies (i.e., unintended properties of FMs which need to be removed) and help to keep maintainability [2]. Such anomalies can exist in different forms and require different types of *analysis operations* capable of identifying those anomalies.

A conceptual process for the automated analysis of FMs is depicted in Figure 3.1. Depending on the specific analysis task, a reasoning engine (a.k.a. solver) is sometimes needed to provide the needed feedback. For example, if we want to figure out the number of features or excludes relationships in an FM, this is just a counting task without the need of activating a solver (e.g., a SAT or constraint solver). On the other hand, if we are interested, for example, in the presence of *dead* of *false optional* features in the FM (these concepts will be defined later), a solver support is needed. For example, if one or more features are dead, a *diagnosis* component can help to identify the responsible FM constraints. Furthermore, *testing & debugging* services can help to systematically test an FM with regard to a test suite specifying the intended semantics of the FM. All these aspects are discussed in this chapter.

A. Felfernig et al., *Feature Models*, SpringerBriefs in Computer Science,
https://doi.org/10.1007/978-3-031-61874-1_3

We want to emphasize that in the context of our discussions specifically due to the focus of this book on *domain and requirements analysis*, the test object is the FM, i.e., we want to assure that the FM represents intended domain knowledge. In related work on analyzing software product lines, the term *testing* often refers to the testing of the software underlying an FM [37].

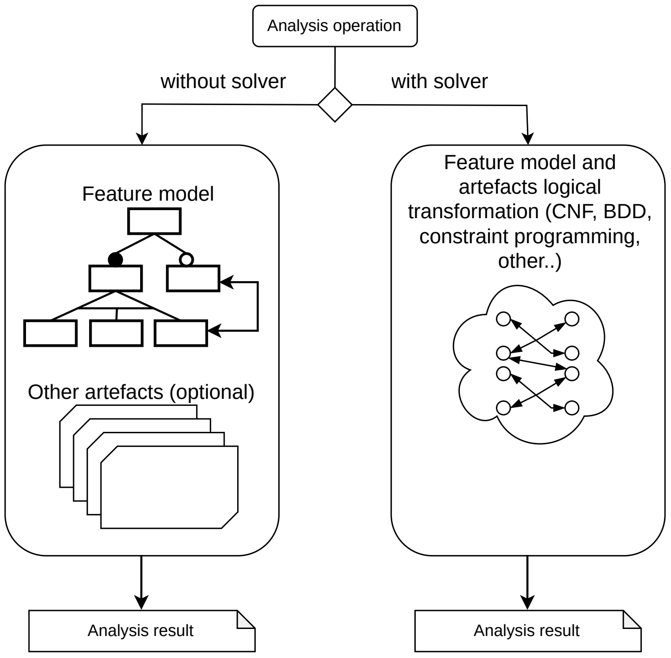

Analysis operation
without solver
with solver
Feature model
Other artefacts (optional)
Feature model and artefacts logical transformation (CNF, BDD, constraint programming, other..)
Analysis result
Analysis result

Fig. 3.1: Automated analysis of FMs conceptual process.

An FM can include other artefacts that are used for analysis purposes. For instance, a table with desired or existing configurations, user ratings of features, or implicit feedback of users during product usage [36, 40] – just to mention a few. It is important to clarify that the software product line engineering process of Figure 2.2 is an iterative process and some of the additional artefacts for FM analysis can be produced in other stages of the process, for example, the implicit feedback of feature usage or the feature ratings after deploying a product.

In the following, we differentiate analysis operations with regard to their need of a corresponding solver or not (see also [23]). Figure 3.2 gives an overview of the structure of the chapter.

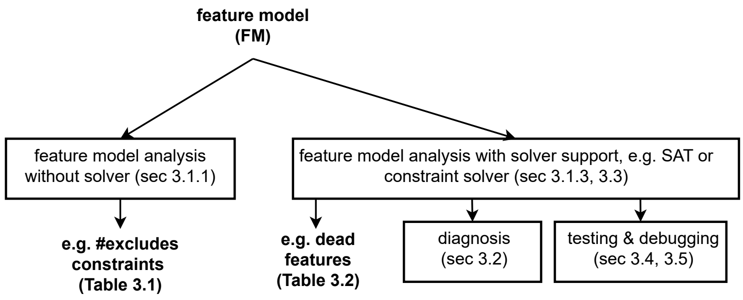

feature model (FM)
feature model analysis without solver (sec 3.1.1)
feature model analysis with solver support, e.g. SAT or constraint solver (sec 3.1.3, 3.3)
e.g. #excludes constraints (Table 3.1)
e.g. dead features (Table 3.2)
diagnosis (sec 3.2)
testing & debugging (sec 3.4, 3.5)

Fig. 3.2: Chapter overview (ids in brackets refer to the corresponding subsection).

3.1.1 Analysis Operations Without Solver Support

There are analysis operations that can be performed without the need of a solver and can be calculated directly from the FM by ad–hoc algorithms. Table 3.1 provides an overview of example operations applied in the context of FM analysis – for related details, we refer to [4] and [22].

Table 3.1: Example FM analysis operations *without the need of solver support*. In this context, F (also denoted as $F(FM)$) denotes the set of features and CF (also denoted as $CF(FM)$) the set of constraints of an FM, $mandatory(c)$ denotes a mandatory relationship c, $optional(c)$ denotes an optional relationship c, $or(c)$ denotes an or relationship c, and $alternative(c)$ denotes an alternative relationship c. Furthermore, $requires(c)$ denotes a requires constraint c and $excludes(c)$ denotes an excludes constraint c.

Analysis Operation	Formalization
#features(FM)	$\|\{f : f \in F(FM)\}\|$
#leaf-features(FM)	$\|\{f \in F(FM) : haschild(f) = false\}\|$
#ancestors(f $\in F(FM)$, FM)	$\|\{parents(f, FM)\}\|$
#mandatory relationships (FM)	$\|\{c : c \in CF(FM) \wedge mandatory(c)\}\|$
#optional relationships (FM)	$\|\{c : c \in CF(FM) \wedge optional(c)\}\|$
#or relationships (FM)	$\|\{c : c \in CF(FM) \wedge or(c)\}\|$
#alternative relationships (FM)	$\|\{c : c \in CF(FM) \wedge alternative(c)\}\|$
#requires relationships (FM)	$\|\{c : c \in CF(FM) \wedge requires(c)\}\|$
#excludes relationships (FM)	$\|\{c : c \in CF(FM) \wedge excludes(c)\}\|$
#commonalities(FM_1,FM_2)	$\|\{f : f \in F(FM_1) \wedge f \in F(FM_2)\}\|$

The notations used in the formalizations included in Tables 3.1 and 3.2 follow the definitions of an FM configuration task and a corresponding FM configuration introduced in Chapter 2. In this context, $F = \{f_1..f_n\}$ denotes the set of features in an FM. The set of constraints is defined as $FMC = CF \cup CR$ where $CF = \{c_1..c_k\}$ is a set of constraints derived from the FM and $CR = \{c_{k+1}..c_m\}$ is a set of constraints representing application requirements, i.e., requirements regarding the inclusion or exclusion of specific features from the user (customer) point of view. Furthermore, an FM configuration $conf$ is an assignment $A = \{f_1 = val(f_1)..f_n = val(f_n)\}$ where $val(f_i) \in \{true, false\}$.

Some analysis operations can be executed without the need of activating a solver (see Table 3.1). In the following, we explain some of those.

Counting features and constraints. These analysis operations basically count the number of features and related constraints (relationships) of an FM. Our example FM (Figure 2.3) has 13 features (including the root feature). There are other operations that traverse the FM tree structure [23]. Examples thereof are the determination of the ancestors (direct and transitive) of a specific feature and the set of leaf-features of an FM. In our example FM, there are 9 leaf-features. The number of ancestors, for example, of the *advancedsolar* feature is 2 (features *energymanagement* and *smartwatch*). Furthermore, the FM includes 2 mandatory relationships, 3 optional relationships, 2 alternative relationships, and 1 *or* relationship. Finally, the model includes 1 requires constraint and 1 excludes constraint. These descriptive numbers help to characterize the size of an FM and can also be used a basis for FM complexity analysis [22, 44].

Differences and commonalities between FMs. Commonalities between FMs can be analyzed, for example, in terms of the number of features with the same names and even further in terms of the number of same constraint types referring to the same features. In contrast, differences between FMs can be analyzed in terms of the number of features only existing in one of the analyzed models. The analysis of differences and commonalities between FMs can play a major role in the context of *FM integration*. For example, if a car provider decides to use the same FM for representing product variabilities in *Europe* as well as in the *US*, the corresponding individual FMs have to be integrated [50]. In this regard, there is another thread of research related with semantic matching, for instance, when an FM expresses a feature with a feature name that is syntactically different from another feature name in another FM but the semantic meaning is the same. Imagine, for instance, that an FM defines a *gps* feature and another one defines a *navigation* feature. Depending on the context, the features can be equivalent. In the past, attempts were made to deal with natural language processing and FMs [42] – such approaches gain momentum specifically in the context of the application of large language models (LLMs) in FM management [19].

3.1.2 Analysis Operations With Solver Support

There are some analysis operations over FMs that are performed with a solver. Table 3.2 provides an overview of example operations frequently applied in the context of FM analysis in the case that solver support is needed (see also [4, 5, 18]). It is important to remark that the discipline has evolved over time and this book intends to recapitulate and provide an updated terminology and conceptual framework. In this sense, reading back some of the related papers shall be done with attention to the terminology used in those papers and in this book. Anyhow, to assure understandability, we try to explain the used terms as much as possible.

Table 3.2: Example FM analysis operations *with solver support*. In this context, F (or $F(FM)$) denotes the set of features and CF (or $CF(FM)$) the set of constraints in FM. Furthermore, $consistent(X)$ indicates that a SAT solver or constraint solver is able to find a solution given the constraints in X. In the context of FMs, this means that at least one configuration $conf$ (i.e., a set of constraints representing feature value assignments) could be identified that complies with the constraints in X. Finally, CF$(FM_x)'$ is the complement of the solution space defined by CF(FM_x) formalized as disjunction of negated constraints of $CF(FM)$. FM_g (FM_s) is the generalized (specialized) FM. The number (#) of satisfiable configurations refers to *complete* configurations. Finally, the *false optional feature* analysis operation excludes *root* and *mandatory* features – this is indicated with $^{(*)}$.

Analysis Operation	Formalization
satisfiable(FM)	$\exists conf : consistent(conf \cup CF(FM))$
satisfiable($conf$,FM)	$consistent(conf \cup CF(FM))$
#satisf. configurations(FM)	$\|\{conf : consistent(conf \cup CF(FM))\}\|$
dead feature (f, FM)	$\neg consistent(\{f = true\} \cup CF(FM))$
false optional (f,FM)$^{(*)}$	$\neg consistent(\{f = false\} \cup CF(FM))$
core feature (f,FM)	$\neg consistent(\{f = false\} \cup CF(FM))$
variant feature (f,FM)	$consistent(\{f = false\} \cup CF(FM)) \wedge$ $consistent(\{f = true\} \cup CF(FM))$
atomic feature set ($S \subseteq F$)	$S = \{f_1..f_\alpha\} : \neg consistent(\{\bigvee_{(f_i \in S, f_j \in S)[i \neq j]} f_i \neq f_j\} \cup CF(FM))$
redundant ($c \in CF$,FM)	$\neg consistent(\{\neg c\} \cup (CF(FM) - \{c\})$
generalization(FM_g, FM_s)	$\neg consistent(CF(FM_s) \cup CF(FM_g)')$
specialization(FM_s, FM_g)	$\neg consistent(CF(FM_s) \cup CF(FM_g)')$
refactoring(FM_1, FM_2)	$\neg consistent(CF(FM_1)' \cup CF(FM_2) \wedge$ $\neg consistent(CF(FM_1) \cup CF(FM_2)'))$
minimal conflict (CS,FM)	$\{CS \subseteq CF(FM) : \neg consistent(CS) \wedge \nexists CS' \subset CS : \neg consistent(CS')\}$

In the following, we give an overview of example analysis operations that can only be executed with a corresponding SAT or constraint solver support.

Satisfiable FM. An FM is *satisfiable* if there exists at least one configuration (*conf*) which is consistent with the FM constraints (defined in CF). The FM in Figure 2.3 is satisfiable, i.e., there exists at least one configuration where all feature settings satisfy the constraints in CF. However, there are non-satisfiable (unsatisfiable) FMs – see, for example, Figure 3.3. In this (faulty) model, two mandatory features are connected with an excludes relationship which induces a contradiction since on the one hand two features are required to co-occur in each configuration, on the other hand, the same features are regarded as incompatible. This operation received different names in the past but basically meaning the same: *void* FM, *invalid* FM, *valid* FM, *consistent* FM, and *solvable* FM [4]. In this book, we use the term FM *satisfiability* to express the meaning of the corresponding analysis operation – this term is also in the line with the concept of satisfiability checking in constraint and SAT solving [3, 18].

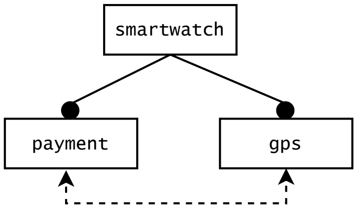


Fig. 3.3: Example of a *non-satisfiable* FM: since both, *payment* and *gps* are mandatory, these features must be part of every configuration, i.e., cannot be incompatible at the same time.

Configuration satisfiability. A configuration is *satisfiable iff* it is *consistent* (all constraints are satisfied) with regard to the FM constraints $FMC = CF \cup CR$. Table 2.1 shows one satisfiable and one non-satisfiable configuration with regard to our example FM. This operation can be useful to determine whether a given configuration is available in a software product line (supported by the FM). In some cases, a configuration or set of configurations are defined and then need to be tested for satisfiability with the FM. If non-satisfiable, the configuration(s) can be changed or maybe the FM itself has to be changed to support the desired configurations [52].

Number of satisfiable configurations. This operation returns the number of configurations represented by the FM. Determining the total number of satisfiable configurations can be relevant in the context of different product line scoping scenarios [35, 17, 20] as well as in the context of deciding about variability properties of products and services, for example, when developing or adapting a company-wide mass customization strategy [7]. FMs can change over time and it can be important to understand the impact of changes on the structure of the configuration (solution) space supported by different versions. The number of satisfiable configurations is also relevant when performing uniform random sampling [21]. The total number of possible satisfiable FM configurations with our example FM of Figure 2.3 is 54.

Dead features. A feature f is *dead* if there does not exist an FM configuration that includes f [6, 9]. Figure 3.4 shows three examples of situations where a feature f is dead. In the first setting, the inclusion of *payment* would require the inclusion of a *basic energy management* resulting in a situation where *advanced solar* could never be included in a configuration. In the second setting, the incompatibility between the features *payment* and *standard* results in a situation where *standard* cannot be part of any configuration. In the third example, since *payment* is incompatible with *gps* and *payment* is mandatory, *gps* will never be part of a configuration, i.e., it is a dead feature. Note that these are just examples and dead features could be induced by an arbitrary combination of complex constraints [9].

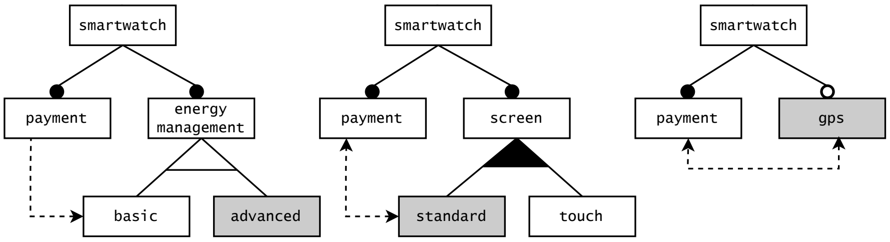


Fig. 3.4: Three basic examples of *dead features* (indicated with a grey background).

False optional features. A feature is *false optional* if it is included in all possible configurations although not being modelled as mandatory. Figure 3.5 shows some examples of false optional features. Having a false optional feature may be a problem if a configurator is built from the FM because a visually optional feature cannot be deselected by the user which can be problematic from a user perspective.

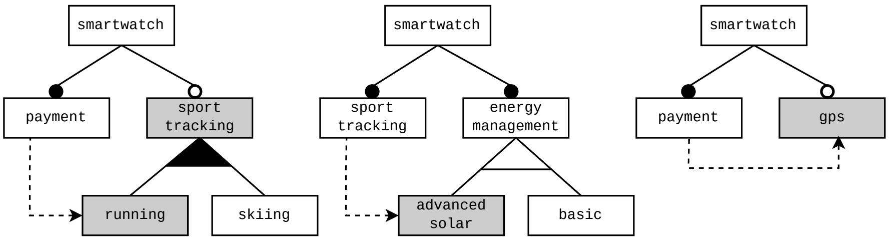


Fig. 3.5: Three basic examples of *false optional features* (grey background).

Figure 3.5 includes three examples of situations where some features become false optional ones. In the first setting, the inclusion of *payment* would require the inclusion of a *running* type *sports tracking* which makes both features – although specified as optional – part of every configuration. Note that once *running* becomes a false optional feature, then *sport tracking* also becomes false optional. In the second setting, *advanced solar* is part of every configuration (although modeled as alternative). In the third example, *gps* is part of every configuration since *payment* is mandatory and requires the feature *gps*. Again, these are just examples – false optional features can be induced by arbitrary constraint combinations [9].

Core and variant features. *Core features* are features that are part of every configuration (see the features *smartwatch*, *screen*, and *energy management* in the running example). *Variant features* are those features that are included in some configurations but also excluded from some others. The set of core features of a non-satisfiable FM is empty. The union of core features, variant features, and dead features, is the set of features of the FM [6].

Atomic sets. Atomic sets of features of an FM can be used as a preprocessing technique for automated analysis and interaction [6]. Informally, an atomic set is a group of features that can be treated as a unit because they are tightly coupled and always appear together in any configuration of an FM. From a formal point of view, atomic sets are nonempty subsets of features such that for every configuration in an FM, all their features appear together in the configuration or none of them appears at all. For a more detailed and formal discussion of atomic sets we refer the reader to [6]. In Table 3.2, the property of an *atomic set* S is defined in terms of the non-existence of a configuration where at least one combination $f_i, f_j (i \neq j)$ of features in S show different values. For example, if $S = \{f_1, f_2, f_3\}$ is an atomic set, then $\{f_1 \neq f_2 \lor f_1 \neq f_3 \lor f_2 \neq f_3\} \cup CF$ is inconsistent.

Redundancies in FMs. An FM can include so-called redundant constraints which – if deleted from the FM – do not change the semantics of the model, i.e., the FM configuration space remains the same (see Figure 3.6).

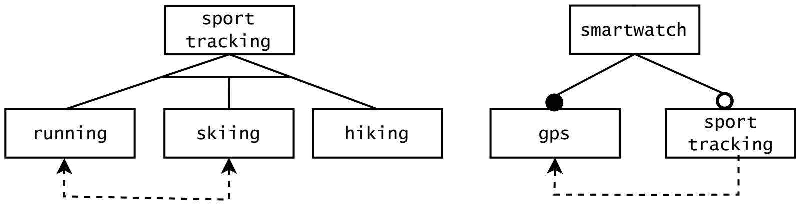


Fig. 3.6: Basic examples of redundancies in FMs : the *excludes* and *requires* constraints in the two models are redundant, i.e., when deleting these constraints, the semantics of the corresponding models remains the same.

On the logical level, a redundant constraint c part of an FM (and the corresponding SAT or constraint satisfaction problem) has the following property: inconsistent($CF - \{c\} \cup \{\neg c\}$), in other words, c logically follows from $CF - \{c\}$ ($CF - \{c\} \models c$). In our example FM, a redundant constraint would be $payment \Rightarrow touch$ since through the exclusion of combining *payment* and *standard* the inclusion of *touch* remains the only allowed alternative for the screen option in the case that payment has been selected. In Figure 3.6, the excludes constraint between *running* and *skiing* is redundant since the excludes semantics is already expressed by the associated alternative relationship. Furthermore, the requires constraint between *sportstracking* and *gps* is also redundant since *gps* has to be part of every configuration. Automated redundancy detection is relevant, for example, in the context of FM development and maintenance. In order to keep an FM understandable, the inclusion of redundant constraints should be avoided.[1]

FM edits. An FM can evolve over time by adding, removing or editing new constraints or features. These changes to FMs are known as *FM edits* [46]. In FM evolution, comparing two FMs can be of help in order to know better the edits that were performed in the model.

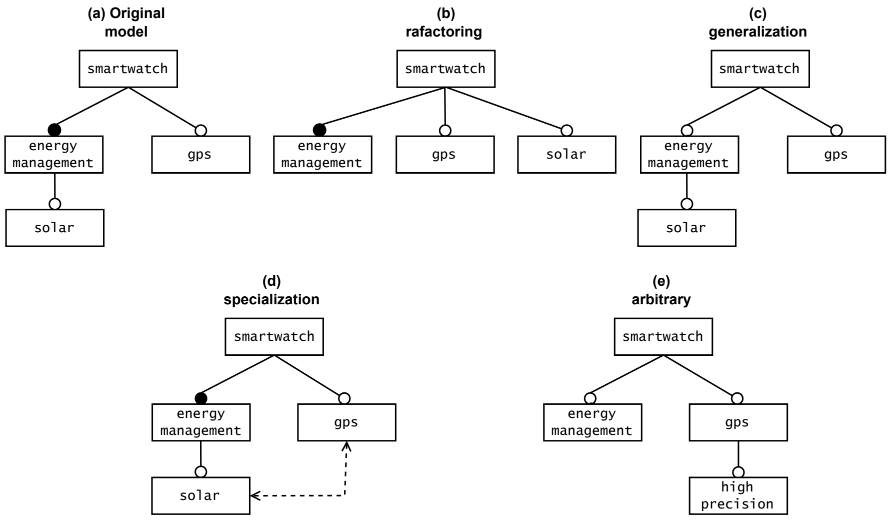


Fig. 3.7: Different types of FM edits: with *refactoring*, the FM configuration space remains the same. With *generalization*, the configuration space is extended with regard to the original model. Furthermore, *specialization* reduces the configuration space. Finally, *arbitrary* edits represent all other FM edit operations.

Comparing edits in FMs is also known as *FM differences* [1]. Figure 3.7 shows how an original FM (a) can be changed by a refactoring (b), generalization (c), specialization (d), or an arbitrary edit (e). These analysis operations are classified as *refactoring* if the original FM represents exactly the same set of configurations as the changed one; *generalization* if the set of configurations of the original FM is a subset of the configurations of the edited FM, *specialization* if the edited FM represents a subset of the configurations of the original FM, or *arbitrary* edit in any other case. These are basic comparisons of FMs but more complex comparisons could be performed in terms of semantic feature similarity. For further discussions on more complex comparisons of FMs we refer the reader to Acher et al. [1]. Also, for a detailed discussion on reasoning about edits in FMs, we refer to Thüm et al. [46]. Note that in our formalization in Table 3.2 we assume that $CF(FM)$ is

[1] An algorithm that can be used for the automated detection of redundant constraints in FMs is discussed in Section 3.3 (see also Le et al. [27]).

always satisfiable. Furthermore, if $CF(FM) = \{c_1, .., c_n\}$ then the corresponding $CF(FM)' = \{\neg c_1 \vee .. \vee \neg c_n\}$.

Dealing with inconsistencies in FMs. An FM could be *non-satisfiable*, i.e., no solution can be found by a corresponding SAT or constraint solver. In such situations, it is important to figure out the sources of such inconsistencies. In other words, we are interested in minimal sets of constraints as part of an FM that have to be deleted or adapted in order to restore model consistency (at least one configuration should be identifiable by a SAT or constraint solver). In this context, conflict detection operations are relevant. In our example, if we include a new constraint $c_{f1} : smartwatch \Leftrightarrow payment$ (basically changing the relationship between *smartwatch* and *payment* from optional to mandatory) and another new constraint $c_{f2} : \neg(payment \wedge touch)$ (see Figure 3.8), this would induce an inconsistency, i.e., no solution could be identified in this situation since *none* of the screen options remains selectable although there is a mandatory relationship between *smartwatch* and *screen*, i.e., at least one screen type should be selectable for a user.

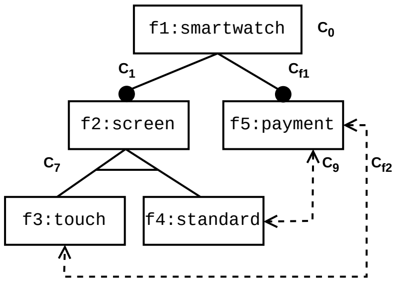


Fig. 3.8: Simplified example faulty (non-satisfiable) FM including the additional constraints $c_{f1} : smartwatch \Leftrightarrow payment$ and $c_{f2} : \neg(touch \wedge payment)$.

Characterizing conflicts and background knowledge. The set of all constraints $CF = \{c_{f1}, c_{f2}, c_0, c_1, c_7, c_9\}$ (see Figure 3.8) makes the FM non-satisfiable. Such a set is denoted as *conflict* or *conflict set* (CS) [24], i.e., an inconsistency-inducing constraint set. In our example, it is impossible to find a configuration that supports both, constraint c_{f1} (the mandatory inclusion of a *payment* feature) and at the same time the exclusion of both, *standard* and *touch* screen, since *screen* is regarded a mandatory feature. Interestingly, deleting an arbitrary constraint from CS allows to restore satisfiability. In this example, the FM has exactly one conflict set comprising all FM constraints. Typically, there are different conflict sets and each of those has to be resolved individually to restore FM consistency. Conflict identification can *focus* on specific CF subsets, for example, a knowledge engineer might be interested to know which new constraints of $\{c_{f1}, c_{f2}\}$ are responsible for a non-satisfiable FM. In such a situation, the constraints from the original FM, i.e., $\{c_0, c_1, c_7, c_9\}$, are regarded as *background knowledge* B and conflict search is focused on $\{c_{f1}, c_{f2}\}$.

Conflicts in FMs and corresponding formalizations (e.g., constraint satisfaction problems) can occur in different situations: (1) if an FM is non-satisfiable, knowledge engineers have to be supported in identifying the responsible conflicts in CF. (2) it could also be the case that the FM is satisfiable, i.e., at least one solution can be identified, however, some configurations derived from the FM do not reflect existing real domain constraints (e.g., the smart watch product line allows to have basic screen but in fact the smart watch factory currently does not provide basic screens). In order to be able to identify such "unintended behaviors" of FMs, test suites (with individual test cases representing real-world properties/constraints) are defined that can be used for quality checking in the context of FM evolution. In Section 3.2, we will introduce basic concepts of conflict detection and conflict resolution on the basis of model-based diagnosis [39]. In Section 3.4, we show how FMs can be analyzed/tested with test suites that define the intended behavior of an FM. More precisely, test cases specifying intended semantics are used to induce conflicts in FMs which are then resolved with model-based diagnosis.

3.2 Diagnosing Inconsistent Constraint Sets

The increasing size and complexity of FMs and their widespread industrial use trigger a demand for the automated identification of faulty FM constraints [34]. The foundation for such a support are *conflict detection* [24, 29, 51] and *diagnosis* [39] algorithms which support the identification of faulty constraints (relationships) that represent an *explanation* for the faulty behavior of an FM (a kind of *why not* explanation). In this context, (1) *conflict detection* is used to identify minimal subsets of constraints that are inconsistent (i.e., minimal unsatisfiable subsets – MUS [31]), and (2) *diagnosis* helps to identify minimal sets of constraints that have to be adapted or deleted from the FM such that the new version of the model is satisfiable (i.e., minimal correction subsets – MCS [45]).

A diagnosis [39, 52] is a *hitting set* which is a set of constraints that have to be deleted from the conflict sets such that all conflicts are resolved. We now show how to determine conflicts and corresponding diagnoses. For demonstration purposes, we use a reduced version of the FM shown in Figure 3.8. The corresponding FM configuration task (F, D, CF) is the following.[2] Note that, we kept the constraint identifiers that have been introduced in Chapter 2.

- $F = \{smartwatch, screen, touch, standard, payment\}$
- $D = \{dom(smartwatch) = dom(screen) = dom(touch) = dom(standard) = dom(payment) = \{true, false\}\}$
- $CF = \{c_0 : smartwatch = true, c_1 : smartwatch \Leftrightarrow screen, c_7 : screen \Leftrightarrow (touch \oplus standard), c_9 : \neg(payment \wedge standard), c_{f1} : smartwatch \Leftrightarrow payment, c_{f2} : \neg(payment \wedge touch)\}$

[2] For a definition of a *FM configuration task* see Chapter 2. Due to our focus on model analysis, we omit application requirements, i.e., we focus on CF.

In the FM shown in Figure 3.8, we have added two faulty constraints for demonstration purposes. (1) the constraint c_{f1} specifies a mandatory inclusion of the payment feature in every possible configuration. Furthermore, constraint c_{f2} specifies an incompatibility between the features *payment* and *touch screen*. In real-world settings, such a combination is obligatory in the sense that payments require interactive user interface elements and for this reason, c_{f2} can be regarded as faulty.

Reasons of faulty constraints. There are various possible reasons for the existence of faulty constraints in an FM. (1) It could be the case that due to an increasing complexity of an FM, cognitive overloads are the reason for misinterpreting specific feature relationships. (2) In some cases, the reason for the inclusion of faulty constraints is missing domain knowledge. (3) Another reason is outdated domain knowledge, i.e., FM elements that have been included long time ago but have not been adapted to reflect the new variability properties.

When starting a constraint or SAT solver to calculate a solution for our example FM configuration task, the result will be *no solution could be identified*. The reason behind is that each *smartwatch* has to include a screen and a screen has to be either a *touch* screen or a *standard* screen. At the same time, the payment feature is defined as being mandatory with further constraints forbidding a combination of *payment* with a *standard* screen or *touch* screen.

Before taking a more detailed look at mechanisms that support conflict detection and resolution, we introduce the following definition of a *conflict set*.

Definition 3.1 (Conflict Set CS). $CS = \{c_1..c_v\}$ is a subset of a constraint set C with inconsistent(CS). CS is *minimal* if $\neg\exists CS' : CS' \subset CS$ and CS' is a conflict set.

Three basic conflict detection scenarios. This definition of a conflict set can be applied in different scenarios. (1) If the set CF of FM constraints does not allow the determination of a solution, then we need to search for a conflict set in CF, i.e., $C = CF$ (see **Subsection 3.2.1**). (2) Assuming the consistency of CF and the inconsistency of $CF \cup CR$, we are interested in figuring out a set of user preferences that are inconsistent with CF. In this case, a conflict set will be found in CR, i.e., $C = CR$ (see **Subsection 3.2.2**). (3) In FM quality assurance, we have to be able to support knowledge engineers in understanding and explaining unintended FM "behavior" (i.e., semantics) in the sense that a SAT or constraint solver proposes FM configurations which are unintended, i.e., in contradiction with related real-world domain constraints. For example, if a *payment* feature must be combined with *touch screen* (real-world domain constraint) but the solver does not allow such configurations, adaptations in the FM are needed in order to reflect real-world domain constraints. Also in this context, we are interested in constraint sets (i.e., FM relationships and cross-tree constraints) which are responsible for the unintended semantics. We have to analyze the constraints in CF, i.e., $C = CF$, where conflicts are induced by test cases specifying intended FM semantics (see **Section 3.4**).

Minimality properties of conflict sets. A minimal conflict set CS allows conflict resolution by deleting only one element from CS. Minimality properties of relevance in this context are *subset minimality* (see Definition 3.1) and *minimal cardinality* where the latter one is more restrictive in the sense of minimizing the number of conflict elements. Minimal cardinality conflict sets are preferred in situations where an increasing conflict set size would deteriorate solution quality. For example, in group decision making, minimal cardinality conflicts can help to reduce the overall communication overhead related to conflict resolution [8, 28, 47]. Subset minimality, on the other hand, is specifically useful in scenarios where there are preference relationships over different features. For example, in the context of our smartwatch example, users might have a strong upper price limit in mind and are more flexible with regard to the inclusion or exclusion of specific other features. In such a situation, conflict resolution should focus on conflicts induced by "unimportant" constraints. Specifically in realtime scenarios such as network load balancing or scheduling, minimality criteria have to be relaxed in order to be able to take into account corresponding response time requirements [16].

3.2.1 Identifying Conflict Sets in Non-Satisfiable Feature Models

In the following, we show how a conflict, more precisely, a minimal conflict set, can be determined on the basis of QuickXPlain [24] which is a widely used *divide-and-conquer* based approach to identification of *subset-minimal* conflicts. The basic idea of QuickXPlain is the following: given an inconsistent set of constraints, for example, $C = \{c_1..c_{20}\}$ and $\{c_1..c_{10}\}$ is inconsistent, a minimal conflict set can be identified in $\{c_1..c_{10}\}$, i.e., the remaining constraints $\{c_{11}..c_{20}\}$ can be omitted after one consistency check. If $\{c_1..c_{10}\}$ is consistent, the conflict has to be searched in $\{c_1..c_{15}\}$ (i.e., $\{c_1..c_{10}\}$ extended with the "first half" of $\{c_{11}..c_{20}\}$). In this situation, the scope of conflict search has to be extended until an inconsistent state is reached. QuickXPlain can be activated with a consideration set C, i.e., the set of constraints with an expected conflict and a constraint set B representing the background knowledge which is assumed to be consistent (see Algorithm 1).

In our example, we assume the background knowledge $B = \{c_0, c_1, c_7, c_9\}$, i.e., the set of constraints which are assumed to be correct. Furthermore, C is the set of constraints inducing an inconsistency with B and – for this reason – includes one or more conflict sets. QuickXPlain is flexible and we are able to apply the algorithm in scenarios where C represents an inconsistent set of user requirements, i.e., $C = CR$, but – beyond that – also in scenarios where the FM constraints are inconsistent (e.g., in the context of a non-satisfiable FM), i.e., $C = CF$. IF $C \cup B$ is consistent or $C = \emptyset$, QuickXPlain returns $\emptyset$. In any other case, the conflict detection process is started by activating QX (Algorithm 2) which is a divide-and-conquer based routine for the identification of minimal conflict sets (in C).

The search for a minimal (irreducible) conflict set λ in the consideration set C is performed by QX (see Algorithm 2) where λ satisfies the following property: $\nexists \lambda' \subset \lambda$: conflict set(λ'), i.e., no proper subset of a minimal (irreducible) conflict set can be a conflict set. If B is consistent and C has more than one element, C is

Algorithm 1 QuickXPlain($C = \{c_1..c_n\}, B) : \lambda$

```
1: if Consistent($C \cup B$) then
2:     return('no conflict')
3: else if $C = \emptyset$ then
4:     return($\emptyset$))
5: else
6:     return($QX(\emptyset, C, B)$))
7: end if
```

divided into two separate sets, where C_1 is added to B in order to analyse further elements of the conflict. If C includes only one element ($|C| = 1$), this element can be considered as as part of the minimal conflict set – this is due to the invariant property *inconsistent*($C \cup B$), i.e., since B is consistent, C must be responsible for inducing the conflict. In the context of the consistency check of B (line 1), δ indicates which constraints have been added to B in the previous step.

Algorithm 2 QX($\delta, C = \{c_1..c_n\}, B) : \lambda$

```
1: if $\delta \neq \emptyset \wedge inconsistent(B)$ then
2:     return($\emptyset$)
3: end if
4: if $|C| = 1$ then
5:     return($C$)
6: else
7:     $k = \lfloor \frac{n}{2} \rfloor$
8:     $C_1 \leftarrow c_1...c_k; C_2 \leftarrow c_{k+1}...c_n;$
9:     $\lambda_2 \leftarrow$ QX$(C_1, C_2, B \cup C_1);$
10:    $\lambda_1 \leftarrow$ QX$(\lambda_2, C_1, B \cup \lambda_2);$
11:    return($\lambda_1 \cup \lambda_2$)
12: end if
```

Assuming the consideration set $C = \{c_{f1}, c_{f2}\}$ consisting of the two additional constraints of our working example (i.e., we first want to focus our search for minimal conflicts on the two new constraints) and $B = \{c_0, c_1, c_7, c_9\}$, we want to sketch the execution of Algorithms 1 – 2. Algorithm 2 is based on depth-first search where in every case the left branch is responsible for determining λ_2 whereas the right branch is responsible for determining λ_1. In our example (see Figure 3.9), the determined minimal conflict set is $\lambda = \{c_{f1}, c_{f2}\}$ which means that there are two possible conflict resolutions: (1) to delete c_{f1} and (2) to delete c_{f2}.

3.2.2 Identifying Conflict Sets in User Requirements

In the example introduced in Chapter 2, $CF = \{c_0..c_{10}\}$ represents the constraints derived from the FM (see Figure 2.3). Let us assume a set of user (customer) re-

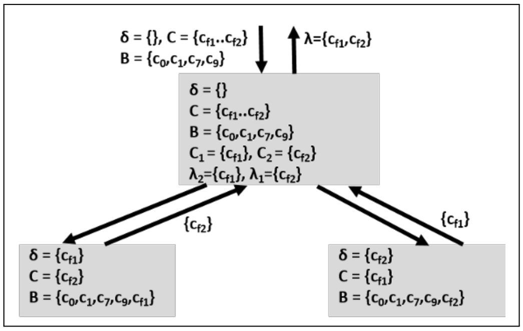


Fig. 3.9: Execution trace of QX on the basis of $C = \{c_{f1}, c_{f2}\}$ and $B = \{c_0, c_1, c_7, c_9\}$ resulting in the minimal conflict set $\lambda = \{c_{f1}, c_{f2}\}$.

quirements specifying preferences regarding the inclusion and exclusion of specific features which is represented as a set of constraints $CR = \{c_{11} : payment = true, c_{12} : running = true, c_{13} : standard = true, c_{14} : gps = false\}$. These requirements specify a *smartwatch* with a standard screen and the payment feature. Furthermore, a *sportstracking* support for *running* is required, however, the user does not want to include a *gps* feature. In this example setting, we assume that the user is free in including or excluding features and the configurator (based, e.g., on a SAT solver or constraint solver) is in charge of checking the consistency of $CF \cup CR$.

Since $CF \cup CR$ is inconsistent in our case, conflict detection can help to resolve the inconsistency. In this context, we assume $B = CF$ (the FM constraints are assumed to be correct) and the consideration set $C = CR$ is the set of application requirements that are responsible for the inconsistency $CF \cup CR$. In this situation, we are interested in the conflict sets in $CR(C)$ that need to be resolved in order to restore the consistency of $CR \cup CF$. The steps to determine all minimal conflict sets in CR using QuickXPlain (Algorithm 1) are shown in Figure 3.10 – 3.11.

The first minimal conflict set derived from $CR = \{c_{11}..c_{14}\}$ is $CS_1 = \lambda = \{c_{11}, c_{13}\}$. This conflict could be shown to the user with the additional information that at least one of the requirements has to be adapted, i.e., either switched from inclusion to exclusion or vice-versa. Let us assume that the user decides to change his/her preferences regarding the requirement c_{11}, i.e., he/she accepts the exclusion of *payment*, our set of user requirements changes to $CR = \{c_{12} : running = true, c_{13} : standard = true, c_{14} : gps = false\}$. Based on this changed situation, the QuickXPlain algorithm can be reactivated with $B = \{c_0..c_{10}\}$ and an adapted set of user requirements $C = \{c_{12}, c_{13}, c_{14}\}$ (no need to include c_{11} since CS_1 has been resolved by deleting/adapting c_{11}).

The second minimal conflict set (derived from CR) is $CS_2 = \lambda = \{c_{12}, c_{14}\}$. Again, a user can decide how to resolve this conflict. For example, if a user has a strong preference in including the *sportstracking running* feature, he/she has to

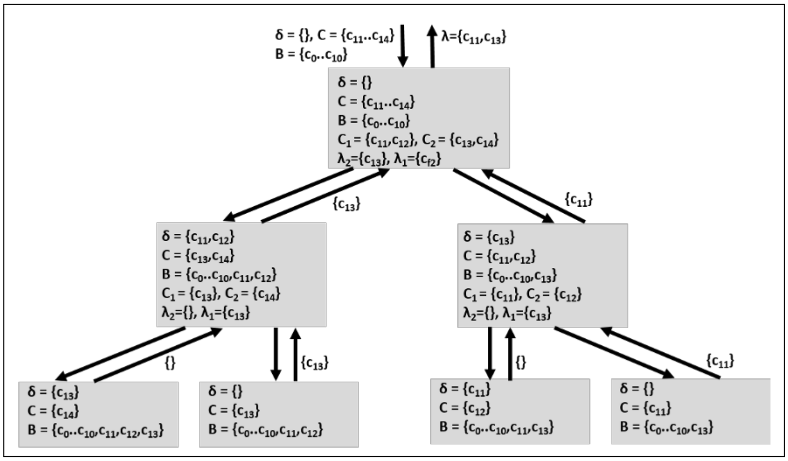


Fig. 3.10: Execution trace of QX on the basis of $C = \{c_{11}..c_{14}\}$ and $B = \{c_0..c_{10}\}$ resulting in the minimal conflict set $\lambda = \{c_{11}, c_{13}\}$.

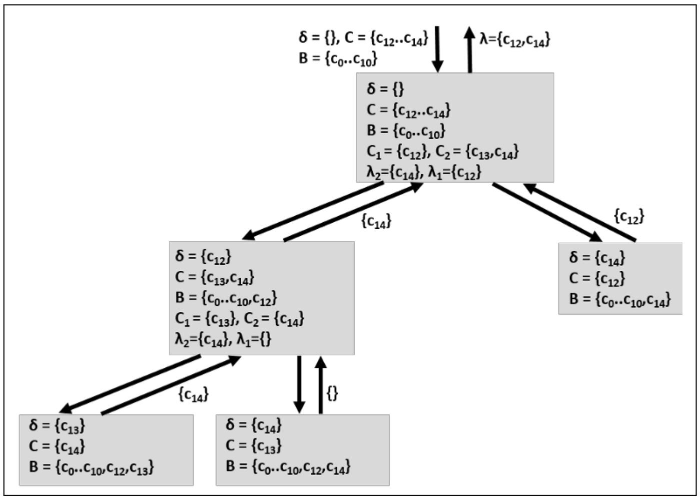


Fig. 3.11: Execution trace of QX on the basis of $C = \{c_{12}..c_{14}\}$ and $B = \{c_0..c_{10}\}$ resulting in the minimal conflict set $\lambda = \{c_{12}, c_{14}\}$.

accept the inclusion of the *gps* feature. Summarizing, in our example a user decided to accept the exclusion of the *payment* feature and also accepted the inclusion of the *gps* feature. These accepted adaptations are also denoted as *hitting sets* or *diagnoses* [39] – corresponding algorithmic approaches will be discussed in the following.

The role of constraint orderings. In many conflict detection scenarios, there is an exponential number of conflicts [24]. QuickXPlain is able to identify so-called *preferred conflicts* (one at a time), i.e., conflicts with a high probability of being of relevance for the user. Using QuickXPlain, the returned minimal conflict set can differ depending on the original ordering of the constraints in the consideration set C where it can be assumed that constraints at the beginning of the constraint list have the lowest importance and constraint importance increases with a corresponding higher ranking in the list. In our example, the first returned conflict set is $\{c_{11}, c_{13}\}$ with c_{11} having the lowest importance of all constraints in $CR = \{c_{11}..c_{14}\}$ representing user requirements CR. If we would change the order of our example constraints to $CR\{c_{14}, c_{13}, c_{12}, c_{11}\}$ (assuming an ordered set semantics), QuickXPlain would first return the minimal conflict set $CS_1 = \{c_{12}, c_{14}\}$.[3]

Further approaches to conflict detection. Further conflict detection approaches are based on the idea of integrating the search for conflicts into solution search. For example, the constraints of an FM can be reformulated as follows: if, for example, $c_1 : smartwatch \Leftrightarrow screen$ is the original constraint, the corresponding reformulation could be $c_1 : c1 = 1 \Rightarrow smartwatch \Leftrightarrow screen$ where c_1 is assumed to be a Boolean variable used for counting the number of activated constraints. Using such a representation, we are able to formulate the conflict set identification task as a minimization problem as follows: *mincardinality-set*($\{\{c_1..c_k\} : inconsistent(\{c_1..c_k\})\}$) which means to minimize $c_1 + .. + c_k$. For an overview of approaches that support the identification of minimal conflict sets (minimal unsatisfiable cores) we refer to Liffiton et al. [31, 48].

Resolution of conflicts based on diagnosis. The overall goal in most settings is to identify a minimal set of constraints that have to be deleted (adapted) in order to be able to restore consistency in a given constraint set – such sets can be denoted as a diagnoses (hitting sets) [15, 39]. In our example of inconsistent user requirements, one diagnosis (the one preferred by the user) is $\{c_{11}, c_{14}\}$, i.e., by adapting the requirements specified with $\{c_{11}, c_{14}\}$, the consistency of $CF \cup CR$ can be restored. In our example non-satisfiable FM, a diagnosis would be c_{f1}. In both cases, a diagnosis denotes a (minimal) set of constraints that have to be adapted or deleted to restore consistency.

Before taking a more detailed look at mechanisms that support diagnosis determination, we introduce a definition of a *diagnosis* (see Definition 3.2).

Definition 3.2 (Diagnosis). A diagnosis $\Delta = \{c_1..c_k\}$ is a subset of C with consistent($C - \Delta$). Δ is *minimal* if $\neg\exists\Delta' : \Delta' \subset \Delta$ and Δ' is a diagnosis.

Three basic diagnosis scenarios. Definition 3.2 can be applied in different scenarios. (1) If an FM is *non-satisfiable*, we need to search for a diagnosis in the set CF of FM constraints. (2) assuming FM consistency and – at the same time – inconsistency of $CF \cup CR$, a diagnosis can be identified in the set CR. (3) if an FM is consistent, it can still be the case that it shows an unintended semantics, for example, it allows to determine configurations which are not allowed in the application domain (and could lead to potentially faulty software configurations).

[3] For details regarding the QuickXPlain constraint ordering, we refer to Junker [24].

In this context, a diagnosis can again be searched in CF but conflicts are induced by *test cases* specifying the intended semantics of an FM [10] (see Section 3.4). Diagnoses are often denoted as *hitting sets* [39] or *minimal correction subset* (MCS) [31]. Furthermore, the complement of a hitting set is denoted as *maximal satisfiable subset* (MSS) [31]. In the context of a minimal diagnosis Δ, no subset of Δ fulfills the diagnosis property. In the context of an MSS Γ, no extension of Γ is consistent, i.e., allows the calculation of a solution.

Minimality properties of diagnoses. An important aspect of diagnosis minimality is that it is guaranteed that only those constraints (requirements) are adapted which need to be adapted in order to restore consistency. Similar to conflict sets, minimal diagnoses can be either *subset minimal* or of *minimal cardinality* (the first interpretation is used in Definition 3.2). Minimal cardinality diagnoses are preferred in scenarios where there are no clear preferences regarding individual adaptations of constraints. In contrast, *subset minimal* diagnoses are used in contexts where preferred constraints should be kept as-is whereas less preferred constraints should be the preferred diagnosis candidates.

Determining minimal cardinality diagnoses in non-satisfiable FMs . The basic approach to diagnosis determination is to delete at least one element from each individual conflict set (then, all conflicts are resolved). In our example non-satisfiable FM (see Figure 3.8), there exists exactly one conflict (set) which is $CS = \{c_{f1}, c_{f2}\}$. There are two possible ways of resolving this conflict which is described by the two diagnoses $\Delta_1 = \{c_{f1}\}$ and $\Delta_2 = \{c_{f2}\}$ meaning that either c_{f1} or c_{f2} has to be adapted or deleted in order to restore the consistency in the FM. This way, we are able to support engineers in the development of FMs by relieving the burden of manually identifying faulty constraints.

Determining minimal diagnoses in user requirements. Similar to the determination of diagnoses in non-satisfiable FMs, diagnosis determination in the context of inconsistent user requirements is based on the resolution of individual conflicts. Given the two conflict sets in the context of inconsistent user requirements ($CS_1 = \{c_{11}, c_{13}\}$ and $CS_2 = \{c_{12}, c_{14}\}$), we are able to derive four corresponding diagnoses (Δ_1, Δ_2, Δ_3, Δ_4) based on the construction of a hitting set directed acyclic graph (HSDAG) [39]. After having resolved, for example, CS_1 by removing the constraint c_{11}, we still have to resolve CS_2 with two remaining options, namely constraint c_{12} or constraint c_{14}. In this example, each path of our example HSDAG leads to a correponding minimal diagnosis. However, this is not always the case, for example, some of the paths have to be closed (not taken into account) since other completed paths already represent a minimal diagnosis (which is a subset of a diagnosis described by the current path). Such a closing of nodes is important to assure efficiency of HSDAG determination [39]. Furthermore, conflict sets do not have to be calculated for every node in the HSDAG. As can be seen in our example (Figure 3.12), the conflict set $CS_2 = \{c_{12}, c_{14}\}$ occurs twice, however, there is no need for recalculation, for example, with QuickXPlain (see Algorithm 1).

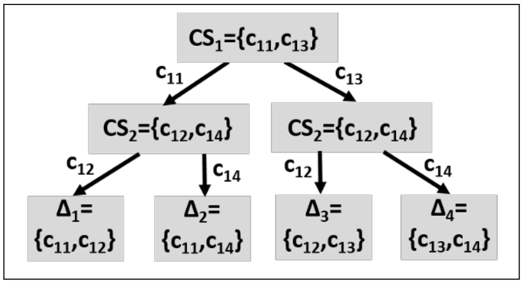


Fig. 3.12: Hitting Set Directed Acyclic Graph (HSDAG) for the conflict sets $CS_1 = \{c_{11}, c_{13}\}$ and $CS_2 = \{c_{12}, c_{14}\}$. The resulting minimal diagnoses are $\Delta_1 = \{c_{11}, c_{12}\}$, $\Delta_2 = \{c_{11}, c_{14}\}$, $\Delta_3 = \{c_{12}, c_{13}\}$, and $\Delta_4 = \{c_{13}, c_{14}\}$
.

Alternative diagnosis algorithms. In contrast to the hitting set based approach to diagnosis determination [39], *direct diagnosis* supports the determination of hitting sets without the need of a corresponding conflict detection. An example of such a diagnosis algorithm is FastDiag [15] which is based on the idea of determining subset minimal diagnoses on the basis of a divide-and-conquer based approach. The underlying idea is the following: given an inconsistent set of constraints, for example, $C = \{c_1..c_{20}\}$ and $\{c_1..c_{10}\}$ is consistent, a minimal diagnosis can be identified in $\{c_{11}..c_{20}\}$, i.e., $\{c_1..c_{10}\}$ can be excluded from diagnosis search. Further diagnosis approaches follow the idea of integrating diagnosis and solution search. As discussed in the context of conflict detection, constraints can be reformulated, for example, $c_1 : smartwatch \Leftrightarrow screen$ can be reformulated as $c_1 : c1 = 1 \Rightarrow smartwatch \Leftrightarrow screen$. A diagnosis task for a constraint set $\{c_1..c_n\}$ can then be interpreted as a minimization task $mincardinalityset(\{\{c_\alpha..c_\lambda\} : consistent(\{c_1..c_n\} - \{c_\alpha..c_\lambda\})\})$ which means to minimize $c\alpha + .. + c\lambda$.

Relationship between diagnoses and conflicts. Note that there is a natural relationship between minimal conflict sets and minimal diagnoses in terms of a duality property [43]: for a given set of minimal conflicts (i.e., a set of sets) we are able to determine a corresponding set of minimal diagnoses (again, a set of sets) on the basis of a hitting set directed acyclic graph (HSDAG) [39]. Vice-versa, exactly the same set of conflicts can be determined by constructing a HSDAG from the given set of diagnoses. This could be useful, for example, in a situation where one is interested in determining minimal cardinality conflict sets (in contrast to subset-minimal conflict sets) which can be determined on the basis of hitting set based breadth-first search [39].

3.3 Redundancy Detection in Feature Models

The development and maintenance of FMs can be a time-consuming task leading to the inclusion of unintended semantics into FMs. A somewhat orthogonal aspect compared to the topics discussed up to now is the occurrence of redundancies in FMs. Elements (constraints) which are redundant can *increase* development and maintenance efforts of FMs (due to a decreased model understandability) and *decrease* the efficiency of constraint and SAT solvers [25, 27]. In FMs, a redundancy can be interpreted as a collection of model elements that can be removed from the FM without changing its semantics in terms of the solution space defined by the FM [25, 26]. More formally, if $CF = \{c_1..c_m\}$ is a set of constraints and c_α is redundant ($c_\alpha \subset CF$) then $CF - \{c_\alpha\} \cup \{\neg c_\alpha\}$ is inconsistent, i.e., the solution space of $CF - \{c_\alpha\}$ corresponds to the original one. Consequently, if $c_\alpha \in CF$ is redundant, c_α *logically follows* from $CF - \{c_\alpha\}$, i.e., $CF - \{c_\alpha\} \models c_\alpha$. An algorithm for determining the complete set of non-redundant constraints (CF_Δ) in an FM is FMREDUNDANCY (see Algorithm 3). The overall idea is to iterate over all constraints defined in CF, i.e., the constraints derived from the FM, and to analyze each constraint with regard to redundancy.

Algorithm 3 FMREDUNDANCY($CF = \{c_1..c_n\}$): CF_Δ

```
1: $CF_\Delta \leftarrow CF$
2: for all $c_\alpha \in CF$ do
3:     if $inconsistent(CF_\Delta - \{c_\alpha\} \cup \{\neg c_\alpha\})$ then
4:         $CF_\Delta \leftarrow CF_\Delta - c_\alpha$
5:     end if
6: end for
7: return($CF_\Delta$)
```

In the FM of Figure 2.3, no redundant constraints can be identified. If we add the constraint $c_r : sportstracking \Rightarrow energymanagement$ to set of FM constraints CF, a corresponding redundancy can be identified: the *energymanagement* feature is mandatory, consequently a constraint requiring the context-dependent inclusion of this feature is redundant since this feature will be included anyway. An execution trace of Algorithm 3 can be found in Table 3.3.

Table 3.3: Identification of redundant constraints in CF based on FMRENDUNDANCY.

id	CF_Δ	current constraint (c_α)	redundant(c_α)
1	$\{c_0, .., c_{10}, c_r\}$	c_0	×
2	$\{c_0, .., c_{10}, c_r\}$	c_1	×
3	$\{c_0, .., c_{10}, c_r\}$	c_2	×
..	$\{c_0, .., c_{10}, c_r\}$	..	×
11	$\{c_0, .., c_{10}\}$	c_r	✓

In our example, CF_Δ (the redundancy-free constraint set derived from CF) is returned by Algorithm 3. This set guarantees that the semantics of the corresponding FM remains the same, i.e., the FM including CF has the same solution (configuration or product) space as the FM including CF_Δ.

3.4 Feature Model Testing and Debugging

In the previous sections, we have discussed the concepts of conflict detection and diagnosis on the basis of the scenarios of (1) restoring the satisfiability of an FM and (2) restoring the consistency of user requirements (within the scope of a configuration process). In this section, we focus on situations where an FM is satisfiable but still does not behave as expected in the sense that FM configurations are supported (or even generated) that are not allowed or expected in the corresponding application domain. The reasons for such an unintended semantics are manyfold and range from insufficient domain knowledge of engineers, outdated knowledge still included in the FM, to cognitive overheads of engineers triggered by FMs of low understandability.

Our example FM depicted in Figure 2.3 does not include any dead features, i.e., every feature is activated in at least one configuration part of the complete set of possible configurations that can be derived from the FM. If we analyze our FM with the corresponding analysis operation (see *dead feature (f)* Table 3.1), the outcome will be as expected. In situations where some features are dead, the question arises in which way the FM has to be adapted in order to exclude dead features. More generally, how to adapt the FM in such a way that specific properties (e.g., satisfiable FMs , no dead features, and no false optional features) are fulfilled. A well-known concept in software engineering scenarios are test cases and test suites which are used to assure a specific intended behavior of a software. If some test cases fail, corresponding adaptations are needed by developers. In the same sense, we are able to define test cases for FMs in such a way that the unintended semantics can be discovered and at the same time those model parts can be automatically identified that are responsible for this unintended semantics.

A simple example of a test suite $T = \{t_1..t_n\}$ is the following: $T = \{t_1 : smartwatch = true, t_2 : screen = true, .., t_{13} : advancedsolar = true\}$, i.e., each feature is represented by a corresponding test case that is used *to assure that no feature is dead*. This approach to define intended FM semantics in terms of test cases can be applied to other types of analysis operations as well as in a more general case by specifying example-wise intended semantics. In the discussed approach, test cases are regarded as constraints that define intended (and also unintended) semantics of FMs.[4] The underlying idea is to exploit a defined set of test cases to induce inconsistencies in an FM and to resolve these inconsistencies on the basis of model-based diagnosis. In order to apply conflict detection and diagnosis in such

[4] In the following, we focus our discussion on the specification of intended semantics (i.e., positive test cases).

scenarios, we need to define the concepts of a conflict set and a corresponding diagnosis (see the Definitions 3.3–3.4). We use the term *debugging* for actions to restore the consistency and semantic correctness of an FM, i.e., getting the FM free from buggy constraint definitions. In the context of FM testing and debugging, a conflict set can be defined as follows (see Definition 3.3).

Definition 3.3 (Conflict Set). A conflict set $CS = \{c_1..c_v\}$ is a subset of CF s.t. $\exists t \in T : inconsistent(CS \cup \{t\})$. CS is *minimal* if $\neg\exists CS' : CS' \subset CS$ and CS' is a conflict set.

In the context of FM testing and debugging, a diagnosis can be defined as follows (see Definition 3.4).

Definition 3.4 (Diagnosis). A diagnosis $\Delta = \{c_1..c_k\}$ is a subset of CF with $\forall t_i \in T$: consistent($CF - \Delta \cup \{t_i\}$). Δ is *minimal* if $\neg\exists\Delta' : \Delta' \subset \Delta$ and Δ' is a diagnosis.

Determining minimal diagnoses in FM testing. Diagnosis in the context of FM testing is based on the resolution of (minimal) conflicts induced by a set of test cases (see Definition 3.3). Let us assume the existence of a set of test cases $T = \{t_1 : payment = true \wedge standard = true, t_2 : energymanagement = false\}$ requiring the existence of FM configurations that include a *payment* feature combined with a *standard screen* as well as configurations that exclude the *energymanagement* feature. In the context of our example FM (see Figure 2.3), both test cases induce a corresponding conflict. (1) the test case $t_1 : payment = true \wedge standard = true$ induces the (singleton) conflict $CS_1 = \{c_9 : \neg(payment \wedge standard)\}$ and (2) the test case $t_2 : energymanagement = false$ induces the (singleton) conflict set $CS_2 = \{c_5 : energymanagent \Leftrightarrow smartwatch\}$.

Based on this information, we are able to construct a hitting set directed acyclic graph which helps to resolve conflicts in a structured fashion (see Figure 3.13). Since we have to deal with two singleton conflict sets, i.e., conflict sets containing exactly one element, the resulting HSDAG includes exactly one diagnosis (Δ_1).

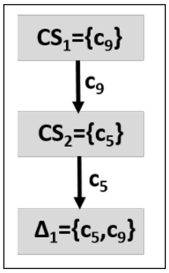


Fig. 3.13: Hitting Set Directed Acyclic Graph (HSDAG) for the conflict sets $CS_1 = \{c_9\}$ and $CS_2 = \{c_5\}$. The resulting minimal diagnosis is $\Delta_1 = \{c_5, c_9\}$.

Alternative algorithms for FM testing and debugging. In contrast to the discussed hitting set based approach, FM testing and debugging can also be implemented on the basis of direct diagnosis [15]. An approach to the related application of FastDiag is discussed in detail in Felfernig et al. [26].

3.5 Machine Learning for Conflict Detection and Diagnosis

In an inconsistent constraint set, there can be numerous conflicts and corresponding diagnoses. In this context, it is important to figure out diagnoses of relevance. In *interactive configuration settings*, diagnoses (repairs) proposed to users should only include preferences of low importance for a user. A user-individual (personalized) diagnosis can be determined on the basis of integrating machine learning concepts that help to infer preference importance, for example, on the basis of the preferences of previous user sessions [14, 38]. We refer to Chapter 4 for concepts helping to tackle such a *no solution could be found dilemma*.

When diagnosing an *unsatisfiable* FM, we are in a similar situation, i.e., we need to identify diagnoses of potential highest relevance. Table 3.4 shows a simple example of diagnosis ranking where Δ_i represent diagnoses determined for the constraints $c_1..c_5$ of an unsatisfiable FM.

Table 3.4: Example of a diagnosis ranking approach: diagnoses Δ_1 and Δ_4 have the highest accumulated constraint occurrence value (represented by the *score* value).

constraint	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	occurrence
c_1	×	×	×	×		4
c_2	×				×	2
c_3			×			1
c_4		×				1
c_5				×	×	2
score	6	5	5	6	4	

A simple approach to rank the Δ_i is to use the constraint occurrence in $\Delta_1..\Delta_5$ as an indicator, i.e., the more often a constraint is part of a diagnosis, the higher its relevance. Following this idea, Δ_1 and Δ_4 (both have an accumulated constraint occurrence value of 6) could be considered as the most relevant diagnoses. In this case, a recommendation for the designer of an FM could be to first look at the constraints contained in Δ_1 and Δ_4. For more details on different ways to rank diagnoses in unsatisfiable constraint sets we refer to Felfernig et al. [12, 14].

3.6 Discussion

In this chapter, we have discussed different topics related to the analysis of FMs. First, analysis operations help to figure out specific properties of an FM, for example, an analysis operation related to void features helps to identify all features that cannot be part of any configuration. In this context, we have introduced a differentiation between analysis operations in the need of a solver support (as it is the case with void features) and analysis operations without a need of a solver support (e.g., counting the features of an FM).

Analysis operations help to understand basic properties of FMs. In the following, we also introduced concepts that help developers of FMs to efficiently deal with inconsistencies in FMs. For example, when confronted with a non-satisfiable FM (no solution could be identified), conflict detection and diagnosis algorithms can be applied to identify the sources of an inconsistency (the conflicts) and to propose corresponding repairs (the diagnoses). In a working example, we showed how to determine diagnoses. Furthermore, we extended the application of diagnosis algorithms to the testing and debugging of FMs. In this context, test cases (represented as constraints) define the intended semantics of FMs. If an FM has a different semantics, the defined test case helps to induce conflicts in the set of FM constraints (CF) which can then be solved on the basis of model-based diagnosis.

As an orthogonal aspect in the context of FM analysis we introduced an algorithmic approach to redundancy detection in FMs. A constraint can be considered redundant if the semantics of an FM does not change even if we delete the constraint. In this context, we have introduced an algorithm for redundancy detection in FMs and showed its operation on the basis of a working example.

In the context of FM analysis, we regard the following aspects as *major issues for future work.*

Synthesis mechanisms for algorithm performance evaluation. Designing and developing conflict detection and diagnosis algorithms requires the structured provision of test feature (configuration) models which allow algorithm performance evaluation in a structured fashion [49]. For example, it should be possible to predefine the number of diagnoses, the number of conflict sets, and the corresponding cardinalities. On the basis of the generated FMs, more structured evaluations can be performed. In this context, large language models (LLMs) can also be regarded as a promising approach to support the generation of test models [19].

Parallelization of FM analysis. Existing parallelization architectures make it feasible to parallelize constraint reasoning as well as related conflict detection and diagnosis processes [30, 51]. A major challenge is to efficiently exploit parallelization architectures to significantly increase the efficiency of the mentioned operations. This is important specifically due to the increasing size and complexity of variability models (e.g., FMs). A further open issue is how to parallelize the identification of redundant constraints – no related algorithms exist up to now.

Cognitive issues in FM development and maintenance. Being able to identify the sources of an inconsistency also requires knowledge from cognitive psychology. For example, in order to identify the set of constraints responsible for the faulty semantics of an FM, knowledge about the cognitive complexity of individual FM elements should be exploited. The reason behind is that constraint structures which are less understandable have a higher probability of being the source of a faulty behavior. A simple example in this context is the logical implication $A \Rightarrow B$ – the corresponding implication $B \Leftarrow A$ with exactly the same semantics requires additional cognitive overheads [13].

Gamification-based conflict detection and diagnosis. Specifically in the context of teaching Artificial Intelligence (AI) topics, it is important to provide intuitive explanations of concepts and algorithms. In this context, gamification has shown to be an appropriate way of making complex algorithms more accessible for students. This idea should be applied in different AI-related settings and could thus contribute to increase the accessibility of different AI techniques and algorithms [11].

Analysis Operations focusing on FM Applications. If one wants to apply FMs in productive use, developers need to assure specific aspects of high relevance for successful configurator applications. For example, when introducing new features, it must be clear that these features are covered by the current infrastructure (e.g., is the production infrastructure capable of producing the configurations defined by customers) [32]. Furthermore, features offered to customers should not be too restrictive [41], i.e., narrow down the configuration space too much and thus leading to situations where no relevant configurations can be identified for a customer [33].

References

1. M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature model differences. In *Advanced Information Systems Engineering: 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings 24*, pages 629–645. Springer, 2012.
2. E. Bagheri and D. Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19:579–612, 2011.
3. D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank. Automated analysis in feature modelling and product configuration. In J. Favaro and M. Morisio, editors, *Safe and Secure Software Reuse*, pages 160–175, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
4. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615 – 636, 2010.
5. D. Benavides, C. Sundermann, S. Vill, K. Feichtinger, , J. A. Galindo, R. Rabiser, and T. Thüm. UVL: Feature Modelling with the Universal Variability Language. Technical report, Elsevier, 2024.
6. A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software & Systems Modeling*, 16(4):1049–1082, 2017.
7. A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Transactions on Engineering Management*, 54(1):41–56, 2007.
8. A. Felfernig. *AI Techniques for Software Requirements Prioritization*, chapter 2, pages 29–47. World Scientific, 2021.
9. A. Felfernig, D. Benavides, J. Galindo, and F. Reinfrank. Towards Anomaly Explanation in Feature Models. In *ConfWS-2013: 15th International Configuration Workshop (2013)*, volume 1128, pages 117–124, 2013.

10. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213 – 234, 2004.
11. A. Felfernig, M. Jeran, T. Ruprechter, A. Ziller, S. Reiterer, and M. Stettinger. Learning Games for Configuration and Diagnosis Tasks. In *Proceedings of the 17th International Configuration Workshop*, pages 111–114, 2015.
12. A. Felfernig, S. Reiterer, M. Stettinger, and J. Tiihonen. Intelligent Techniques for Configuration Knowledge Evolution. In *9th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 51–58, New York, NY, USA, 2015. ACM.
13. A. Felfernig, S. Reiterer, M. Stettinger, and J. Tiihonen. Towards Understanding Cognitive Aspects of Configuration Knowledge Formalization. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems(VaMoS '15)*, pages 117–123, New York, NY, USA, 2015. Association for Computing Machinery.
14. A. Felfernig, M. Schubert, and S. Reiterer. Personalized Diagnosis for Over-Constrained Problems. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence (IJCAI '13)*, pages 1990–1996. AAAI Press, 2013.
15. A. Felfernig, M. Schubert, and C. Zehentner. An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artif. Intell. Eng. Des. Anal. Manuf.*, 26(1):53–62, Feb. 2012.
16. A. Felfernig, R. Walter, J. Galindo, D. Benavides, M. Atas, S. Polat-Erdeniz, and S. Reiterer. Anytime Diagnosis for Reconfiguration. *Journal of Intelligent Inf. Sys.*, 51:161–182, 2018.
17. J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides. Exploiting the enumeration of all feature model configurations: A new perspective with distributed computing. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 74–78, 2016.
18. J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, 101(5):387–433, 2019.
19. J. A. Galindo, A. J. Dominguez, J. White, and D. Benavides. Large language models to generate meaningful feature model instances. In *27th ACM International Systems and Software Product Line Conference-Volume 1*, 2023.
20. R. Heradio, D. Fernandez-Amoros, J. A. Galindo, and D. Benavides. Uniform and scalable sat-sampling for configurable systems. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–11, 2020.
21. R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering*, 27(2):1–34, 2022.
22. J. Horcas, J. Galindo, M. Pinto, L. Fuentes, and D. Benavides. FM Fact Label: A Configurable and Interactive Visualization of Feature Model Characterizations. In *26th ACM International Systems and Software Product Line Conference - Volume B*, pages 42–45. ACM, 2022.
23. J. Horcas, M. Pinto, and L. Fuentes. Empirical Analysis of the Tool Support for Software Product Lines. *Software and Systems Modeling*, 22:377–414, 2023.
24. U. Junker. QuickXPlain: Preferred Explanations and Relaxations for over-Constrained Problems. In *AAAI'04*, page 167–172. AAAI Press, 2004.
25. M. Kowal, S. Ananieva, and T. Thüm. Explaining anomalies in feature models. *SIGPLAN Not.*, 52(3):132–143, 2016.
26. V. Le, A. Felfernig, M. Uta, D. Benavides, J. Galindo, and T. Tran. DirectDebug: Automated Testing and Debugging of Feature Models. In *IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 81–85. IEEE/ACM, 2021.
27. V. Le, A. Felfernig, M. Uta, T. Tran, and C. Silva. WipeOutR: Automated Redundancy Detection for Feature Models. In *26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, pages 164–169, New York, NY, USA, 2022. Association for Computing Machinery.
28. V. Le, T. Tran, and A. Felfernig. Consistency-Based Integration of Multi-Stakeholder Recommender Systems with Feature Model Configuration. In *26th ACM International Systems and Software Product Line Conference*, pages 178–182, New York, NY, USA, 2022. ACM.

29. V.-M. Le, A. Felfernig, T. N. T. Tran, and M. Uta. INFORMEDQX: Informed Conflict Detection for Over-Constrained Problems. In *38th Annual AAAI Conference on Artificial Intelligence*, pages 10616–10623, Vancouver, Canada, 2024. AAAI.
30. V.-M. Le, C. V. Silva, A. Felfernig, D. Benavides, J. Galindo, and T. N. T. Tran. Fastdiagp: An algorithm for parallelized direct diagnosis. In *AAAI'23/IAAI'23/EAAI'23*. AAAI Press, 2023.
31. M. H. Liffiton and K. A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40:1–33, 2008.
32. R. Lindohf, J. Krüger, E. Herzog, and T. Berger. Software Product-Line Evaluation in the Large. *Empirical Softw. Eng.*, 26(2), 2021.
33. S. Lubos, A. Felfernig, V.-M. Le, T. N. T. Tran, D. Benavides, J. A. Zamudio, and D. Garber. Analysis operations on the run: Feature model analysis in constraint-based recommender systems. In *27th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '23, pages 111–116, New York, NY, USA, 2023. Association for Computing Machinery.
34. M. Hentze and T. Pett and T. Thüm and I. Schaefer. Hyper Explanations for Feature-Model Defect Analysis. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery.
35. L. Marchezan, E. Rodrigues, W. K. G. Assunção, M. Bernardino, F. Basso, and J. Carbonell. Software Product Line Scoping: A Systematic Literature Review. *Journal of Systems and Software*, 186:111189, 2022.
36. R. Medeiros, O. Díaz, and D. Benavides. Unleashing the power of implicit feedback in software product lines: Benefits ahead. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 113–121, 2023.
37. J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake. An Overview on Analysis Tools for Software Product Lines. In *18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, pages 94–101. ACM, 2014.
38. A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V. Le, K. Pilsl, M. Enzelsberger, and T. Tran. An Overview of Machine Learning Techniques in Constraint Solving. *Journal of Intelligent Inf. Sys.*, 58(1):91–118, 2022.
39. R. Reiter. A Theory of Diagnosis From First Principles. *AI Journal*, 32(1):57–95, 1987.
40. J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, and D. Benavides. Selection of Software Product Line Implementation Components Using Recommender Systems: An Application to Wordpress. *IEEE Access*, 7:69226–69245, 2019.
41. D. Romero-Organvidez, D. Benavides, J.-M. Horcas, and M. T. Gómez-López. Variability in data transformation: towards data migration product lines. In *18th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 83–92, 2024.
42. A. Sree-Kumar, E. Planas, and R. Clarisó. Extracting software product line feature models from natural language specifications. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pages 43–53, 2018.
43. R. Stern, M. Kalech, A. Feldman, and G. Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI'12*, pages 828–834. AAAI, 2012.
44. C. Sundermann, T. Thüm, and I. Schaefer. Evaluating #SAT Solvers on Industrial Feature Models. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems*, VAMOS '20, New York, NY, USA, 2020. Association for Computing Machinery.
45. O. Tazl, C. Tafeit, F. Wotawa, and A. Felfernig. DDMin versus QuickXplain - An Experimental Comparison of two Algorithms for Minimizing Collections. In *SEKE 2022*, pages 481–486, 2022.
46. T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *31st International Conference on Software Engineering*, ICSE '09, page 254–264, USA, 2009. IEEE Computer Society.
47. T. Tran, A. Felfernig, and V. Le. An overview of consensus models for group decision-making and group recommender systems. *User Model User-Adap Inter*, 2023.

48. P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008. Agile Product Line Engineering.
49. M. Uta, A. Felfernig, V. Le, A. Popescu, T. Tran, and D. Helic. Evaluating Recommender Systems in Feature Model Configuration. In *25th ACM International Systems and Software Product Line Conference (SPLC 2021)*, pages 58–63. ACM, 2021.
50. M. Uta, A. Felfernig, G. Schenner, and J. Spöcklberger. Consistency-based Merging of Variability Models. In *21st International Configuration Workshop*, pages 9–12, Hamburg, Germany, 2019.
51. C. Vidal, A. Felfernig, J. Galindo, M. Atas, and D. Benavides. Explanations for Over-constrained Problems using QuickXPlain with Speculative Executions. *Journal of Intelligent Inf. Sys.*, 57(3):491–508, 2021.
52. J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. Automated Diagnosis of Feature Model Configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010. SPLC 2008.

Chapter 4
Interacting with Feature Model Configurators

Abstract In this chapter, we discuss different AI techniques that can be applied to support interactive FM configuration scenarios. We have in mind situations where the user of a FM configurator is in the need of support, for example, in terms of requiring recommendations and related explanations for feature inclusions or exclusions or recommendations of how to get out of an inconsistent situation. We show how to support feature selection on the basis of recommendation technologies and also show how to apply the concepts of conflict detection and model-based diagnosis to support users in inconsistent situations as well as in the context of reconfiguration.

4.1 Feature Model Configuration

Feature model (FM) configuration is often an interactive process where a user specifies her/his preferences regarding the given features [3, 4, 17, 18]. FM configurators support a.o. (1) checking the consistency of the articulated preferences ($CR \cup CF$ must be consistent), (2) recommending features, (3) explaining configurations, (4) finding ways to get out of situations where no solution can be identified by the configurator, and (5) reconfiguration, i.e., helping to adapt an already existing configuration in such a way that new user requirements are fulfilled (see Figure 4.1).

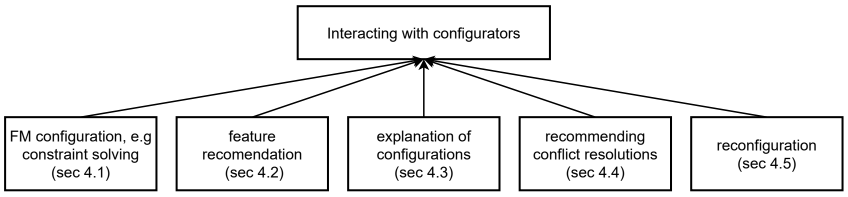

Interacting with configurators
FM configuration, e.g constraint solving (sec 4.1)
feature recomendation (sec 4.2)
explanation of configurations (sec 4.3)
recommending conflict resolutions (sec 4.4)
reconfiguration (sec 4.5)

Fig. 4.1: Interacting with FM configurators: basic AI techniques (ids in brackets refer to the corresponding subsection).

A. Felfernig et al., *Feature Models*, SpringerBriefs in Computer Science,
https://doi.org/10.1007/978-3-031-61874-1_4

The task of consistency checking and configuration completion is taken over by SAT solvers or constraint solvers. Especially in interactive settings, efficient response times are needed. In this context, solvers apply different types of search heuristics that can help to perform solution search in an efficient fashion. More advanced configuration approaches apply machine learning for determining a set of search heuristics that help to further improve the performance of the solver [33]. Furthermore, knowledge compilation approaches such as binary decision diagrams (BDDs) [1] help to further improve the performance of solution search. Beyond consistency checking and constraint reasoning, FM configuration also has to be able to deal with inconsistent situations where, for example, user preferences become inconsistent with the FM constraints. In such situations, conflict detection [22, 41] and diagnosis [34, 43] can support users in identifying the sources of an inconsistency and counteract correspondingly.[1]

Importantly, solution search, i.e., FM configuration, as well as inconsistency handling have to be *personalized* in the sense that depending on the preferences of the current user, different completions of the current configuration should be proposed and also different alternatives to resolve an inconsistency have to be provided. Such personalization services are crucial to support users in finding their preferred configuration and help to make the overall FM configuration process more efficient. In order to provide the mentioned personalization capabilities, different types of recommendation services have to be integrated with constraint solving (or other reasoning approaches such as SAT solving). In this chapter, we show different ways how such an integration can take place. In this context, we primarily focus on recommendation approaches that are based on supervised machine learning, i.e., a set of already completed *configuration sessions*[2] will be used to infer user-specific recommendations (see Table 4.1).

Following the definitions of an FM configuration and an FM configuration task (see the Definitions 2.6 and 2.8), we introduce a set of configurations that have already been completed in previous configuration sessions (see Table 4.1). In the line of Definition 2.6, each session entry in Table 4.1 is a consistent and complete FM configuration represented as an assignment set A. In the following, we discuss various ways in which the entries in Table 4.1 can be applied to personalize the interaction with an FM configurator.

4.2 Recommending Features

There are different reasons why users are not able to specify, i.e., include or exclude a specific feature within the scope of an FM configuration process. (1) users might not have the domain knowledge needed to decide about the inclusion of a specific feature – this might be the case with technical features or new features the user was

[1] For details see Chapter 3.

[2] Collection of valid configurations typically created by configurator users.

Table 4.1: Example: already completed configuration sessions. In the following, we will use these example session data to show the integration of different recommendation approaches into interactive configuration sessions (1 = inclusion, 0 = exclusion of a feature, ? = not specified yet).

$feature$	$Session_1$	$Session_2$	$Session_3$	$current$
smartwatch	1	1	1	1
screen	1	1	1	1
touch	1	1	1	0
standard	0	0	0	1
payment	0	1	0	?
gps	0	1	0	?
sportstracking	0	1	0	1
running	0	1	1	?
skiing	0	0	1	?
hiking	0	1	1	?
energymanagement	1	1	1	?
basic	0	0	0	?
advancedsolar	0	0	1	?

not confronted with up to now. (2) another explanation can be limited time resources, i.e., users do not have the time to specify every feature and for some features prefer to just rely on the recommendations provided by the FM configurator. (3) although users know the feature, they tend to accept recommendations provided by the configurator – this can happen due to the fact that users are risk-aware and want to avoid situations where they run into the risk of suboptimal configurations [2, 29]. An example of such an envisioned suboptimal configuration can be found in operating systems where specific system parameters could lead to suboptimal response times. In the following, we will discuss scenarios in which recommender systems [5, 10, 40] can be applied to support users in the completion of a configuration process.

A *recommender system* can be defined as *any system that guides a user in a personalized way to interesting or useful objects in a large space of possible options or that produces such objects as output* [9, 15]. For the purposes of our discussions, we distinguish between three types of recommender systems which are widely applied in different industrial contexts. (1) *collaborative filtering* is based on the idea of word-of-mouth promotion where the opinions of family members and friends are the major input for a recommendation. In the context of recommender systems, the role of family members and friends is taken over by so-called nearest neighbors (NNs) which are users with preferences similar to those of the current user. In the context of online sales platforms, collaborative filtering is primarily applied to predict the rating of a user for an item she/he has not consumed/seen up to now [19, 36]. (2) *content-based filtering* exploits previous item consumptions stored in the profile of the current user and tries to identify items that are similar to those that have been consumed in the past. (3) *group recommender systems* [8] determine recommendations for groups of users. In a first step, item ratings or items are specified by individual group members. Thereafter, the item preferences of group members are aggregated on the basis of an aggregation function. For example, *majority voting* recommends those items which are preferred by the majority of group members. In the following, we show how the mentioned recommendation approaches can be applied in configuration scenarios.

Collaborative filtering for recommending features. A basic approach to determine feature recommendations for individual users is to apply collaborative filtering [6, 7, 31, 39]. Formula 4.1 can be used to determine the similarity between two FM configuration sessions sa and sb where $F(sx)$ denotes the set of features already specified in session sx. In the example sessions of Table 4.1, Sessions 1–3 represent already completed configuration sessions, i.e., each feature is explicitly included or excluded, whereas the *current* session shows a subset of the features specified. Formula 4.1 is specified in such a way that similarities are only determined for those features specified in both sessions, i.e., sa and sb. Such a recommendation approach based on similarities between consumed items is also denoted as *memory-based collaborative filtering* [23].

$$sim(sa, sb) = \frac{|\{f \in F(sa) \cap F(sb) : sa.f = sb.f\}|}{|f \in F(sa) \cap F(sb)|} \tag{4.1}$$

In our working example (see Table 4.1), $sim(current, session_1) = \frac{2}{5} = 0.4$, $sim(current, session_2) = \frac{3}{5} = 0.6$, and $sim(current, session_3) = \frac{2}{5} = 0.4$. Consequently, $session_2$ is the nearest neighbor of the *current* session and the feature settings of this session could be used for recommending feature settings to the user of the *current* session. For example, for *gps* we could recommend feature inclusion since the user in $Session_2$ also decided to include feature *gps*. Continuing this idea, we could recommend the inclusion of the *payment* feature and further features. However, recommending the *payment* feature triggers an issue since the user in the *current* session has already selected the *standard screen* which incompatible with the *payment* feature. As a consequence, recommendations directly determined on the basis of collaborative filtering have to be checked for consistency with the constraints in the FM [10].

Integrating feature recommendations with variable domain orderings. In order to avoid the mentioned (potentially repeated) checking of the consistency of feature recommendations with the constraints in the FM, we can apply feature recommendations for determining a kind of variable domain ordering (heuristics) which can then be used by a constraint or SAT solver [32]. For example, if the inclusion of *payment* is recommended this would result in the definition of a variable domain ordering [1,0] instructing the solver to try to include the feature *payment* into the configuration (if possible). In the case of an inconsistency, backtracking would be triggered by the solver resulting in an exclusion of this feature. Following this strategy also helps to avoid inconsistencies.

Content-based filtering for recommending features. Content-based filtering [30] can be used in scenarios where the preferences of a user in terms of feature

Table 4.2: Example recommendation for the user in the *current* session (see Table 4.1). Instead of directly recommending feature settings, these settings can be included in corresponding variable domain orderings exploited by a constraint or SAT solver. In real-world contexts, the domain of the root feature (e.g., *smartwatch*) is assumed to be always *true* (1=true, 0=false).

feature	recommendation	domain ordering
smartwatch	1	[1,0]
screen	1	[1,0]
touch	0	[0,1]
standard	1	[1,0]
payment	1	[1,0]
gps	1	[1,0]
sportstracking	1	[1,0]
running	1	[1,0]
skiing	0	[0,1]
hiking	1	[1,0]
energymanagement	1	[1,0]
basic	0	[0,1]
advancedsolar	1	[1,0]

inclusions and exclusions from the past can be directly applied in future recommendation scenarios. The major difference between content-based filtering and collaborative filtering is that the former determines recommendations based on similarities between new items and items a user liked in the past whereas the latter focuses on determining recommendations based on similarities between the current user and related nearest neighbors. Content-based recommendation builds user profiles that collect in a compressed form information about a user's item consumptions in the past. In the context of FM recommendation, such a profile could simply include those features of configurations previously selected by the user. A simple similarity function determining the similarity between the profile (p) of the current user and a new configuration ($conf$) is shown in Formula 4.2.

$$sim(p, conf) = \frac{|\{f \in F(p) \cap F(conf) : p.f = conf.f\}|}{|f \in F(p) \cap F(conf)|} \tag{4.2}$$

When using content-based filtering in the context of FM configuration, the set of features from the user profile can be regarded as user requirements to be fulfilled by a new FM configuration. If an inconsistency occurs in this context, conflict detection and diagnosis can help to identify minimal sets of features to be adaptated such that a solution can be identified. In the context of our working example (*smartwatch* feature recommendation), basic properties of a *smartwatch* purchased in the past can be used to recommend similar smartwatches in upcoming smartwatch configuration scenarios. For example, the current *smartwatch* of a user gets damaged and the user is in the need of a new *smartwatch* or a new model of a *smartwatch* is released and should be primarily recommended to those users interested in similar smartwatches in the past. In our working example, we know that the user associated with configuration $Session_3$ prefers an *advancedsolar* management and a new smartwatch with a new generation of solar management features could be recommended. Analogously to the inclusion of collaborative filtering results into solver variable domain orderings, this approach can also be used in the context of content-based filtering: if a specific feature has been selected (deselected) in the past, the same feature should be selected (deselected) per default when building a new configuration.

Other example domains where content-based recommendation can be applied in the context of FM configuration are the configuration of round trips in the travel domain (e.g., based on the preferences of a person from previous travel packages, the destinations and services for the new travel package (configuration) can be recommended). On the basis of information of previous software packages installed for a user, new (similar) software packages and corresponding parametrizations can be recommended to the user when setting up a new operating system.

Enforcing configuration minimality. An important issue in the context of recommending feature inclusion (or exclusion) is to assure that only features are included that are really needed, i.e., to answer the question *what is the minimum set of additional features to be included in a configuration A such that all user requirements and FM constraints are taken into account*? For example, if we assume the existence of a partial configuration $A = \{smartwatch = true, screen = true, sportstracking = false, energymanagement = true\}$, it could be relevant to find a consistent and complete configuration A' with a minimum set of additional features included. Achieving such a goal can be relevant since solvers do not care about solution minimality and it can be relevant to support users in terms of indicating decision alternatives regarding a minimal and complete configuration. In the following, we show how such decision alternatives can be represented in terms of minimal conflict sets, and the corresponding conflict resolutions are represented as diagnoses, i.e., minimal sets of needed extensions to an existing configuration A such that the resulting configuration A' is consistent with $CR \cup CF$ and complete, i.e., each feature has an assigned setting indicating inclusion or exclusion.

We now introduce a set $\overline{A} = \{touch = false, standard = false, payment = false, gps = false, running = false, skiing = false, hiking = false, basic = false, advancedsolar = false\}$ with all those features f_i not specified in A assumed to be excluded, i.e., $f_i = false$. For the features in $\overline{A}$, we are able to determine all minimal conflict sets (see Chapter 3) with regard to the feature settings in A. The minimal conflict sets that can be identified in A' are the following: $CS_1 = \{touch = false, standard = false\}$, $CS_2 = \{basic = false, advancedsolar = false\}$. The diagnoses that can be derived from the identified minimal conflict sets are $\Delta_1 = \{touch = false, basic = false\}$, $\Delta_2 = \{touch = false, advancedsolar = false\}$, $\Delta_3 = \{standard = false, basic = false\}$, and $\Delta_4 = \{standard = false, advancedsolar = false\}$. These diagnoses indicate possible minimal extensions of the current partial configuration A to come up with a consistent and complete configuration A' [42]. If we choose, for example, Δ_1 as a possible minimal extension for A, this would result in a configuration $A' = \{smartwatch =$

$true, screen = true, sportstracking = false, energymanagement = true\} \cup \{touch = true, basic = true\} \cup \{standard = false, payment = false, gps = false, running = false, skiing = false, hiking = false, advancedsolar = false\}$ including (1) all feature settings of A, (2) the feature settings of Δ_1 in negated form, i.e., {touch=true,basic=true}, and (3) all features of $\overline{A} - \Delta_1$.

4.3 Explaining Configurations

There exist different ways of explaining a configuration to a user [20, 25]. Without any claim to completeness, in the following, we discuss basic explanation scenarios of potential interest for FM configuration.

"Why" explanations. A user might be interested in an explanation as to *why* specific features have been additionally included in a configuration [25] – this can be explained as follows: (1) Enumerating the user-specified preferences and indicating the relationship to the corresponding configuration (e.g., since you have selected *sportstracking*, the *gps* feature has been included since it is required for the support of *sportstracking*). In this context, we exploit a *requires* cross-tree constraint for explaining the inclusion of a specific feature. Kramer et al. [25] show how such explanations can be generated in a systematic fashion by proposing so-called *explanatory knowledge fragments* which specify explanation patterns for individual FM elements. For example, an explanation regarding a mandatory feature can always be concluded with the statement *in all configurations*. (2) The inclusion of features can also be explained on the basis of the used recommendation algorithm, for example, when applying a collaborative filtering algorithm, an explanation could refer to the preferences of the nearest neighbors (e.g., the *advanced solar* energy management feature has been selected by users with similar preferences). When applying content-based filtering, an explanation can refer to past preferences of the current user (e.g., the *advanced solar* energy management feature has been included since you included such a feature also in your previous purchases).

"How" explanations. A user might be interested in an explanation *how* a specific configuration has been determined – this can be explained as follows: (1) Explaining the sequence of constraints that were active when determining the current configuration (e.g., the energy management feature has been included since it is mandatory. Thereafter, a standard screen has been included since you did not want to include the payment feature, ...). (2) If different candidate configurations exist and those are ranked, for example, on the basis of a utility function [10], the approach to determine the corresponding interest dimensions could be explained to the user (e.g., configuration 1 has been ranked highest, since it has the highest utility with regard to the interest dimension *sustainability* which you have specified as the most important interest dimension). In group decision scenarios [26], such an explanation could refer to the applied aggregation strategy (e.g., *average* of the user ratings) used for ranking the configurations [35]. Consequently, such explanations are generated by using knowledge about the way solutions are determined [16].

"Why not" explanations. In FM configuration scenarios (and beyond), users can also end up in situations where no solution could be identified for the defined set of user preferences [13, 20, 28]. In such a setting, users could be interested in those requirement specifications that are responsible for the non-existence of a solution. A conflict (set) (see Section 4.4) indicates individual sets of requirements that induce an inconsistency – using this concept, users have to take a decision how to resolve each individual conflict. A diagnosis (also Section 4.4) indicates a way of how to change his/her requirements in one single step. Important to mention, specifically in constraint-solving and configuration-related AI research, conflicts as well as diagnoses are regarded as specific types of explanation [13, 20] (see also Chapter 3). In the following section, these concepts will be discussed in the context of identifying relevant conflict resolutions for inconsistent user requirements.

4.4 Predicting Relevant Conflict Resolutions

In Chapter 3, we have introduced different approaches that help to deal with inconsistencies between user requirements (CR) and the underlying FM constraints (CF). In this context, a conflict set CS is defined as $CS \subseteq CR$ such that *inconsistent*$(CS \cup CF)$. In our working example (see Figure 2.3), $CF = \{c_0..c_{10}\}$ can be derived from the FM in Figure 2.3. For the following example, we assume $CR = \{c_{11} : payment = true, c_{12} : sportstracking = true, c_{13} : standard = true, c_{14} : gps = false\}$.

In this example, the user requirements are inconsistent with the FM constraints, i.e., $CR \cup CF$ is inconsistent, which means that we have to activate conflict detection for figuring out the corresponding conflict sets [11, 12]. As discussed in Chapter 3, the first minimal conflict set derived is $CS_1 = \{c_{11}, c_{13}\}$, i.e., the user is interested in including the *payment* feature but has already included the *standard* screen feature which is in contradiction to the constraints defined in the FM (represented by the constraints in CF). An overview of the complete set of minimal conflict sets that can be derived from CR is shown in Table 4.3.

Table 4.3: Minimal conflict sets derived by QuickXPlain for $CR = \{c_{11} : payment = true, c_{12} : sportstracking = true, c_{13} : standard = true, c_{14} : gps = false\}$ and the FM constraints $CF = \{c_0..c_{10}\}$.

ID	min. conflict set (CS)
CS_1	$\{c_{11}, c_{13}\}$
CS_2	$\{c_{12}, c_{14}\}$

Conflicts as those shown in Table 4.3 can be resolved (1) interactively, i.e., a customer explicitly specifies accepted changes in his/her current requirements or (2) on the basis of additional knowledge about the importance weights of the given requirements. In interactive settings, users could be explicitly asked for selecting those preferences out of a conflict set, which they would accept to be adapted. For example, on the basis of the conflict sets contained in Table 4.3, a user could be asked to select which requirement (preference) to delete/change from CS_1 and thereafter the same information will be requested for conflict set CS_2. After having selected at least one requirement per conflict set, all conflicts are resolved (given conflict minimality – see also Chapter 3). Alternatively, we can assume the existence of a dataset of user-specific preferences from the past (see Table 4.4).

Table 4.4: Example set of completed configuration sessions where for each session *CR* represents an initially inconsistent set of requirements and *A* the finally consistent configuration as result of the configuration process. Furthermore, *REC* represents a recommendation for the change of the currently inconsistent set of user requirements. In this example, the features *smartwatch*, *screen*, and *energymanagement* are assumed to be mandatory.

$feature$	$Session_1$ (NN)		$Session_2$		$Session_3$		$current$	
	CR	A	CR	A	CR	A	CR	REC
touch	?	1	1	1	1	0	?	?
standard	**1**	0	**?**	0	**?**	0	**1**	0
payment	**1**	1	**0**	0	**0**	0	**1**	1
gps	**0**	0	**0**	1	**0**	1	**0**	0
sportstracking	**1**	0	**1**	1	**1**	1	**1**	0
running	?	0	?	1	1	1	?	?
skiing	?	0	?	1	0	0	?	?
hiking	?	0	?	1	0	0	?	?
basic	?	0	?	1	?	1	?	?
advancedsolar	?	1	?	0	?	0	?	?

In the dataset shown in Table 4.4, each session contains the set *CR* of originally defined user requirements (which are inconsistent) and the set *A* specifying the final consistent (and complete) configuration confirmed by the user. Given the requirements $CR = \{c_{11} : payment = true, c_{12} : sportstracking = true, c_{13} : standard = true, c_{14} : gps = false\}$ of the user in the current session, two minimal conflict sets are induced with regard to $CF = \{c_1..c_{10}\}$: $CS_1 = \{c_{11}, c_{13}\}$ and $CS_2 = \{c_{12}, c_{14}\}$. The most similar session (on a scale [*0*=not similar .. 1=very similar]) determined by Formula 4.1 compared to the current session is $session_1$: sim($current, session_1$)=$\frac{4}{4}$ = 1.0, sim($current, session_2$)=$\frac{2}{3}$ = 0.66, and sim($current, session_3$)=$\frac{2}{3}$ = 0.66.

Our goal now is to identify a minimal set of changes in CR such that consistency is restored with regard to the FM constraints CF. Following our discussions in Section 3.2, we are able to construct a hitting set directed acyclic graph (HSDAG) [34] that helps to identify minimal conflict resolutions (see Figure 4.2). In each step of the HSDAG construction, we can compare alternative conflict resolutions Δ_κ with regard to their conformance (*con*) with the configuration chosen by the nearest neighbor (NN) (see Formula 4.3).

$$con(\Delta_\kappa, A) = |\{f \in F(\Delta_\kappa) \cap F(A) : (\neg\Delta_\kappa.f = A.f)\}| \tag{4.3}$$

The conformance of a conflict resolution is specified by the number of changes proposed by Δ_κ which are in line with the feature settings in the nearest neighbor configuration A (the more changes conform with A, the better). Figure 4.2 shows how to determine a preferred diagnosis Δ_{pref} (more precisely, $\Delta_{3(pref)}$) by analyzing to which extent a set of conflict resolutions is in the line with the configuration A of the nearest neighbor (in our case, $Session_1$).[3]

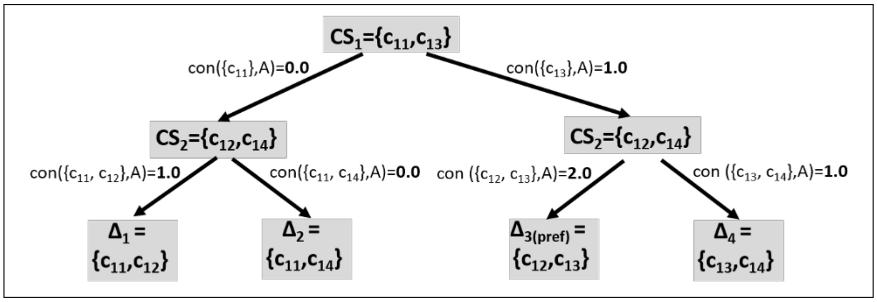


Fig. 4.2: Hitting Set Directed Acyclic Graph (HSDAG) for the conflict sets $CS_1 = \{c_{11} : payment = true, c_{13} : standard = true\}$ and $CS_2 = \{c_{12} : sportstracking = true, c_{14} : gps = false\}$. The search for a user-preferred diagnosis is guided by the conformance (*con*) measure (see Formula 4.3). The resulting preferred minimal diagnosis is $\Delta_{pref} = \{c_{12}, c_{13}\}$.

In the first step of the HSDAG construction (after having identified the first conflict set CS_1), we have to calculate the conformance of (1) c_{11} and (2) c_{13} with the feature selections in the configuration A of the nearest neighbor. This results in $conformance(\{c_{11}\}, A)$=0.0 and $conformance(\{c_{13}\}, A)$=1.0, i.e., only the conflict resolution regarding c_{13} is in line with A. After having resolved CS_1, the conflict set CS_2 is the only remaining conflict set. In terms conformance (see Formula 4.3), the diagnosis $\Delta_{pref} = \{c_{12}, c_{13}\}$ has the highest conformance value (2.0) and thus is the candidate for being presented as repair alternative to the user.

For the FM configurator user this means that it is recommended (by Δ_{pref}) to change the requirements regarding *sportstracking* and *standard*, i.e., to exclude these two features. The recommendation *REC* in Table 4.4 is the result of applying diagnosis Δ_{pref} to the original user requirements CR in the *current* session.

Direct diagnosis for personalized conflict resolution. Up to now, we have discussed different approaches that support the identification and resolution of conflict sets on the basis of the concepts of hitting set directed acyclic graphs [34]. Following

[3] In this example, there is exactly one preferred diagnosis, but it could be more.

such an approach means to (1) identify the relevant minimal conflicts and (2) to resolve those conflicts by deleting or adapting at least one element of each conflict set. A major drawback of this approach is that the determination of conflict sets is computationally expensive [22] which can trigger performance (response time) issues specifically in interactive settings. An alternative to the computation and resolution of minimal conflict sets is to apply the concepts of *direct diagnosis* which follows the idea of determining minimal diagnoses directly and omitting the step of identifying minimal conflict sets. A direct diagnosis algorithm for the determination of preferred minimal diagnoses is FastDiag [13, 28] which follows a divide-and-conquer based approach for the determination of minimal diagnoses.

The basic divide-and-conquer based approach of FastDiag has already been motivated in Chapter 3, i.e., diagnosis search in a constraint set $C = C_1 \cup C_2$ (assuming that both subsets include a nearly equal number of constraints) can be reduced by half if the constraints in one half, for example, C_1 appear to be consistent. Our motivation for providing more details on FastDiag in this chapter is that the algorithm allows the determination of preferred diagnoses, i.e., diagnoses that can be regarded as potentially relevant for a user. FastDiag can be activated with a consideration set C, i.e., the set of constraints in which a diagnosis needs to be identified and a constraint set B (the background knowledge which is assumed to not contain any diagnosis elements).

FastDiag is flexible since it allows to support scenarios where C is a set of customer requirements inconsistent with the FM constraints B but also scenarios where the FM configuration knowledge base is inconsistent. In this case, C represents the FM constraints and $B = \emptyset$. If $C \cup B$ is inconsistent, FastMSS is activated and returns a maximum satisfiable subset Ω of C. In FastDiag, Ω is used to derive the corresponding diagnosis Δ by building the complement $= C - \Omega$, i.e., $\Delta = C - \Omega$. If $C \cup B$ is consistent, no diagnosis process needs to be activated and $\emptyset$ is returned by FastDiag (see Algorithm 4).[4]

Algorithm 4 FastDiag$(C = \{c_1..c_n\}, B) : \Delta$

```
1: if Inconsistent($C \cup B$) then
2:     return($C$-FastMSS($\emptyset, C, B$))
3: else
4:     return($\emptyset$)
5: end if
```

The search for a maximum satisfiable subset (MSS) Ω in C is performed by FastMSS (see Algorithm 5) where Ω satisfies the following property: $\nexists \Omega' \supset \Omega : MSS(\Omega')$, i.e., no superset of a maximum satisfiable subset can be a maximum satisfiable subset (MSS). If $C \cup B$ is consistent, the whole set C is consistent and can be regarded as part of the maximum satisfiable subset. In this context, the parameter δ is used to avoid redundant consistency checks of $C \cup B$. If $|C| = 1$, it can be

[4] Algorithm 4 is a variant of FastDiag introduced in [13].

assumed to be part of a diagnosis since otherwise it would have been returned as a consistent constraint earlier. In every other case ($|C| > 1$), diagnosis search has to be continued in a divide-and-conquer fashion, i.e., C is divided into the two subsets C_1 and C_2 resulting in two further activations of FastMSS – the first one for checking C_1 for further Ω elements and the second one for checking for Ω elements in C_2 (Ω_2 includes MMS identified in C_1). All MSS elements, i.e., $\Omega_1 \cup \Omega_2$, that could be identified on a specific recursive level of FastMSS are returned to the previous activation level.

Algorithm 5 FastMSS$(\delta, C = \{c_1..c_n\}, B) : \Omega$

```
1: if $\delta \neq \emptyset \wedge$ IsConsistent$(C \cup B)$ then
2:     return$(C)$
3: end if
4: if $|C| = 1$ then
5:     return$(\emptyset)$
6: end if
7: $k = \lfloor \frac{n}{2} \rfloor$
8: $C_1 \leftarrow c_1...c_k; C_2 \leftarrow c_{k+1}...c_n;$
9: $\Omega_2 \leftarrow$ FastMSS$(C_1, C_1, B)$;
10: $\Omega_1 \leftarrow$ FastMSS$(C_1 - \Omega_2, C_2, B \cup \Omega_2)$;
11: return$(\Omega_1 \cup \Omega_2)$
```

Assuming the customer requirements $CR = \{c_{11} : payment = true, c_{12} : sportstracking = true, c_{13} : standard = true, c_{14} : gps = false\}$, we now want to sketch the execution of Algorithms 4–5 on the basis of our working example (see Figure 4.3). In order to be applicable for FastDiag, we have to define the contents of C and B. In our working example, we can assume the consistency of the FM and the corresponding FM constraints, i.e., we can focus diagnosis search on CR where we assume $c_0 : smartwatch \wedge screen \wedge energymanagement$.

FastMSS is based on depth-first search where in every case the left branch is responsible for determining Ω_2 whereas the right branch of the search tree is responsible for determining Ω_1. In our example (see Figure 4.3), the determined maximum satisfiable subset (MSS) is $\Omega = \{c_{11}, c_{12}\}$ indicating that $\Omega \cup B$ is consistent and $\nexists \Omega' \supset \Omega : MSS(\Omega')$. The complement (the diagnosis) Δ of $\Omega = \{c_{11}, c_{12}\}$ is $C - \Omega$ which results in $\Delta = \{c_{13}, c_{14}\}$. As mentioned, FastDiag allows for the determination of preferred diagnoses. More precisely, diagnoses can differ depending on the original ordering of the constraints in the consideration set C [13].

In our working example, we assumed the constraint ordering $[c_{11}, c_{12}, c_{13}, c_{14}]$ assuming that constraints at the beginning have the highest importance and constraint importance decreases with a corresponding lower ranking in the list, i.e., in the given example, constraint c_{14} has the lowest importance. In this context, we assume that constraints with a lower importance have a higher probability of being accepted by the user as a diagnosis element. If we would change the order of our example constraints in C to $[c_{14}, c_{13}, c_{12}, c_{11}]$, the diagnosis returned by FastDiag would

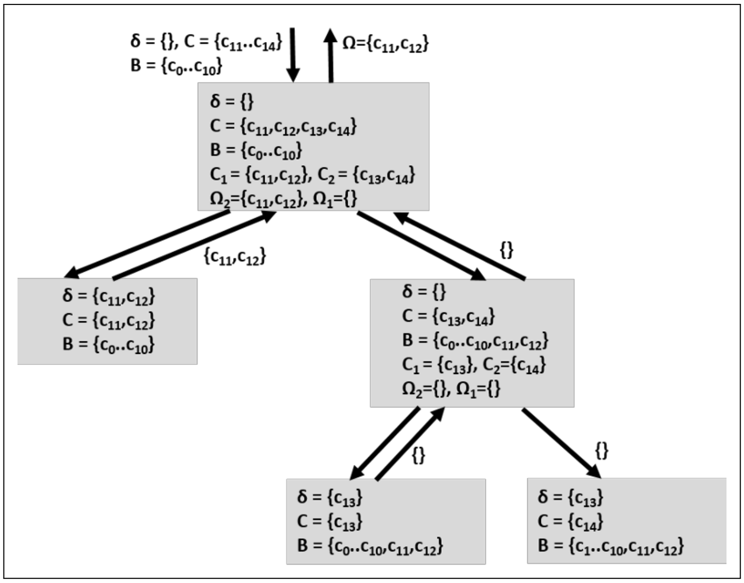


Fig. 4.3: Execution trace of FastMSS on the basis of $C = \{c_{11}..c_{14}\}$ and $B = \{c_0..c_{10}\}$ resulting in the maximum satisfiable subset (MSS) $\Omega = \{c_{11}, c_{12}\}$. The corresponding preferred diagnosis returned by FastDiag is $\Delta = \{c_{13}, c_{14}\}$.

be $\Delta = \{c_{11}, c_{12}\}$. The diagnoses returned by FastDiag are subset-minimal (see Chapter 3) but not necessarily of minimal cardinality. In diagnosis scenarios where minimal cardinality is required, we recommend the standard approach of determining minimal conflict sets [22, 27] and a corresponding conflict resolution based on hitting set directed acyclic graphs (HSDAGs) [34].

4.5 Reconfiguration

In situations where an FM configuration has already been completed, after-configuration tasks can become relevant. Such tasks can be summarized as *reconfiguration tasks* [14] with the goal to adapt an existing configuration in such a way that new requirements are fulfilled. In the following, we discuss two basic scenarios which are (1) the estimation *which new feature should be recommended* with regard to a specific configuration and (2) a situation where a FM configuration has to be adapted in order to take into account a *new set of user requirements*. We show how to apply the concepts of matrix factorization [24] to perform such prediction tasks.

Matrix factorization for new feature recommendations. Table 4.5 sketches a basic reconfiguration setting where the major task is to predict for a new feature *musicplay* whether this feature should be recommended to users u_i who have already completed an FM configuration.[5] In the following, we will show how this task can be completed on the basis of *matrix factorization* which is a widely used model-based collaborative filtering approach [24] (in contrast to memory-based collaborative filtering which has been used in Section 4.2). In this example, we assume that users u_1 and u_2 have already integrated the *musicplay* feature in their smartwatch (e.g., on the basis of a software upgrade), and user u_4 was not interested in this upgrade.

Table 4.5: The task of predicting the relevance of a new feature *(mu)sicplay* for different users u_i who have already completed a configuration. The symbol ? in the matrix (T) indicates the task to predict whether the new feature should be recommended to the user u_i (customer).

	sm	sc	to	st	pa	gp	sp	ru	sk	hi	en	ba	ad	mu
u_1	1	1	1	0	1	1	1	1	1	1	1	0	1	1
u_2	1	1	1	0	0	1	1	1	0	0	1	0	1	1
u_3	1	1	0	1	0	1	0	0	0	0	1	1	0	?
u_4	1	1	1	0	1	0	0	0	0	0	1	1	0	0
u_5	1	1	0	1	0	0	0	0	0	0	1	1	0	?
u_6	1	1	0	1	0	0	0	0	0	0	1	1	0	?
u_7	1	1	0	1	0	0	0	0	0	0	1	1	0	?
u_8	1	1	0	1	0	1	1	1	0	1	1	0	1	?

The entries in Table 4.5 can help to predict the relevance of individual new features for users. In our example, an additional feature *musicplay* has been integrated which is a software update that allows the activation of music sharing via smartwatch. In a marketing context, it is important to know which users (of a potentially large group of users) could be interested in this additional feature. A similar scenario is one where users who already purchased a smartwatch x could be interested in a new version of x due to the mentioned upgrade.

The prediction of the relevance of a new feature can be supported, for example, on the basis of memory-based collaborative filtering [23] where relevance estimation is implemented by simply analyzing the preferences of similar users with regard to new features. In contrast to the previously discussed scenarios, the recommendation of new features and – more generally – reconfiguration scenarios can often be handled in an *offline* fashion which makes these scenarios more accessible to model-based collaborative filtering approaches such as *matrix factorization* (MF) [24]. The overall idea of matrix factorization is to optimize a set of so-called *interest dimensions* (in machine learning contexts often denoted as *hidden features*) in such a way that user-individual preferences can be estimated with a high prediction quality.

Using matrix factorization, the entries in Table 4.5 (the matrix T) can be reconstructed on the basis of *dimensionality reduction* which follows the approach

[5] For better readability, in this example, we apply the following abbreviations for feature names: {(sm)artwatch, (sc)reen, (to)uch, (st)andard, (pa)yment, (gp)s, (sp)ortstracking, (ru)nning, (sk)iing, (hi)king, (en)ergymanagement, (ba)sic, (ad)vancedsolar, (mu)sicplay}.

of learning two low-dimensional matrices (UA and AF representing the machine learning model) that can be used to derive a matrix $T' \approx T$, i.e., T' can be regarded as an approximation of T. Following this approach, we are able to generalize from determining recommendations based on individual user preferences to a machine learning model based on dimensionality reduction which means that abstract *interest dimensions* (in machine learning terms denoted as *features*) are learned and used to predict item preferences of individual users.

For demonstration purposes, we construct the matrices UA (Table 4.6) and AF (Table 4.7) including the *hidden features* (interest dimensions) dim_1 and dim_2. These dimensions are denoted as *hidden*, since the underlying machine learning (matrix factorization) algorithm is not aware of the semantics of these features. We chose to include two dimensions, however, in real-world application contexts the number of such hidden features could be much higher. The role of such hidden features can be best explained by example interest dimensions with a corresponding semantic, i.e., dim_1 could (as said, we do not know) represent the interest dimension *flexibility* (i.e., the more features are included the better) and dim_2 could represent the dimension *simplicity* (i.e., the less features included, the better).

If we use matrix factorization for learning the user × feature (interest dimension) relationship and the interest dimension × feature (of the FM) relationship, the corresponding table entries are learned, i.e., do not have to be filled out manually. In this context, the learning goal is to optimize (maximize) the *similarity* between T and T' where T' is the result of applying a matrix multiplication of UA • AF – Table 4.8 is the result of a corresponding matrix multiplication applied to our example Tables 4.6 and 4.7. In this context, the feature *musicplay* (*mu*) has a high estimate for the users u_1, u_2, and u_8 and a low estimate for all other users. The entry in Table 4.8 also confirms (predicts) a low interest of user u_4 in the new *mu* feature.

Table 4.6: Matrix UA representing relationships between between the users $u_1..u_8$ and interest dimensions (hidden features dim_1 and dim_2).

user	dim_1	dim_2
$u1$	1.00	0.00
$u2$	0.92	0.01
$u3$	0.10	0.98
$u4$	0.18	0.74
$u5$	0.00	1.00
$u6$	0.00	1.00
$u7$	0.00	1.00
$u8$	0.83	0.25

Again, we have to emphasize that when applying matrix factorization [24], i.e., learning the interest dimension/user relationships, the corresponding machine learning features are *hidden*, i.e., do not have a meaning. In other words, we do not know exactly in which way the hidden features used by matrix factorization have a direct

Table 4.7: Matrix AF representing relationships between interest dimensions (hidden features) and selected features of our example FM .

dimension	sm	sc	to	st	pa	gp	sp	ru	sk	hi	en	ba	ad	mu
dim_1	1.00	1.00	0.80	0.20	0.44	1.00	1.00	1.00	0.39	0.71	1.00	0.00	1.00	1.00
dim_2	0.99	0.99	0.09	0.91	0.12	0.17	0.00	0.00	0.00	0.00	0.99	1.00	0.00	0.00

Table 4.8: Matrix T' as a result of a matrix multiplication UA • AF. The feature *musicplay* (mu) appears to be potentially relevant for users u_1, u_2, and u_8.

	sm	sc	to	st	pa	gp	sp	ru	sk	hi	en	ba	ad	mu
u_1	1.00	1.00	0.80	0.20	0.44	1.00	1.00	1.00	0.3892	0.7114	1.00009	0.0002	0.9999	**0.9999**
u_2	0.93	0.93	0.74	0.20	0.40	0.92	0.92	0.92	0.3575	0.65365	0.9312	0.01270	0.9187	**0.9187**
u_3	1.07	1.07	0.16	0.91	0.16	0.27	0.10	0.10	0.0372	0.06800	1.0677	0.97834	0.0955	0.0955
u_4	0.91	0.91	0.21	0.71	0.17	0.30	0.18	0.18	0.0682	0.12477	0.9114	0.74079	0.1753	0.1753
u_5	0.99	0.99	0.09	0.91	0.12	0.17	0.00	0.00	0.0001	0.00017	0.99396	0.99999	0.0002	0.0002
u_6	0.99	0.99	0.09	0.91	0.12	0.17	0.00	0.00	0.0001	0.00017	0.99395	0.99999	0.0002	0.0002
u_7	0.99	0.99	0.09	0.91	0.12	0.17	0.00	0.00	0.0001	0.00017	0.9939	1.0000	0.0002	0.0002
u_8	1.07	1.07	0.68	0.39	0.39	0.87	0.83	0.83	0.3221	0.5888	1.07320	0.2471	0.8276	**0.8276**

relationship to the (explicitly defined) features used in our working example. Consequently, machine learning approaches such as matrix factorization provide help in terms of automatically learning user × item preferences but come along with the disadvantage of a low degree of explainability due to a lack of semantic knowledge about user × item relationships.

FM reconfiguration. In the previous section, we already took a look at a simple reconfiguration scenario focusing on analyzing a potential need of extending the current FM configuration with the *inclusion of a new feature*. Beyond that, there also exist scenarios where feature settings of an existing configuration A have to be adapted in order to be able to take into account a *new set of user requirements* (CR). On the software level of a smartwatch, the inclusion of specific additional features could trigger a need of reconfiguration [14, 21]. In our *smartwatch* example, the inclusion of additional features, for example, additional software components supporting *sportstracking* could trigger a need of changing also other settings in the existing FM configuration A.

Let us assume the existence of a configuration $A = \{smartwatch = true, screen = true, touch = false, standard = true, payment = false, gps = false, sportstracking = false, running = false, skiing = false, hiking = false, energymanagement = true, basic = true, advancedsolar = false\}$. The user now changes his/her mind and wants to include the *payment* feature. Since *payment* excludes a *standard screen*, the (singleton) requirement $payment = true$ induces an inconsistency in the feature settings of A. In our example, two conflict sets are induced which are $CS_1 = \{standard = true\}$ and $CS_2 = \{touch = false\}$. For CS_1 and CS_2, there exists one related diagnosis which is $\Delta = \{standard =$

$true, touch = false\}$ indicating that $standard = true$ has to be replaced with $standard = false$ and $touch = false$ has to be replaced with $touch = true$ in A' resulting in a corresponding reconfiguration A'.

Due to the binary domain of individual features, a reconfiguration can be directly derived from a diagnosis Δ. A reconfiguration A' is an adaptation of the original configuration A in such a way that the new requirements CR are consistent with the feature settings in A'. In this context, the setting of those features remains the same which are included in A but are not part of Δ. Vice-versa, feature elements of Δ have to be deleted from A and included in negated form into the new configuration A' which itself represents the reconfiguration. Equation 4.4 represents a construction rule for each setting of a feature f in the new configuration (reconfiguration) A' where $f(A')$ denotes the feature setting of feature f in A' (e.g., $touch(A') = true$), $f(A)$ denotes the feature setting of feature f in the original configuration A (e.g., $touch(A) = false$), and $\overline{f}(\Delta)$ denotes the new feature setting of feature f in A (e.g., $\overline{touch}(\Delta)$ represents the new feature setting $touch = true$).

$$f(A') = \begin{cases} f(A), \text{if} f = X \notin \Delta \\ \overline{f}(\Delta), \text{otherwise} \end{cases} \tag{4.4}$$

Recommending reconfigurations. Recommendations for reconfigurations can be determined in a fashion similar to the recommendation of conflict resolutions (see Section 4.4). As discussed, a set of new requirements (within the scope of a reconfiguration scenario) can induce a conflict in the current configuration A. The identified conflicts can be resolved on the basis of the concepts of model-based diagnosis [34]. Given a dataset which includes the original configuration A as well as the corresponding reconfiguration A', a collaborative filtering approach could be applied by (1) identifying (in the dataset) a configuration A which is similar to the configuration of the current user and (2) to guide conflict resolution in such a way that the chosen resolutions lead to a reconfiguration A' which is as similar as possible to the feature settings in the reconfiguration of the nearest neighbor.

4.6 Discussion

In this chapter, we have discussed different topics in the context of supporting FM configuration in interactive scenarios, i.e., a user is interacting with a configurator with the goal to build a complete and consistent FM configuration. FM configuration can become a tedious task since users might not always have detailed domain knowledge resulting in situations where some of the features could not be specified or get specified in a suboptimal fashion. Furthermore, a complete specification of all features might simply not be possible due to size of the configuration model.

In order to provide a better support for users in interactive FM configuration scenarios, we have shown how different approaches from machine learning and recommender systems can be applied to predict the relevance of inclusion or exclusion of specific features. In this context, we discussed (1) approaches to recommend feature inclusion or exclusion, (2) approaches to the recommendation of adaptations of feature preferences in inconsistent situations, and (3) approaches to support reconfiguration scenarios, for example, in terms of determining minimal sets of adaptations needed for already existing configurations such that a new set of user requirements can be taken into account.

In the context of the topic of interacting with FM configurators, we regard the following aspects as major issues for future work.

Search heuristics beyond variable domain orderings. We discussed different approaches to support the recommendation of feature inclusion or exclusion. In this context, we have sketched ways to integrate such a recommendation task directly into the variable value ordering of a solver. The inclusion of variable domain orderings into solver search heuristics can be regarded as a first step towards *accuracy-aware FM configuration*, however, further approaches, for example, variable ordering and the generation of dynamic search heuristics, i.e., search heuristics defined during solver runtime, have to be analyzed in detail.

Integration of machine learning with constraint reasoning. In the line of the topic of integrating search heuristics with recommendation, a more general issue is the integration of machine learning with constraint reasoning [33]. For example, it is important to further improve the predictive quality of recommendation services. This can be achieved by analyzing different possibilities to integrate domain knowledge into the machine learning process, for example, by explicitly encoding domain constraints in a neural network.

Cognitive issues in interactive configuration. There are issues located far beyond technical issues of interactive FM configuration. In many cases, configuration is a highly interactive process (with the exception of batch configuration scenarios) where users are interacting with a configurator with the goal to build a consistent configuration entailing user-relevant features. In this context, it is important to take into account theories of human decision making to be able to better understand how to best support configurator users [38].

Group-based configuration. In contrast to single-user scenarios, there are also many scenarios where a configuration task has to be completed by a group of users [26]. In this context, a group of users has to achieve consensus with regard to the inclusion or exclusion of a specified set of features [37]. Furthermore, conflicts (in the case of inconsistent requirements among different group members) have to be resolved in a way somehow acceptable for all group members. New user interfaces supporting group decision making in the context of FM configuration as well as new recommendation and diagnosis algorithms have to be developed to provide an efficient user support in such contexts.

References

1. S. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.
2. M. Atas, A. Felfernig, S. Polat-Erdeniz, A. Popescu, T. Tran, and M. Uta. Towards Psychology-Aware Preference Construction in Recommender Systems: Overview and Research Issues. *J. Intell. Inf. Syst.*, 57(3):467–489, 2021.
3. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615 – 636, 2010.
4. D. Benavides, C. Sundermann, S. Vill, K. Feichtinger, , J. A. Galindo, R. Rabiser, and T. Thüm. UVL: Feature Modelling with the Universal Variability Language. Technical report, Elsevier, 2024.
5. R. Burke, A. Felfernig, and M. Göker. Recommender Systems: An Overview. *AI Magazine*, 32(3):13–18, 2011.
6. M. Ekstrand, J. Riedl, and J. Konstan. Collaborative Filtering Recommender Systems. *Foundations and Trends in Human-Computer Interaction*, 4(2):81–173, 2011.
7. A. Falkner, A. Felfernig, and A. Haag. Recommendation Technologies for Configurable Products. *AI Magazine*, 32(3):99–108, 2011.
8. A. Felfernig, L. Boratto, M. Stettinger, and M. Tkalcic. *Group Recommender Systems: An Introduction*. Springer Publishing Company Inc., 2nd edition, 2024.
9. A. Felfernig and R. Burke. Constraint-based recommender systems: Technologies and research issues. In *Proceedings of the 10th International Conference on Electronic Commerce*, ICEC '08, New York, NY, USA, 2008. Association for Computing Machinery.
10. A. Felfernig, V. Le, A. Popescu, M. Uta, T. Tran, and M. Atas. An Overview of Recommender Systems and Machine Learning in Feature Modeling and Configuration. In *15th Intl. Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '21. ACM, 2021.
11. A. Felfernig, M. Mairitsch, M. Mandl, M. Schubert, and E. Teppan. Utility-based repair of inconsistent requirements. In B.-C. Chien, T.-P. Hong, S.-M. Chen, and M. Ali, editors, *Next-Generation Applied Intelligence*, pages 162–171, Berlin, Heidelberg, 2009. Springer.
12. A. Felfernig, M. Schubert, and S. Reiterer. Personalized Diagnosis for Over-Constrained Problems. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence (IJCAI '13)*, pages 1990–1996. AAAI Press, 2013.
13. A. Felfernig, M. Schubert, and C. Zehentner. An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artif. Intell. Eng. Des. Anal. Manuf.*, 26(1):53–62, Feb. 2012.
14. A. Felfernig, R. Walter, J. Galindo, D. Benavides, M. Atas, S. Polat-Erdeniz, and S. Reiterer. Anytime Diagnosis for Reconfiguration. *Journal of Intelligent Inf. Sys.*, 51:161–182, 2018.
15. A. Felfernig, M. Wundara, T. Tran, V.-M. Le, S. Lubos, and S. Polat-Erdeniz. Sports recommender systems: Overview and research issues. *Journal of Intelligent Inf. Sys.*, 2024.
16. G. Friedrich and M. Zanker. A Taxonomy for Generating Explanations in Recommender Systems. *AI Magazine*, 32(3):90–98, 2011.
17. J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, 101(5):387–433, 2019.
18. J. A. Galindo, J.-M. Horcas, A. Felfernig, D. Fernandez-Amoros, and D. Benavides. Flama: A collaborative effort to build a new framework for the automated analysis of feature models. In *27th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '23, pages 16–19, New York, NY, USA, 2023. Association for Computing Machinery.
19. C. Gomez-Uribe and N. Hunt. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Transactions on Management Information Systems*, 6(4), 2016.
20. S. Gupta, B. Genc, and B. O'Sullivan. Explanation in Constraint Satisfaction: A Survey. In Z. Zhi-Hua, editor, *13th International Joint Conference on Artificial Intelligence*, pages 4400–4407. International Joint Conference on Artificial Intelligence Organization, 2021.
21. M. Janota, G. Botterweck, and J. Marques-Silva. On Lazy and Eager Interactive Reconfiguration. In *8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14)*, pages 8:1–8:8, Sophia Antipolis, France, 2014. ACM.

22. U. Junker. QuickXPlain: Preferred Explanations and Relaxations for over-Constrained Problems. In *AAAI'04*, page 167–172. AAAI Press, 2004.
23. J. Konstan, B. Miller, J. Herlocker, L. Gordon, and J. Riedl. GroupLens: Applying Collaborative Filtering to Usenet News. *Communications of the ACM*, 40(3):77–87, 1997.
24. Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*, 42(8):30–37, 2009.
25. D. Kramer, C. Sauer, and T. Roth-Berghofer. Towards Explanation Generation using Feature Models in Software Product Lines. In G. Nalepa and J. Baumeister, editors, *9th Workshop on Knowledge Engineering and Software Engineering (KESE9)*, volume 1070. CEUR, 2013.
26. V. Le, T. Tran, and A. Felfernig. Consistency-Based Integration of Multi-Stakeholder Recommender Systems with Feature Model Configuration. In *26th ACM International Systems and Software Product Line Conference*, pages 178–182, New York, NY, USA, 2022. ACM.
27. V.-M. Le, A. Felfernig, T. N. T. Tran, and M. Uta. INFORMEDQX: Informed Conflict Detection for Over-Constrained Problems. In *38th Annual AAAI Conference on Artificial Intelligence*, pages 10616–10623, Vancouver, Canada, 2024. AAAI.
28. V.-M. Le, C. V. Silva, A. Felfernig, D. Benavides, J. Galindo, and T. N. T. Tran. Fastdiagp: An algorithm for parallelized direct diagnosis. In *AAAI'23/IAAI'23/EAAI'23*. AAAI Press, 2023.
29. M. Mandl, A. Felfernig, J. Tiihonen, and K. Isak. Status Quo Bias in Configuration Systems. In K. Mehrotra, C. Mohan, J. Oh, P. Varshney, and M. Ali, editors, *Modern Approaches in Applied Intelligence*, pages 105–114. Springer, 2011.
30. M. Pazzani and D. Billsus. Content-Based Recommendation Systems. In *The Adaptive Web: Methods and Strategies of Web Personalization*, pages 325–341. Springer, 2007.
31. J. Pereira, P. Matuszyk, S. Krieter, S. piliopoulou, and G. Saake. Personalized Recommender Systems for Product-Line Configuration Processes. *Comput. Lang. Syst. Struct.*, 54(C):451–471, 2018.
32. S. Polat-Erdeniz, A. Felfernig, R. Samer, and M. Atas. Matrix Factorization Based Heuristics for Constraint-Based Recommenders. In *34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1655–1662, New York, NY, USA, 2019. ACM.
33. A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V. Le, K. Pilsl, M. Enzelsberger, and T. Tran. An Overview of Machine Learning Techniques in Constraint Solving. *Journal of Intelligent Inf. Sys.*, 58(1):91–118, 2022.
34. R. Reiter. A Theory of Diagnosis From First Principles. *AI Journal*, 32(1):57–95, 1987.
35. J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, and D. Benavides. Selection of Software Product Line Implementation Components Using Recommender Systems: An Application to Wordpress. *IEEE Access*, 7:69226–69245, 2019.
36. B. Smith and G. Linden. Two Decades of Recommender Systems at Amazon.Com. *IEEE Internet Computing*, 21(3):12–18, may 2017.
37. T. Tran, A. Felfernig, and V. Le. An overview of consensus models for group decision-making and group recommender systems. *User Model User-Adap Inter*, 2023.
38. T. Tran, A. Felfernig, and N. Tintarev. Humanized Recommender Systems: State-of-the-Art and Research Issues. *ACM Trans. Interact. Intell. Syst.*, 11(2), 2021.
39. M. Uta, A. Felfernig, D. Helic, and V. Le. Accuracy- and Consistency-Aware Recommendation of Configurations. In *26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, page 79–84. ACM, 2022.
40. M. Uta, A. Felfernig, V. Le, T. Tran, D. Garber, S. Lubos, and T. Burgstaller. Knowledge-based Recommender Systems: Overview and Research Directions. *Frontiers in Big Data*, 7:1–30, 2024.
41. C. Vidal, A. Felfernig, J. Galindo, M. Atas, and D. Benavides. Explanations for Over-constrained Problems using QuickXPlain with Speculative Executions. *Journal of Intelligent Inf. Sys.*, 57(3):491–508, 2021.
42. C. Vidal-Silva, J. G. J. Giráldez-Cru, and D. Benavides. Automated Completion of Partial Configurations as a Diagnosis Task Using FastDiag to Improve Performance. In *Intelligent Systems in Industrial Applications (ISMIS 2020)*, volume 949, pages 81–85. Springer, 2021.
43. J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. Automated Diagnosis of Feature Model Configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010. SPLC 2008.

Chapter 5
Tools and Applications

Abstract Feature Models (FMs) are not only an active scientific topic but they are supported by many tools from industry and academia. In this chapter, we provide an overview of example feature modelling tools and corresponding FM configurator applications. In our discussion, we first focus on different tools supporting the design of FMs. Thereafter, we provide an overview of tools that also support FM analysis. Finally, we discuss different existing FM configurator applications.

5.1 Tool and Application Landscape

We will now show how the concepts discussed in Chapters 2–4 are integrated into real-world systems (ranging from industrial applications to research-driven prototypes). Importantly, all of the discussed systems cover some subsets of the concepts discussed in the previous chapters.

Without claiming to be complete, we provide an overview of example tools and applications. In Section 5.2, we provide an overview of example feature modelling tools which are of great importance for different kinds of variability management processes [11, 25, 99]. In this context, we discuss the functionalities provided by those tools and provide insights into the corresponding graphical user interfaces. In Section 5.3, we focus on the way different types of FM analysis operations are included in feature modelling tools. Finally, in Section 5.4, we discuss examples of FM configurator applications.

Table 5.1 provides an overview of example descriptions/presentations of tools supporting (1) the *design* of Feature Models (FMs), (2) their *analysis*, and (3) *FM configuration*. Following the major objectives of this book, we will specifically focus on discussing AI techniques related to the topics of *knowledge representation and reasoning* (KRR), *explainable AI* (XAI), and *machine learning* (ML). In this context, we provide example screenshots of tools if corresponding test versions were publicly accessible without the need of purchasing a license. For an in-depth analysis of the existing tool support in software product lines, we refer to Horcas et al. [57].

A. Felfernig et al., *Feature Models*, SpringerBriefs in Computer Science,
https://doi.org/10.1007/978-3-031-61874-1_5

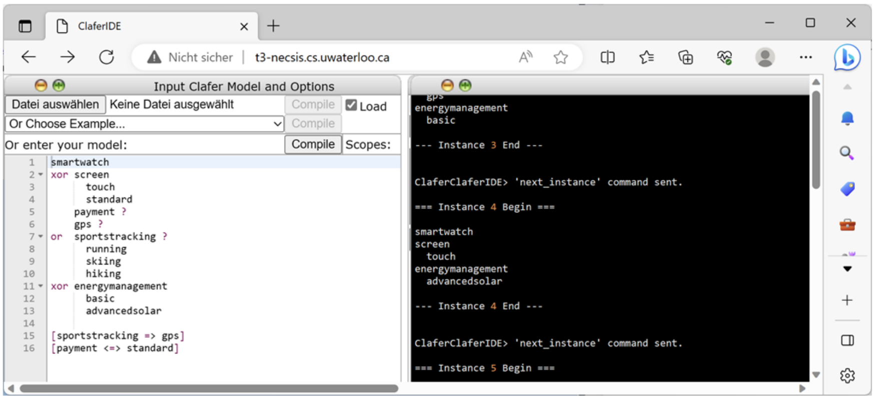
AI Areas	Tools		Applications
	Feature modelling	FM analysis	Interacting with FM configurators
knowledge representation (KR)	*basic FM* [7, 18, 65, 60, 70, 78, 106] *attribute-based FM* [2, 7, 18, 106] *cardinality-based FM* [7, 18, 106]	*operations with/without solver support* [7, 14, 18, 56, 65, 70, 78, 81, 100, 106]	*basic FM configuration* [9, 46, 51, 94] *attribute-based* [2, 9, 12, 48, 92] *cardinality-based* [9, 23]
reasoning (R)	*CSP-based* [2, 7, 82, 92, 118] *SAT-based* [18, 46, 48, 106] *rule-based* [65, 70]	*CSP-based analysis* [7, 14, 16] *SAT-based analysis* [14, 18, 106] *rule-based analysis* [65, 70]	*CSP-based* [12, 82, 91, 105] *SAT-based* [79] *rule-based* [20, 38, 75]
explainable AI (XAI)	*argumentation-based explanations* [35, 109] *consistency-based explanations* [31, 92, 107]	*FM inconsistencies* [14, 17, 74, 67] *FM redundancies analysis* [14, 69]	*explaining configurations and inconsistencies* [19, 91, 92, 111, 116]
machine learning (ML)	*configuration space learning* [85] *knowledge extraction from data* [80, 72, 104, 112]	*predicting faulty FM elements* [40]	*recommending features* [4, 6, 37, 88, 94] *recommending reconfigurations* [4, 37, 44, 46]

Table 5.1: Tool and application landscape: example tools and applications (*feature modelling, FM analysis, and FM configuration*).

5.2 Feature Modelling Tools

Clafer [7] is a knowledge representation language and environment for feature modelling and configuration, class and object modelling, and metamodelling. The system is available as a desktop application and as a set of publicly available web-based tools.[1] It supports variability modelling including also non-Boolean features (attributes) and constraints about their values, arbitrary multiplicity in group features (e.g., x..y, where x can be distinct from 1 and y distinct from *), feature clones and abstract classes, and multi-objective optimization. Using Clafer, complete and consistent configurations can already be generated in the modelling phase which helps to more easily understand the semantics of the FM. An example of applying Clafer to represent our *smartwatch* FM is shown in Figure 5.1.

[1] https://www.clafer.org/



```
smartwatch
xor screen
        touch
        standard
    payment ?
    gps ?
or  sportstracking ?
        running
        skiing
        hiking
xor energymanagement
        basic
        advancedsolar

[sportstracking => gps]
[payment <=> standard]
```

```
  gps
energymanagement
  basic

--- Instance 3 End ---


ClaferClaferIDE> 'next_instance' command sent.

=== Instance 4 Begin ===

smartwatch
screen
  touch
energymanagement
  advancedsolar

--- Instance 4 End ---


ClaferClaferIDE> 'next_instance' command sent.

=== Instance 5 Begin ===
```

Fig. 5.1: Feature modelling and generation of configurations with CLAFER. The FM is shown on the left-hand-side, a corresponding configuration on the right-hand-side.

FEATUREIDE [106][2] is an open-source Eclipse framework for feature-oriented software development (FOSD) with a plug-in-based extension mechanism to integrate and test existing tools and SPL approaches. As FEATUREIDE supports abstract features. In FEATUREIDE, the FM and the corresponding configuration interface are closely connected [86], for example, the configuration interface is based on the same hierarchical structure as defined in the FM (see Figure 5.2). In FEATUREIDE, solver-based propagation also enforces consistency between selected features and the constraints defined in the FM. For example, the deselection of some features could also enforce the deselection of related features. FEATUREIDE also supports the concept of *focused views* with the idea that only those features are visible to the user which are in the current focus, for example, if a user selects a specific feature, he/she might be interested also in related subfeatures but there is no need to display the complete feature tree.

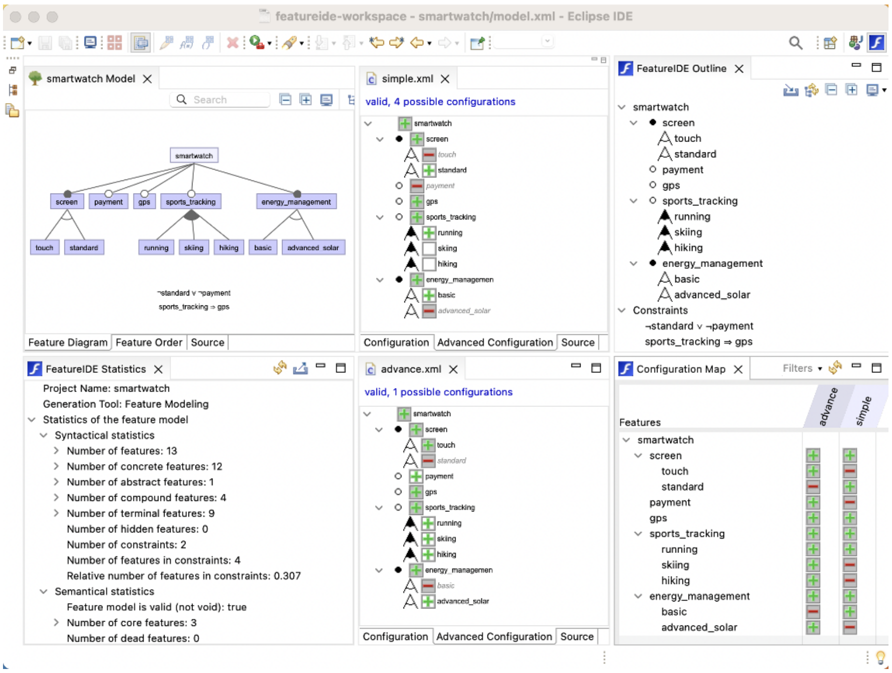
PURE::VARIANTS [18][3] is an Eclipse-based solution supporting different variability modelling concepts such as features with attributes, feature clones, variant instances, hierarchical variant composition, and OCL-type constraints. Based on specifications in its family model, it supports the generation and validation of the final configuration, i.e., the code of various programming and scripting languages. An example screenshot of the PURE::VARIANTS environment is provided in Figure 5.3.

S.P.L.O.T. (SOFTWARE PRODUCT LINES ONLINE TOOLS) [78][4] is a web-based environment for the design, analysis, and configuration of FMs. It supports logic-based reasoning tasks on the basis of reasoning approaches such as SAT solving and binary decision diagrams (BDDs). Furthermore, S.P.L.O.T. provides a large

[2] http://www.featureide.com/

[3] https://www.pure-systems.com/

[4] http://www.splot-research.org/



featureide-workspace - smartwatch/model.xml - Eclipse IDE

smartwatch Model

Feature Diagram | Feature Order | Source

simple.xml

valid, 4 possible configurations

Configuration | Advanced Configuration | Source

FeatureIDE Outline

```
smartwatch
  ● screen
      touch
      standard
  ○ payment
  ○ gps
  ○ sports_tracking
      running
      skiing
      hiking
  ● energy_management
      basic
      advanced_solar
Constraints
  ¬standard ∨ ¬payment
  sports_tracking ⇒ gps
```

FeatureIDE Statistics

```
Project Name: smartwatch
Generation Tool: Feature Modeling
Statistics of the feature model
  Syntactical statistics
    Number of features: 13
    Number of concrete features: 12
    Number of abstract features: 1
    Number of compound features: 4
    Number of terminal features: 9
    Number of hidden features: 0
    Number of constraints: 2
    Number of features in constraints: 4
    Relative number of features in constraints: 0.307
  Semantical statistics
    Feature model is valid (not void): true
    Number of core features: 3
    Number of dead features: 0
```

advance.xml

valid, 1 possible configurations

Configuration | Advanced Configuration | Source

Configuration Map

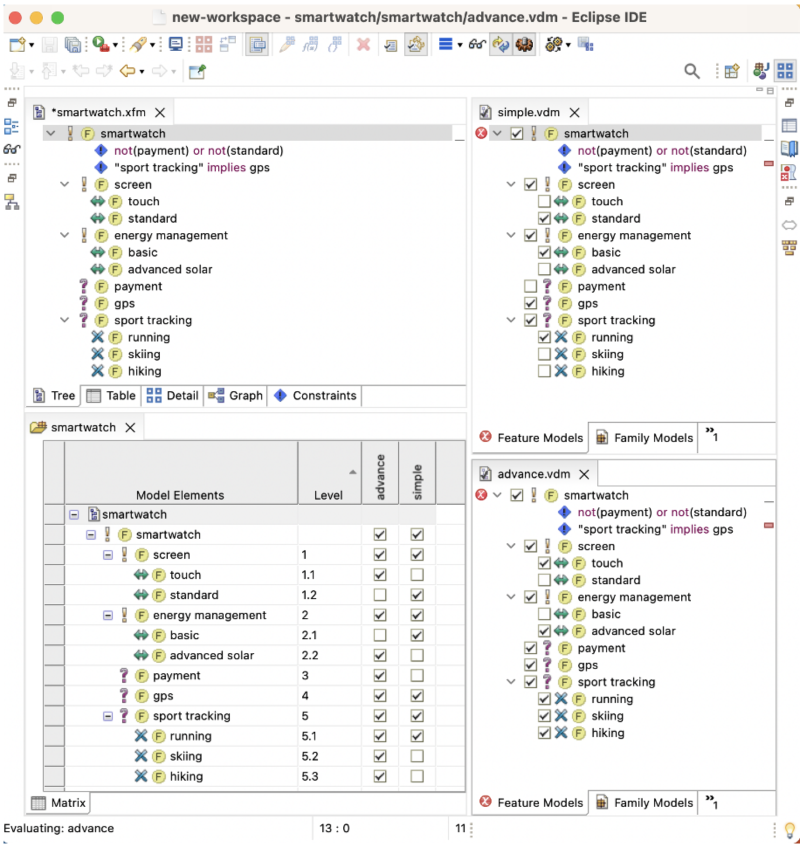
Features	advance	simple
smartwatch		
screen	🟩	🟩
touch	🟩	🟥
standard	🟥	🟩
payment	🟩	🟥
gps	🟩	🟩
sports_tracking	🟩	🟩
running	🟩	🟩
skiing	🟩	🟥
hiking	🟩	🟥
energy_management	🟩	🟩
basic	🟥	🟩
advanced_solar	🟩	🟥

Fig. 5.2: FM configuration with FeatureIDE. The configuration user interface follows the hierarchical structure defined in the FM .

repository of FMs[5] which is a kind of configuration benchmark suite used in various evaluation contexts supporting a structured comparison of different configuration problem solving approaches. The system provides a flexible web-based user interface which supports the design of FMs as well as corresponding FM configuration tasks – see the Figures 5.4 and 5.5.

The FM diagram is represented in the form of a tree view. FM-related cross-tree constraints are shown in a separate view where individual constraints can be defined in terms of logical disjunctions. In a further user view, FM statistics are displayed giving an overview of the different FM properties, for example, *#features* and *#xor groups* (i.e., alternative relationships). Finally, the environment also supports FM analysis operations including FM satisfiability, dead features, and core features. With S.P.L.O.T. as web-based application, no related installation procedures are needed. The simple graphical user interface makes it specifically applicable in the context of university courses, e.g., to give students a short but representative overview of feature modelling concepts, their semantics, and related FM configuration processes. Based on a set of input parameters, for example, *number of features*, *minimum and maximum feature branching factor*, and *consistency of generated models*, S.P.L.O.T.

[5] Also available via UVLHub [96, 103] – see https://www.uvlhub.io/.



new-workspace - smartwatch/smartwatch/advance.vdm - Eclipse IDE

*smartwatch.xfm

```
smartwatch
    not(payment) or not(standard)
    "sport tracking" implies gps
  screen
    touch
    standard
  energy management
    basic
    advanced solar
  payment
  gps
  sport tracking
    running
    skiing
    hiking
```

Tree | Table | Detail | Graph | Constraints

simple.vdm

```
[✓] smartwatch
      not(payment) or not(standard)
      "sport tracking" implies gps
  [✓] screen
      [ ] touch
      [✓] standard
  [✓] energy management
      [✓] basic
      [ ] advanced solar
  [ ] payment
  [✓] gps
  [✓] sport tracking
      [✓] running
      [ ] skiing
      [ ] hiking
```

Feature Models | Family Models

smartwatch

Model Elements	Level	advance	simple
smartwatch			
smartwatch		✓	✓
screen	1	✓	✓
touch	1.1	✓	☐
standard	1.2	☐	✓
energy management	2	✓	✓
basic	2.1	☐	✓
advanced solar	2.2	✓	☐
payment	3	✓	☐
gps	4	✓	✓
sport tracking	5	✓	✓
running	5.1	✓	✓
skiing	5.2	✓	☐
hiking	5.3	✓	☐

Matrix

advance.vdm

```
[✓] smartwatch
      not(payment) or not(standard)
      "sport tracking" implies gps
  [✓] screen
      [✓] touch
      [ ] standard
  [✓] energy management
      [ ] basic
      [✓] advanced solar
  [✓] payment
  [✓] gps
  [✓] sport tracking
      [✓] running
      [✓] skiing
      [✓] hiking
```

Feature Models | Family Models

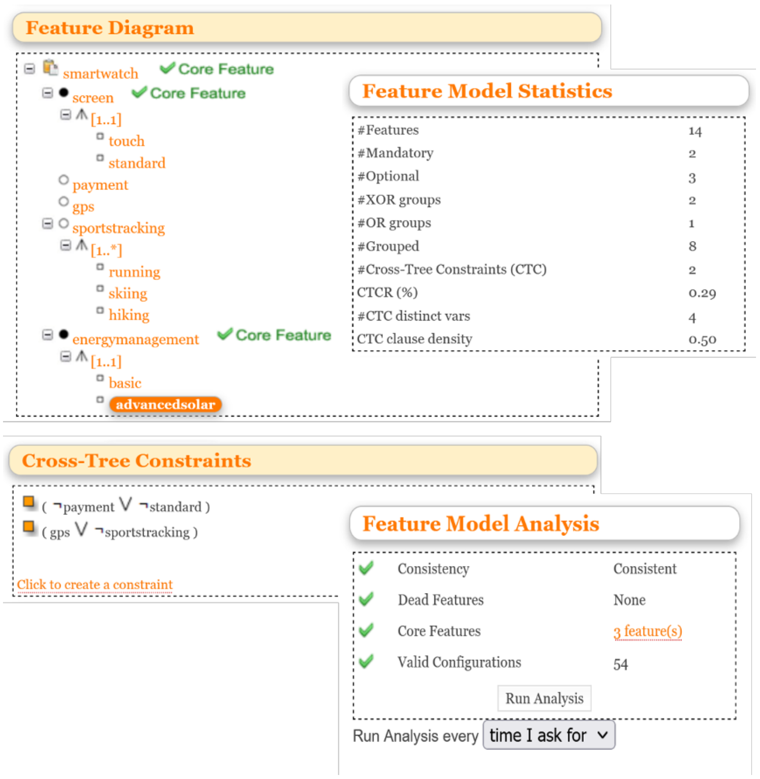
Evaluating: advance   13 : 0   11

Fig. 5.3: Example screenshots of the PURE::VARIANTS user interface with the SMARTWATCH FM on the left-hand-side and related configurations on the right-hand-side.

also supports the generation (synthesis) of FMs which is of high relevance specifically when evaluating problem solving algorithms.

FM2EXCONF [70][6] is an environment supporting the definition of configuration tasks on the basis of FMs with the basic modelling concepts of *feature*, *or*, *alternative*, *mandatory*, *optional*, and the cross tree constraints *requires* and *excludes*. FMs can be imported on the basis of the exchange formats *SXFM* (used in S.P.L.O.T.), FEATUREIDE *XML* format, and GLENCOE *JSON*. As depicted in Figure 5.6, these models can also be analyzed on the basis of analysis operations such as dead and false optional features – see Benavides et al. [13]. Out of a given FM, the system supports the direct generation of a corresponding MICROSOFT EXCEL based configurator application. Such a configurator depicts those features derived from the FM –

[6] https://github.com/AIG-ist-tugraz/FM2ExConf



Feature Diagram

- smartwatch ✓ Core Feature
  - ● screen ✓ Core Feature
    - [1..1]
      - touch
      - standard
  - ○ payment
  - ○ gps
  - ○ sportstracking
    - [1..*]
      - running
      - skiing
      - hiking
  - ● energymanagement ✓ Core Feature
    - [1..1]
      - basic
      - advancedsolar

Feature Model Statistics

#Features	14
#Mandatory	2
#Optional	3
#XOR groups	2
#OR groups	1
#Grouped	8
#Cross-Tree Constraints (CTC)	2
CTCR (%)	0.29
#CTC distinct vars	4
CTC clause density	0.50

Cross-Tree Constraints

- (¬payment ∨ ¬standard)
- (gps ∨ ¬sportstracking)

Click to create a constraint

Feature Model Analysis

✓	Consistency	Consistent
✓	Dead Features	None
✓	Core Features	3 feature(s)
✓	Valid Configurations	54

Run Analysis

Run Analysis every: time I ask for

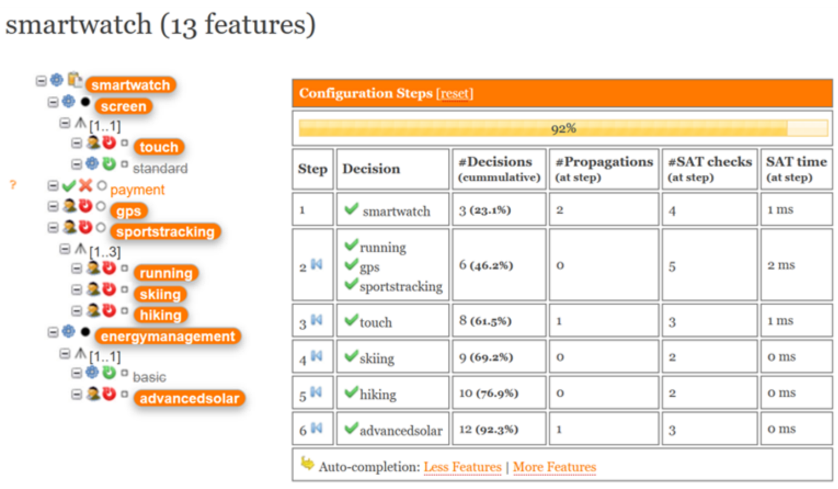
Fig. 5.4: *Smartwatch* FM developed in S.P.L.O.T. (Software Product Lines Online Tools).

on the basis of specifying 0 (feature exclusion) and 1 (feature inclusion), users can articulate their requirements with regard to a final configuration (see Figure 5.7).

For the purpose of increasing user interface understandability, the constraints derived from the FM are explicitly shown to the user. In the case of a constraint violation, a corresponding message is displayed to help the user to find a way out from the *no solution could be found* dilemma. In contrast to feature modelling environments such as FEATUREIDE, FM2EXCONF does not provide a solver integration, i.e., functionalities such as automated FM diagnosis (see Chapter 3) and configuration completion (see Chapter 4) are not supported. However, due to the simple definition and corresponding configurator generation, this environment can easily be used in knowledge representation related courses.

EVENTHELPR [35][7] is a publicly available general-purpose group decision support tool. In the context of product (line) scoping, EVENTHELPR can support stakeholders

[7] https://www.eventhelpr.com



smartwatch (13 features)

- smartwatch
  - screen
    - [1..1]
      - touch
      - standard
  - ? payment
  - gps
  - sportstracking
    - [1..3]
      - running
      - skiing
      - hiking
  - energymanagement
    - [1..1]
      - basic
      - advancedsolar

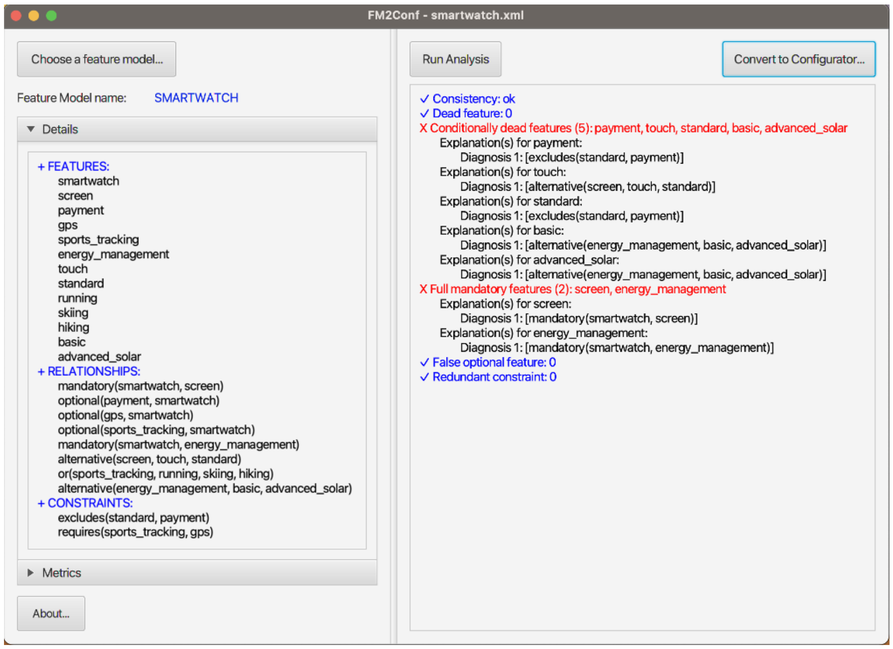
Configuration Steps [reset]

92%

Step	Decision	#Decisions (cummulative)	#Propagations (at step)	#SAT checks (at step)	SAT time (at step)
1	✓ smartwatch	3 (23.1%)	2	4	1 ms
2	✓ running ✓ gps ✓ sportstracking	6 (46.2%)	0	5	2 ms
3	✓ touch	8 (61.5%)	1	3	1 ms
4	✓ skiing	9 (69.2%)	0	2	0 ms
5	✓ hiking	10 (76.9%)	0	2	0 ms
6	✓ advancedsolar	12 (92.3%)	1	3	0 ms

Auto-completion: Less Features | More Features

Fig. 5.5: *Smartwatch* FM configuration in S.P.L.O.T..



FM2Conf - smartwatch.xml

Choose a feature model... | Run Analysis | Convert to Configurator...

Feature Model name: SMARTWATCH

Details

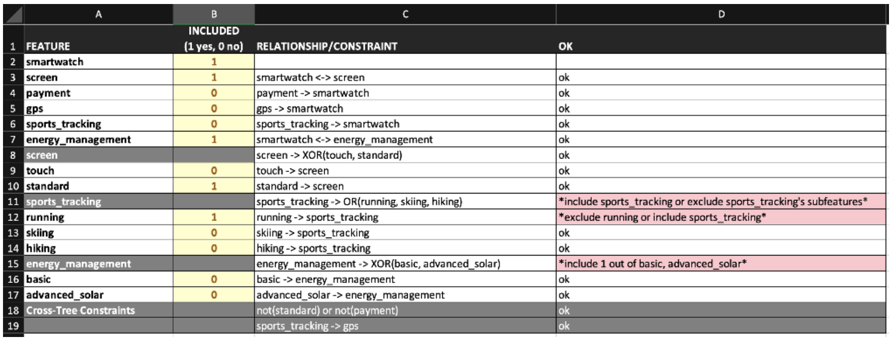
```
+ FEATURES:
    smartwatch
    screen
    payment
    gps
    sports_tracking
    energy_management
    touch
    standard
    running
    skiing
    hiking
    basic
    advanced_solar
+ RELATIONSHIPS:
    mandatory(smartwatch, screen)
    optional(payment, smartwatch)
    optional(gps, smartwatch)
    optional(sports_tracking, smartwatch)
    mandatory(smartwatch, energy_management)
    alternative(screen, touch, standard)
    or(sports_tracking, running, skiing, hiking)
    alternative(energy_management, basic, advanced_solar)
+ CONSTRAINTS:
    excludes(standard, payment)
    requires(sports_tracking, gps)
```

Metrics

About...

```
✓ Consistency: ok
✓ Dead feature: 0
X Conditionally dead features (5): payment, touch, standard, basic, advanced_solar
    Explanation(s) for payment:
        Diagnosis 1: [excludes(standard, payment)]
    Explanation(s) for touch:
        Diagnosis 1: [alternative(screen, touch, standard)]
    Explanation(s) for standard:
        Diagnosis 1: [excludes(standard, payment)]
    Explanation(s) for basic:
        Diagnosis 1: [alternative(energy_management, basic, advanced_solar)]
    Explanation(s) for advanced_solar:
        Diagnosis 1: [alternative(energy_management, basic, advanced_solar)]
X Full mandatory features (2): screen, energy_management
    Explanation(s) for screen:
        Diagnosis 1: [mandatory(smartwatch, screen)]
    Explanation(s) for energy_management:
        Diagnosis 1: [mandatory(smartwatch, energy_management)]
✓ False optional feature: 0
✓ Redundant constraint: 0
```

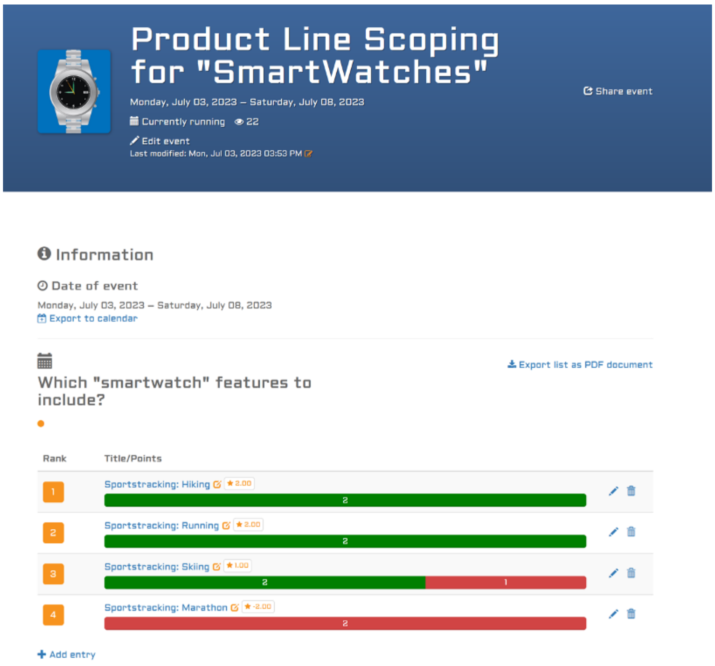
Fig. 5.6: Example screenshot of the FM2EXCONF modelling environment [70].



	A	B	C	D
1	FEATURE	INCLUDED (1 yes, 0 no)	RELATIONSHIP/CONSTRAINT	OK
2	smartwatch	1		
3	screen	1	smartwatch <-> screen	ok
4	payment	0	payment -> smartwatch	ok
5	gps	0	gps -> smartwatch	ok
6	sports_tracking	0	sports_tracking -> smartwatch	ok
7	energy_management	1	smartwatch <-> energy_management	ok
8	screen		screen -> XOR(touch, standard)	ok
9	touch	0	touch -> screen	ok
10	standard	1	standard -> screen	ok
11	sports_tracking		sports_tracking -> OR(running, skiing, hiking)	*include sports_tracking or exclude sports_tracking's subfeatures*
12	running	1	running -> sports_tracking	*exclude running or include sports_tracking*
13	skiing	0	skiing -> sports_tracking	ok
14	hiking	0	hiking -> sports_tracking	ok
15	energy_management		energy_management -> XOR(basic, advanced_solar)	*include 1 out of basic, advanced_solar*
16	basic	0	basic -> energy_management	ok
17	advanced_solar	0	advanced_solar -> energy_management	ok
18	Cross-Tree Constraints		not(standard) or not(payment)	ok
19			sports_tracking -> gps	ok

Fig. 5.7: Example screenshot of FM2EXCONF [70].

in finding solid arguments regarding the inclusion and exclusion of features. An example screenshot of EventHelpR is shown in Figure 5.8.



Product Line Scoping for "SmartWatches"

Monday, July 03, 2023 – Saturday, July 08, 2023

Currently running 22

Edit event

Last modified: Mon, Jul 03, 2023 03:53 PM

Share event

Information

Date of event

Monday, July 03, 2023 – Saturday, July 08, 2023

Export to calendar

Which "smartwatch" features to include?

Export list as PDF document

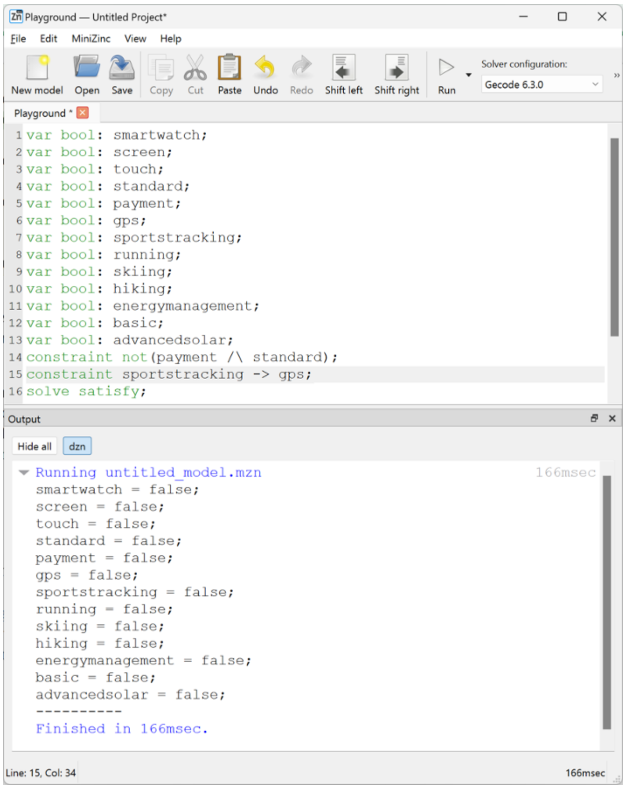
Rank	Title/Points
1	Sportstracking: Hiking ★ 2.00 — 2
2	Sportstracking: Running ★ 2.00 — 2
3	Sportstracking: Skiing ★ 1.00 — 2 / 1
4	Sportstracking: Marathon ★ -2.00 — 2

Add entry

Fig. 5.8: Example screenshot of EventHelpR.

The underlying idea of EventHelpR [35] is to allow users (stakeholders) to provide arguments for or against the inclusion of specific features. These arguments are then aggregated feature-wise indicating a “global” tendency of feature inclusion or exclusion. Such type of preference elicitation user interfaces can help group members to focus on the exchange of decision-relevant information, i.e., arguments, and thus to significantly improve the overall decision quality (in terms of the selected features). With this, EVENTHELPR provides a kind of explanation-based user interface which helps to make the reasons for feature inclusion and exclusion transparent. The aggregation of the preferences of individual users is supported in terms of a group aggregation function [33, 108, 109] which determines the share of positive and negative arguments on a graphical level (see Figure 5.8). In the line of EVENTHELPR, the open source requirements engineering environment OPENREQ[8] supports group decision making in the context of prioritizing software features [31, 43, 107].

MINIZINC IDE [82][9] is a tool that allows for the specification and solving of constraint satisfaction problems (CSPs) in a graphical environment. The specification of our example FM in MINIZINC IDE is depicted in Figure 5.9.


Playground — Untitled Project*
File Edit MiniZinc View Help
New model Open Save Copy Cut Paste Undo Redo Shift left Shift right Run
Solver configuration:
Gecode 6.3.0
Playground *
Output
Hide all dzn
Line: 15, Col: 34
166msec

```
var bool: smartwatch;
var bool: screen;
var bool: touch;
var bool: standard;
var bool: payment;
var bool: gps;
var bool: sportstracking;
var bool: running;
var bool: skiing;
var bool: hiking;
var bool: energymanagement;
var bool: basic;
var bool: advancedsolar;
constraint not(payment /\ standard);
constraint sportstracking -> gps;
solve satisfy;
```

```
Running untitled_model.mzn                                   166msec
  smartwatch = false;
  screen = false;
  touch = false;
  standard = false;
  payment = false;
  gps = false;
  sportstracking = false;
  running = false;
  skiing = false;
  hiking = false;
  energymanagement = false;
  basic = false;
  advancedsolar = false;
  ----------
  Finished in 166msec.
```

Fig. 5.9: Example screenshot of MINIZINC IDE.

[8] https://openreq.eu/

[9] https://www.minizinc.org/ide/

On the one hand, a major disadvantage with regard to feature modelling is that IDEs such as the MiniZinc environment support the specification of variables and constraints, however, no related graphical knowledge representation of features, relationships, and cross-tree constraints is provided. On the other hand, models (represented as CSPs) can easily be extended with attributes, for example, for each relevant feature, we could introduce a corresponding price attribute indicating the price of a feature (e.g., for the feature gps, we can introduce the attribute $pricegps$). Furthermore, an attribute $totalprice$ would represent the overall price of a configuration using an additional resource constraint of type $totalprice = pricefeature_1 + .. + pricefeature_n$. This way, MiniZinc IDE could be used within the scope of different courses related to knowledge representation and reasoning. Specifically, in the context of constraint solving (and beyond), search optimization plays a major role. With highly complex FMs, a corresponding solver search optimization on the basis of the concepts of machine learning becomes increasingly relevant [90, 113]. Related topics are (1) *configuration space learning* [85] which includes intelligent synthesis methods for the generation of relevant test configurations and (2) *knowledge extraction from data* [80, 72, 104, 112] which can help to increase the efficiency of modelling processes, for example, by the automated extraction of features from natural language text.

Finally, Gears [65][10] is a product line engineering tool and lifecycle framework which supports all phases of the SPL process. Product line engineering is interpreted as a highly automated task similar to the manufacturing of physical products in a factory. Gears provides a quasi-standard unified variant management approach that is vendor-independent but integrates with other third party and proprietary tools, assets, and processes across each stage of the lifecycle — and across engineering and operations disciplines. This helps to reduce complexity, time, effort, and errors on the one hand and breaks down organizational and operational silos enabling better communication and alignment and greater collaboration on the other hand.

Product Configuration Environments. Specifically, in the context of knowledge-based product configuration scenarios [36, 98, 102], there exists a plethora of commercial environments supporting the development of configurator applications. Without any claim to completeness, related example systems are camos[11], ConfigIt[12], Tacton[13], encoway[14], and Variantum[15]. Note that these systems are not primarily based on FMs but in many cases on a more object- or component-oriented knowledge representation (see, e.g., [30, 34]). The corresponding reasoning (solver) support can range from constraint-based and SAT-based to rule-based reasoning [36, 98, 102].

[10] https://biglever.com/solution/gears/

[11] https://www.camos.de

[12] https://configit.com

[13] https://www.tacton.com/

[14] https://www.encoway.de

[15] https://variantum.com/

Based on the given overview of feature modelling tools (and beyond), we now discuss examples of tools that support the *analysis* of FMs.

5.3 Feature Model Analysis: Tool Support

Due to changing requirements and dependencies between features, the development and maintenance of large and complex product lines can become a difficult task [63]. In the following, we discuss different tools that assist developers in the design and maintenance of FMs.

`flama` [50][16] is a tool suite for variability model analysis in general and FM analysis in particular. It is developed as Python framework with a plugin–based architecture where different plugins for FM languages can be developed as well as reasoning capabilities. `flama` supports UVL models (see Section 2.7) and SAT, BDD, and SMT reasoning capabilities [76]. It also supports analysis operations that do not need a solver support (see Section 3.1.1). The project is maintained and promoted by 4 different universities and its spirit is to serve a common base for the development of FM analysis and configuration capabilities. Many applications use `flama` as background for analysis capabilities [15, 49, 58, 59, 71, 77, 95, 96, 103].

FaMa [14][17] is a wide-spread application for FM analysis written in Java. It is a framework for the automated analysis of FMs integrating several of the most commonly used logic-based representations and solvers proposed in the literature (BDD, SAT, CSP solvers). After having imported a FM from nearly any other FM tool, it provides support for validity checking and finding inconsistencies. By that, it covers the domain analysis phase really well, for example, features with attributes, numeric values, and constraints. For requirements analysis, too, FaMa stands out with its automatic reasoning capabilities based on symbolic AI methods, such as model validation (e.g., non-satisfiable model), anomaly detection (e.g., dead features, false-optional features), model counting (e.g., number of configurations), and redundancy detection [69]. Currently, the project lacks support since its main contributors are now developing `flama`.

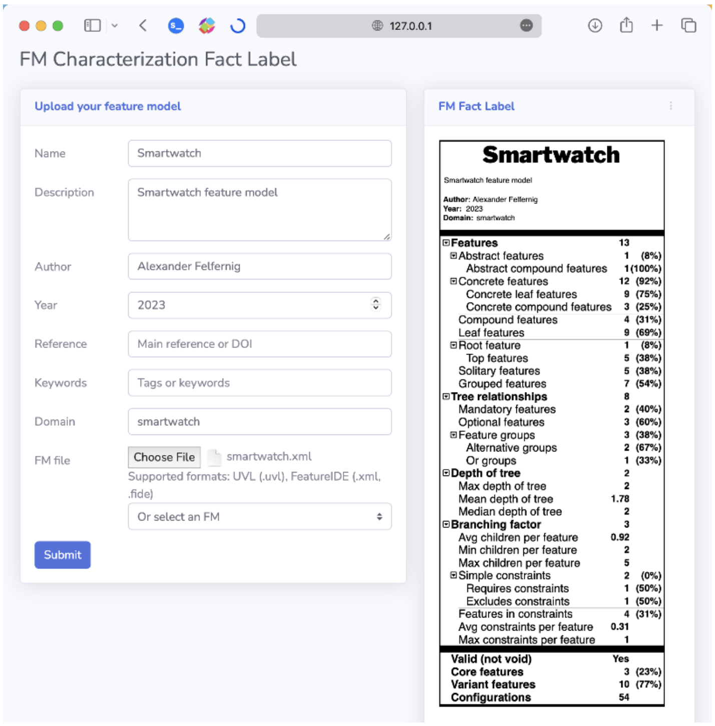
FactLabel [56][18] is a web-based tool that supports (in a configurable fashion) the interactive visualization of FM characteristics (as a result of executing various FM analysis operations) which can then also be exported to other tools in diverse exchange formats (e.g., the Universal Variability Language). The result of applying FactLabel to our example FM is depicted in Figure 5.10.

FeatureIDE [106] provides a set of analysis operations (a.o. dead features, false optionals, and redundant constraints) which are defined on a logical basis used to determine anomaly for developers [63]. Such a reasoning about different FM properties can help to generate explanations that provide reasons as to why specific

[16] https://flamapy.github.io

[17] http://www.fama-ts.us.es/

[18] https://fmfactlabel.adabyron.uma.es/



Name	Smartwatch
Description	Smartwatch feature model
Author	Alexander Felfernig
Year	2023
Reference	Main reference or DOI
Keywords	Tags or keywords
Domain	smartwatch
FM file	smartwatch.xml Supported formats: UVL (.uvl), FeatureIDE (.xml, .fide) Or select an FM

Smartwatch

Smartwatch feature model

Author: Alexander Felfernig
Year: 2023
Domain: smartwatch

Features	13	
Abstract features	1	(8%)
Abstract compound features	1	(100%)
Concrete features	12	(92%)
Concrete leaf features	9	(75%)
Concrete compound features	3	(25%)
Compound features	4	(31%)
Leaf features	9	(69%)
Root feature	1	(8%)
Top features	5	(38%)
Solitary features	5	(38%)
Grouped features	7	(54%)
Tree relationships	8	
Mandatory features	2	(40%)
Optional features	3	(60%)
Feature groups	3	(38%)
Alternative groups	2	(67%)
Or groups	1	(33%)
Depth of tree	2	
Max depth of tree	2	
Mean depth of tree	1.78	
Median depth of tree	2	
Branching factor	3	
Avg children per feature	0.92	
Min children per feature	2	
Max children per feature	5	
Simple constraints	2	(0%)
Requires constraints	1	(50%)
Excludes constraints	1	(50%)
Features in constraints	4	(31%)
Avg constraints per feature	0.31	
Max constraints per feature	1	
Valid (not void)	Yes	
Core features	3	(23%)
Variant features	10	(77%)
Configurations	54	

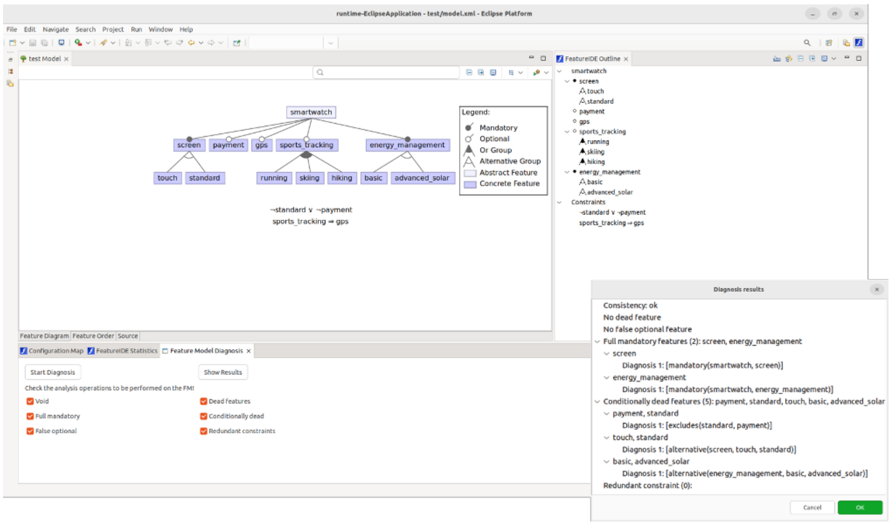
Fig. 5.10: FactLabel user interface: output for the *smartwatch* FM.

expected FM properties do not hold. For example, an explanation can indicate minimal sets of FM relationships and cross-tree constraints that are responsible for a specific unintended FM semantics [32, 63, 74].

FMTesting [21][19] is a FeatureIDE plugin focusing on the application of model-based diagnosis [32, 93] for explaining anomalies in FMs. The determined diagnoses represent minimal sets of FM relationships and constraints responsible for an unintended behavior of a FM (e.g., which are the relationships and constraints that make a specific feature a void feature). Given a specific FM, the developer can select analysis operations to be activated (e.g., void features) and the plugin determines the set of void features with the corresponding explanations (diagnoses). A screenshot of FMTesting with a corresponding diagnosis output is shown in Figure 5.11.

Similar to FMTesting, Hentze et al. [74] present a FeatureIDE service that supports the determination of diagnoses (denoted as hyper-explanations) for dead features. Furthermore, Bendík and I. Černá [17] present a tool that supports the determination of *minimal unsatisfiable subsets* (conflict sets) which are a basis for diagnosis determination [40, 93] (see Chapter 3). Finally, Le et al. introduce DirectDebug [67, 68] which is a Java library for the automated diagnosis of FMs.

[19] https://github.com/AIG-ist-tugraz/FMTesting



¬standard ∨ ¬payment
sports_tracking ⇒ gps

Check the analysis operations to be performed on the FM!

- [✓ Void]
- [✓ Full mandatory]
- [✓ False optional]
- [✓ Dead features]
- [✓ Conditionally dead]
- [✓ Redundant constraints]

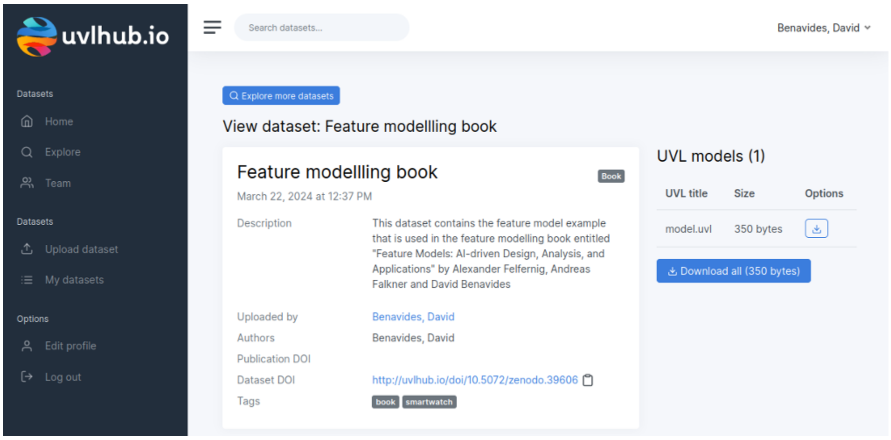
```
Consistency: ok
No dead feature
No false optional feature
Full mandatory features (2): screen, energy_management
  screen
    Diagnosis 1: [mandatory(smartwatch, screen)]
  energy_management
    Diagnosis 1: [mandatory(smartwatch, energy_management)]
Conditionally dead features (5): payment, standard, touch, basic, advanced_solar
  payment, standard
    Diagnosis 1: [excludes(standard, payment)]
  touch, standard
    Diagnosis 1: [alternative(screen, touch, standard)]
  basic, advanced_solar
    Diagnosis 1: [alternative(energy_management, basic, advanced_solar)]
Redundant constraint (0):
```

Fig. 5.11: FMTesting: a diagnosis plugin for FeatureIDE.

UVLHub [96][20] is a data set repository supporting feature models in UVL format using open science principles. Open science principles promote transparency, accessibility, and collaboration in scientific research. UVLHub provides a front-end that facilitates the search, upload, storage, and management of feature model datasets, improving the capabilities of discontinued proposals such as S.P.L.O.T.. It communicates with Zenodo providing a permanent location for datasets and it is maintained by three active universities in variability modelling. Figure 5.12 shows UVLHub in action where the data set of the feature model shown in the book is displayed.

In addition to the previously discussed examples, the following tools support FM analysis operations [57]. pure::variants [18] supports a set of analysis operations comparable to those of FeatureIDE. In pure::variants, it is possible to determine the number of configurations for individual subtrees of the FM [57]. S.P.L.O.T. [78] also provides a basic set of analysis operations including dead features and FM consistency (see Figure 5.5). Analysis operations supported in FM2EXCONF (see Figure 5.7) [38, 70] resemble those provided by the FMTesting environment (see Figure 5.11). Finally, a formalization in terms of mixed integer linear programs for the analysis of Clafer models is provided in Weckesser et al. [115] – compared to most of other existing analysis approaches, Clafer [7] FM analysis has to deal with a higher complexity due to a higher expressivity of the underlying FM language [115].

[20] https://www.uvlhub.io/



View dataset: Feature modellling book

Feature modellling book

March 22, 2024 at 12:37 PM

Description	This dataset contains the feature model example that is used in the feature modelling book entitled "Feature Models: AI-driven Design, Analysis, and Applications" by Alexander Felfernig, Andreas Falkner and David Benavides
Uploaded by	Benavides, David
Authors	Benavides, David
Publication DOI	
Dataset DOI	http://uvlhub.io/doi/10.5072/zenodo.39606
Tags	book smartwatch

UVL models (1)

UVL title	Size	Options
model.uvl	350 bytes	

Fig. 5.12: UVLHub: a data set repository of feature models in UVL format [96].

5.4 Feature Model Configurator Applications

On the basis of the previous presentations of feature modelling tools and corresponding FM analysis approaches, we now discuss different FM configurator applications. In our discussion, we specifically focus on those applications with available associated descriptions of the used AI methods.

Software Product Line Configuration. FMs help to represent the configuration space of a large number of different systems which can be assembled out of a set of pre-defined (implemented) software artifacts. Thus, the concept of software artifacts in SPL configuration resembles the concept of components when configuring physical systems [36]. FMs of industrial software product lines [8, 89, 84] can become large and complex – see, for example, the Linux operating system FM [97, 105]. Asikainen et al. [9] introduce the configuration environment WeCoTin and show how their configuration environment can be used to model and implement a text editor configurator application. The system includes a modelling environment which can be used to create (attribute- and cardinality-based) FMs and thereof (in an automated fashion) a corresponding configurator user interface. In the line of this work, various software systems and tools support the task of software product line configuration. Related SPLs exist, for example, in operating systems [97], automotive systems [27, 28, 117], synthetic biology [24], and software product lines for large telescope control software [55].

Configuring Control Systems. Beek et al. [12] define a single FM covering the complexity (high variability) of the European Train Control System (ETCS), an automatic train protection (ATP) system which continuously supervises all trains on a railway line, ensuring that the safety speed and distance are not exceeded. This model shows the different components to be installed at the different levels established by the ETCS standards and helps engineers to understand and solve specific issues, such as aligning the interfaces between different systems (e.g. onboard and wayside equipment), development of sustainable solutions by involved manufacturers, interoperability among systems at different ETCS levels, backward compatibility of involved subsystems, and evolution towards new requirements. The FM can also act as facilitator in cost and performance analysis for planning purposes because it is implemented in CLAFER which supports the generation of the complete set of instances (products) from the model. Furthermore, attributes associated to features and quality constraints allow assignment of costs and their corresponding optimization.

Runtime Configuration. Capilla et al. [5, 23] introduce concepts supporting the specific scenario of *runtime configuration* in the context of dynamic software product lines. Such a type of configuration enables the addition and removal of variants on-the-fly, runtime dependencies and constraints checking, dynamic and optimized reconfiguration, and multiple binding and re-binding. Applications range from service-oriented and cloud systems over mobile software and ecosystems for autonomous and self-adaptive systems to cyber-physical systems [66] which have to reuse, reorganize, and reconfigure their components during runtime. Benefits are that variants are bound at the latest time possible which ensures high flexibility, for example, (de)activation of system features or adaptation to changing conditions of the environment. Related ideas on *anytime diagnosis* supporting efficient reconfiguration tasks are discussed in Felfernig et al. [44].

Release Plan Configuration and Reconfiguration. In requirements engineering, dependencies are key concerns to be taken into account in the context of prioritization processes. Raatikainen et al. [92] relate individual requirements to individual features in a FM and thus represent the task of requirements prioritization as a FM configuration task. Their feature modelling environment supports the inclusion of attributes, for example, a release can be regarded as an attribute of a feature. Having completed an FM, it can be translated into the representation of a constraint satisfaction problem (CSP). A specific aspect of this environment is the inclusion of *explanations* that help stakeholders to figure out the sources of an inconsistency. For example, the maximum allowed amount of efforts assigned to a release plan cannot be taken into account due to the fact that too many features are required to be included, explanations help to figure out minimal sets of feature sets that – if excluded from the current release plan – allow the identification of a solution (release plan). In a similar fashion, the same explanation concepts can be applied to reconfigure an existing release plan to take into account a set of new requirements [44]. Such explanations are determined on the basis of the concepts of direct diagnosis [42] (see also Chapters 3 and 4).

Configuration in Augmented Reality. Motivated by the trend of mobile shopping, Gottschalk et al. [51] present a FM configurator application that supports model-based configuration of furniture, for example, the configuration of kitchens. A product modeler application supports the use of basic feature modelling concepts, i.e., *feature*, *mandatory*, *optional*, *alternative*, *or*, *requires*, and *excludes*. The corresponding product configurator (derived from the feature model) supports the creation of individual furniture configurations ($3D$ object compositions) where a collection of assets ($3D$ objects and textures) is used for generating a $3D$ visualization of the solution (configuration) generated by the configurator.

Configuring Operating Systems. The LINUX operating system kernel can be regarded as a kind of holy grail of the SPL community [105]. Sincero and Schröder-Preikschat [101] introduce the original variability management of the LINUX kernel configurator. The underlying FM supports basic FM variability modelling concepts. A more up-to-date summary of the existing LINUX kernel configuration support is provided in Franz et al. [46] where XCONFIG is mentioned as corresponding graphical configurator. The underyling variability definitions follow basic FM variability concepts [62]. An extension of XCONFIG is CONFIGFIX which is a tool that supports conflict resolution in the case of detected inconsistencies in the current configuration. Conflict detection and resolution in CONFIGFIX is based on SAT solving [46]. As an additional supportive service in the context of LINUX kernel configuration, Acher et al. [4] propose a machine learning approach to predict the kernel size of a LINUX kernel configuration – such a service can be applied to recommend features and to rank configuration candidates. Furthermore, in the context of optimizing a kernel configuration, predictions can be chosen to select specific reconfiguration options. Herzog et al. [54] introduce a machine learning based approach that helps to optimize operating system parameters on the basis of linear models and neural networks. A configuration front-end for Kconfig-based software product lines is also presented in Friesel et al. [47].

Configuration in Automotive. For decades, automotive industry has been among the most extreme applications of SPLs with highly complex products with a literally astronomical number of variants [117, 118]. Modern automobiles can comprise hundreds of separate engineering systems, such as engines, brakes, air bags, lights, windshield wipers, climate control, infotainment, etc. Some of them are extraordinarily complex, such as high beam headlights that react to oncoming traffic at night. For a discussion of related details on automotive product line engineering, we refer to Wozniak and Clements [117]. From the end-user (customer) point of view, nearly every car provider also offers configuration services to their customer communities. An example thereof is the RENAULT configurator as discussed in Xu et al. [118]. This configurator is based on constraint satisfaction (CSP) knowledge representations [10] and corresponding knowledge compilation (compression) approaches which can lead to significant reductions in terms of time needed for identifying a configuration.

Configuring Videos. Acher et al. [2] present VIVID which is a variability-based tool for configuring video sequences. In VIVID, variability modelling is based on attribute-based FMs which can be translated into a corresponding representation of a constraint satisfaction problem (CSP). As a constraint solver, the systems uses CHOCO, an open source JAVA library for constraint programming.[21] In the line of [2], Lubos et al. [73] introduce a FM configuration approach with a similar objective in the sense that videos should be configured in such a way that different criteria such as maximum duration and topic coverage are fulfilled. In this context, basic FMs are used for variability modelling.

Personalized Configuration. The idea of personalized configuration is to support configurator users in finding a configuration that satisfies their preferences, for

[21] https://choco-solver.org/

example, in terms of providing user-individual recommendations of components and features of potential relevance [29, 87]. Following the idea of personalizing the interactions with configurators, Pereira et al. [88] introduce a FM configuration environment enhanced with different recommendation approaches [22, 114]. The used recommendation algorithms determine features of potential relevance for the user which are shown within the scope of a corresponding configuration process. Also following the idea of personalized configuration, Rodas-Silva et al. [94] present a recommender system that suggests implementation components based on a set of selected features (related to potential WordPress website configurations). The underlying idea is that in product lines often a selected feature can be implemented by different components – finding the optimal components to implement a given configuration is the task to be supported. In this context, basic feature modelling concepts are supported to represent variability properties in the FM.

Further Configuration Services. Jézéquel et al. [60] introduce an authentication library which offers a huge variety of options (features) where only a subset is needed for each concrete installation of an application on a server, for example, authentication by password or fingerprint but not retinal scan. Not all features are either selected or excluded – some stay open so that an administrator can change settings at runtime. Such runtime features are called *feature toggles*. In order to avoid unnecessary code, which might make hacker attacks easier, they propose automated source code creation, removal, and injection based on the selected, excluded, and open design time features. Fritsch et al. [48] present YAP (*Yet Another Product Configurator*) which is based on FeatureIDE combined with an underlying SAT solver. The configurator has been developed for customers of a German bank with the goal of assisting non-technical-affine users in their product design [48]. The underlying attribute-based FM consisting of around 940 features and 1,200 cross tree constraints also supports a kind of standardization in terms of ensuring consistent offers and corresponding customer information documents. Niederer et al. [83] present a product configurator with the goal to provide configurator user interfaces with more flexibility regarding the specification of user preferences. In product configuration, users can be overwhelmed by the complexity of a product variability (in terms of features and constraints). Furthermore, many configuration user interfaces do not differentiate between novice users and experts with regard to the product assortment. The context-aware chatbot introduced in Niederer et al. [83] provides more flexibility in terms of available conversation paths and the way user preferences can be specified, which leads to a lower perceived complexity when interacting with the configurator. Similar observations regarding improvements in the quality of user interaction have also been reported in the context of constraint-based recommender systems [52].

5.5 Discussion

In this chapter, we have provided an overview of existing FM tools and applications ranging from *feature modelling*, *FM analysis*, to different *FM configuration applica-*

tions. We selected applications specifically from SPLC[22] and VAMOS[23]. Despite the successful application of AI technologies for software and systems product lines, we also see open challenges and important topics for future research.

Bridging design and runtime variability. The flexible continuum of design and runtime variability needs to be bridged and managed [60]. A formal semantics of the underlying models can help to ensure result correctness or completeness, for example, in the context of risk models [26]. For many industrial domains, it would be very helpful to build an extensible ontology which can serve as a backbone for joint research. Higher automation of software verification (e.g., model checking), of change impact analysis on test cases, and of test case creation and repair would help to reduce development efforts [1]. Often, it is difficult to find a good trade-off between complexity and benefit [55]. Strategies for that should be examined.

Inclusion of Large Language Models (LLMs). By the end of 2022, LLMs for understanding natural language and corresponding systems for human-like interaction such as ChatGPT[24] have gained much attention. It is worth evaluating how those systems could be used to improve feature modelling and feature configuration [3, 49], for example, as "intelligent" natural language assistants for recommending or explaining configuration decisions. Important aspects thereby are to avoid the notorious "hallucination" (i.e. the tendency of such systems to introduce "false facts" to satisfy user requirements or if they lack true information) and the protection of data (i.e. ensure non-disclosure and non-derivability of private and confidential data).

Integration of Standard Algorithms. Although applied and integrated in different research prototypes, FM diagnosis (for supporting explanations) is often not supported or only supported on the basis of some proprietary, often incomplete algorithmic solutions. Tool providers need to emphasize the integration of standard algorithms, such as QuickXPlain [61] for conflict detection and model-based diagnosis for identifying minimal hitting sets (diagnoses) [42, 93].

Cognitive Issues in FM Development. The understandability and maintainability of FMs depend on the used knowledge representation. For example, the way in which constraints are ordered in a knowledge base or the way specific logical properties (e.g., an implication) are specified, can have an enormous impact on the overall understandability of a knowledge base [39]. Research is needed to better understand which knowledge structures help to optimize the overall understandability of a knowledge base (and the corresponding feature model) [41].

Integration of FM Configuration with Machine Learning. Although related solutions already exist in terms of research prototypes, an integration of FM configuration with corresponding machine learning approaches is rather the exception of the rule. Such technologies help to better assist configurator users to select the

[22] https://splc.net/

[23] https://dblp.org/db/conf/vamos/index.html

[24] https://openai.com/blog/chatgpt

relevant features and also help companies to better decide on which features should be recommended to which users [37, 90].

Configuration Space Learning. Although highly relevant in different application contexts (e.g., optimizing solver search heuristics or determining stable operating systems parameter settings), configuration space learning is still more a research topic than on the way of being integrated into existing FM configuration environments [85]. The integration of such techniques into feature modelling and configuration environments can help to significantly improve configurator runtime performance as well as the performance of the generated products.

Explaining Configurations and Beyond. In (feature model) configuration and CSP/SAT solving, there exist various research contributions regarding the provision of explanations [53, 64]. However, there exist open issues specifically with regard to taking into account the aspect of explanation goals which have a significant impact on the way explanations are formulated and presented to the user [110]. An example of such an explanation goal could be *persuasiveness*, i.e., to sensitize a user with regard to a specific aspect, for example, *configuration sustainability* [45].

References

1. M. Abbas, R. Jongeling, C. Lindskog, E. Enoiu, M. Saadatmand, and D. Sundmark. Product Line Adoption in Industry: An Experience Report from the Railway Domain. In *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, pages 1–11. ACM, 2020.
2. M. Acher, M. Alférez, J. Galindo, P. Romenteau, and B. Baudry. ViViD: A Variability-Based Tool for Synthesizing Video Sequences. In *18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, pages 143–147. ACM, 2014.
3. M. Acher, J. G. Duarte, and J.-M. Jézéquel. On programming variability with large language model-based assistant. In *27th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '23, pages 8–14, New York, NY, USA, 2023. ACM.
4. M. Acher, H. Martin, L. Lesoil, A. Blouin, J. Jézéquel, D. Khelladi, E. Djamel, O. Barais, and J. Pereira. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 85–96. ACM, 2022.
5. O. Aguayo and S. Sepulveda. Variability Management in Dynamic Software Product Lines for Self-Adaptive Systems – A Systematic Mapping. *Applied Sciences*, 12(20), 2022.
6. Y. Amraoui, M. Blay-Fornarino, P. Collet, F. Precioso, and J. Muller. Evolvable SPL Management with Partial Knowledge: An Application to Anomaly Detection in Time Series. In *26th ACM International Systems and Software Product Line Conference (SPLC 2022)*, pages 222–233. ACM, 2022.
7. M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. Liang, and K. Czarnecki. Clafer Tools for Product Line Engineering. In *17th International Software Product Line Conference Co-Located Workshops*, SPLC '13 Workshops, pages 130–135, New York, NY, USA, 2013. ACM.
8. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented Software Product Lines*. Springer, 2016.

9. T. Asikainen, T. Männistö, and T. Soininen. Using a Configurator for Modelling and Configuring Software Product Lines based on Feature Models. In *Workshop on Software Variability Management for Product Derivation (SPLC'2004)*, pages 24–35, 2004.
10. J. Astesana, L. Cosserat, and H. Fargier. Constraint-Based Vehicle Configuration: A Case Study. In *22nd IEEE International Conference on Tools with Artificial Intelligence*, pages 68–75. IEEE, 2010.
11. R. Bashroush, M. Garba, R. Rabiser, I. Groher, and G. Botterweck. CASE Tool Support for Variability Management in Software Product Lines. *ACM Comput. Surv.*, 50(1), 2017.
12. M. Beek, A. Fantechi, and S. Gnesi. Product Line Models of Large Cyber-Physical Systems: The Case of ERTMS/ETCS. In *22nd International Systems and Software Product Line Conference - Volume 1*, pages 208–214. ACM, 2018.
13. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615 – 636, 2010.
14. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *1st International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.
15. D. Benavides, C. Sundermann, S. Vill, K. Feichtinger, , J. A. Galindo, R. Rabiser, and T. Thüm. UVL: Feature Modelling with the Universal Variability Language. Technical report, Elsevier, 2024.
16. D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.
17. J. Bendík and I. Černá. MUST: Minimal Unsatisfiable Subsets Enumeration Tool. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 135–152. Springer, 2020.
18. D. Beuche. Using pure: Variants across the product line lifecycle. In *20th International Systems and Software Product Line Conference*, pages 333–336, New York, NY, USA, 2016. Association for Computing Machinery.
19. B. Bogaerts, E. Gamba, and T. Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:1–21, 2021.
20. L. Brownsword and P. Clements. A Case Study in Successful Product Line Development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.
21. T. Burgstaller, V. Le, T. Tran, and A. Felfernig. FMTesting: A FeatureIDE Plug-in for Automated Feature Model Analysis and Diagnosis. In *12th International Conference on Prestigious Applications of Intelligent Systems*, pages 3190–3195, 2023.
22. R. Burke, A. Felfernig, and M. Göker. Recommender Systems: An Overview. *AI Magazine*, 32(3):13–18, 2011.
23. R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014.
24. M. Cashman, J. Firestone, M. Cohen, T. Thianniwet, and W. Niu. DNA as Features: Organic Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A*, pages 108–118. ACM, 2019.
25. A. E. Chacón-Luna, A. M. Gutiérrez, J. A. Galindo, and D. Benavides. Empirical software product line engineering: a systematic literature review. *Information and Software Technology*, 128:106389, 2020.
26. D. Dhungana, A. Haselböck, R. Ruiz-Torrubiano, and S. Wallner. Variability of Safety Risks in Production Environments. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 178–187. ACM, 2022.
27. C. Dumitrescu, R. Mazo, C. Salinesi, and A. Dauron. Bridging the Gap between Product Lines and Systems Engineering: An Experience in Variability Management for Automotive Model Based Systems Engineering. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, page 254–263, New York, NY, USA, 2013. Association for Computing Machinery.

28. M. Eggert, K. Günther, J. Maletschek, A. Maxiniuc, and A. Mann-Wahrenberg. In Three Steps to Software Product Lines: A Practical Example from the Automotive Industry. In *26th ACM International Systems and Software Product Line Conference - Volume A*, page 170–177. ACM, 2022.
29. A. Falkner, A. Felfernig, and A. Haag. Recommendation Technologies for Configurable Products. *AI Magazine*, 32(3):99–108, 2011.
30. A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Transactions on Engineering Management*, 54(1):41–56, 2007.
31. A. Felfernig. *AI Techniques for Software Requirements Prioritization*, chapter 2, pages 29–47. World Scientific, 2021.
32. A. Felfernig, D. Benavides, J. Galindo, and F. Reinfrank. Towards Anomaly Explanation in Feature Models. In *15th International Configuration Workshop*, volume 1128, 08 2013.
33. A. Felfernig, L. Boratto, M. Stettinger, and M. Tkalcic. *Group Recommender Systems: An Introduction*. Springer Publishing Company Inc., 2nd edition, 2024.
34. A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker. Configuration Knowledge Representations for Semantic Web Applications. *Artif. Intell. Eng. Des. Anal. Manuf.*, 17(1):31–50, 2003.
35. A. Felfernig, T. Gruber, G. Brandner, P. Blazek, and M. Stettinger. Customizing Events with EventHelpr. In *8th International Conference on Mass Customization and Personalization – Community of Europe (MCP-CE 2018)*, pages 88–91, 2018.
36. A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen. *Knowledge-based Configuration – From Research to Business Cases*. Morgan Kaufmann, 2014.
37. A. Felfernig, V. Le, A. Popescu, M. Uta, T. Tran, and M. Atas. An Overview of Recommender Systems and Machine Learning in Feature Modeling and Configuration. In *15th Intl. Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '21. ACM, 2021.
38. A. Felfernig, V. Le, and T. Tran. Supporting Feature Model Based Configuration in Microsoft Excel. In *22nd International Configuration Workshop*, pages 35–38, 2020.
39. A. Felfernig, S. Reiterer, M. Stettinger, F. Reinfrank, M. Jeran, and G. Ninaus. Recommender Systems for Configuration Knowledge Engineering. In *15th International Configuration Workshop*, pages 51–54. AAAI, 2013.
40. A. Felfernig, S. Reiterer, M. Stettinger, and J. Tiihonen. Intelligent Techniques for Configuration Knowledge Evolution. In *9th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 51–58, New York, NY, USA, 2015. ACM.
41. A. Felfernig, S. Reiterer, M. Stettinger, and J. Tiihonen. Towards Understanding Cognitive Aspects of Configuration Knowledge Formalization. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems(VaMoS '15)*, pages 117–123, New York, NY, USA, 2015. Association for Computing Machinery.
42. A. Felfernig, M. Schubert, and C. Zehentner. An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artif. Intell. Eng. Des. Anal. Manuf.*, 26(1):53–62, Feb. 2012.
43. A. Felfernig, M. Stettinger, A. Falkner, M. Atas, X. Franch, and C. Palomares. OpenReq: Recommender Systems in Requirements Engineering. In *2nd Workshop on Recommender Systems and Big Data Analytics (RS-BDA'17)*, pages 1–4, 2017.
44. A. Felfernig, R. Walter, J. Galindo, D. Benavides, M. Atas, S. Polat-Erdeniz, and S. Reiterer. Anytime Diagnosis for Reconfiguration. *Journal of Intelligent Inf. Sys.*, 51:161–182, 2018.
45. A. Felfernig, M. Wundara, T. Tran, S. Polat-Erdeniz, S. Lubos, M. E. Mansi, and D. G. V. Le. Recommender systems for sustainability: overview and research issues. *Frontiers in Big Data*, 6, 2023.
46. P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2021.
47. D. Friesel, K. Elmenhorst, L. Kaiser, M. Müller, and O. Spinczyk. Kconfig-Webconf: Retrofitting Performance Models onto Kconfig-Based Software Product Lines. In *26th ACM International Systems and Software Product Line Conference - Volume B*, pages 58–61. ACM, 2022.

48. C. Fritsch, R. Abt, and B. Renz. The Benefits of a Feature Model in Banking. In *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. ACM, 2020.
49. J. A. Galindo, A. J. Dominguez, J. White, and D. Benavides. Large language models to generate meaningful feature model instances. In *27th ACM International Systems and Software Product Line Conference-Volume 1*, 2023.
50. J. A. Galindo, J.-M. Horcas, A. Felfernig, D. Fernandez-Amoros, and D. Benavides. Flama: A collaborative effort to build a new framework for the automated analysis of feature models. In *27th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '23, pages 16–19, New York, NY, USA, 2023. Association for Computing Machinery.
51. S. Gottschalk, E. Yigitbas, E. Schmidt, and G. Engels. Model-Based Product Configuration in Augmented Reality Applications. In T. Bernhaupt, C. Ardito, and S. Sauer, editors, *Human-Centered Software Engineering*, pages 84–104. Springer, 2020.
52. P. Grasch, A. Felfernig, and F. Reinfrank. ReComment: Towards Critiquing-Based Recommendation with Speech Interaction. In *7th ACM Conference on Recommender Systems*, pages 157–164. ACM, 2013.
53. S. Gupta, B. Genc, and B. O'Sullivan. Explanation in Constraint Satisfaction: A Survey. In Z. Zhi-Hua, editor, *13th International Joint Conference on Artificial Intelligence*, pages 4400–4407. International Joint Conference on Artificial Intelligence Organization, 2021.
54. B. Herzog, F. Hügel, S. Reif, T. Hönig, and W. Schröder-Preikschat. Automated Selection of Energy-Efficient Operating System Configurations. In *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, pages 309–315. ACM, 2021.
55. J. Hofer and M. B. A. Schäfer. Behavioral Customization of State Machine Models at ESO. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 188–198, New York, NY, USA, 2022. ACM.
56. J. Horcas, J. Galindo, M. Pinto, L. Fuentes, and D. Benavides. FM Fact Label: A Configurable and Interactive Visualization of Feature Model Characterizations. In *26th ACM International Systems and Software Product Line Conference - Volume B*, pages 42–45. ACM, 2022.
57. J. Horcas, M. Pinto, and L. Fuentes. Empirical Analysis of the Tool Support for Software Product Lines. *Software and Systems Modeling*, 22:377–414, 2023.
58. J.-M. Horcas, J. A. Galindo, and D. Benavides. Variability in data visualization: a software product line approach. In *26th ACM International Systems and Software Product Line Conference-Volume A*, pages 55–66, 2022.
59. J.-M. Horcas, J. A. Galindo, R. Heradio, D. Fernandez-Amoros, and D. Benavides. A monte carlo tree search conceptual framework for feature model analyses. *Journal of Systems and Software*, 195:111551, 2023.
60. J. Jézéquel, J. Kienzle, and M. Acher. From Feature Models to Feature Toggles in Practice. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 234–244. ACM, 2022.
61. U. Junker. QuickXPlain: Preferred Explanations and Relaxations for over-Constrained Problems. In *AAAI'04*, page 167–172. AAAI Press, 2004.
62. K. Kang, C. Sholom, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
63. M. Kowal, S. Ananieva, and T. Thüm. Explaining anomalies in feature models. *SIGPLAN Not.*, 52(3):132–143, 2016.
64. D. Kramer, C. Sauer, and T. Roth-Berghofer. Towards Explanation Generation using Feature Models in Software Product Lines. In G. Nalepa and J. Baumeister, editors, *9th Workshop on Knowledge Engineering and Software Engineering (KESE9)*, volume 1070. CEUR, 2013.
65. C. Krueger and P. Clements. Feature-Based Systems and Software Product Line Engineering with Gears from BigLever. In *22nd International Systems and Software Product Line Conference - Volume 2*, page 1–4, New York, NY, USA, 2018. Association for Computing Machinery.
66. J. Krüger, S. Nielebock, S. Krieter, C. Diedrich, T. Leich, G. Saake, S. Zug, and F. Ortmeier. Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems. In *21st*

International Systems and Software Product Line Conference - Volume A, pages 237–241, New York, NY, USA, 2017. Association for Computing Machinery.

67. V. Le, A. Felfernig, T. Tran, M. Atas, M. Uta, D. Benavides, and J. Galindo. DirectDebug: A software package for the automated testing and debugging of feature models. *Software Impacts*, 9:100085, 2021.
68. V. Le, A. Felfernig, M. Uta, D. Benavides, J. Galindo, and T. Tran. DirectDebug: Automated Testing and Debugging of Feature Models. In *IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 81–85. IEEE/ACM, 2021.
69. V. Le, A. Felfernig, M. Uta, T. Tran, and C. Silva. WipeOutR: Automated Redundancy Detection for Feature Models. In *26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, pages 164–169, New York, NY, USA, 2022. Association for Computing Machinery.
70. V. Le, T. Tran, and A. Felfernig. A Conversion of Feature Models into an Executable Representation in Microsoft Excel. In *Studies in Computational Intelligence*, volume 949, pages 153–168, Berlin, Heidelberg, 2021. Springer.
71. V.-M. Le, C. V. Silva, A. Felfernig, D. Benavides, J. Galindo, and T. N. T. Tran. Fastdiagp: An algorithm for parallelized direct diagnosis. In *AAAI'23/IAAI'23/EAAI'23*. AAAI Press, 2023.
72. Y. Li, S. Schulze, H. Scherrebeck, and T. Fogdal. Automated Extraction of Domain Knowledge in Practice: The Case of Feature Extraction from Requirements at Danfoss. In *24th ACM International Systems and Software Product Line Conference*, pages 1–11, New York, NY, USA, 2020. ACM.
73. S. Lubos, M. Tautschnig, A. Felfernig, and V. Le. Knowledge-Based Configuration of Videos Using Feature Models. In *26th ACM International Systems and Software Product Line Conference - Volume B*, pages 188–192. ACM, 2022.
74. M. Hentze and T. Pett and T. Thüm and I. Schaefer. Hyper Explanations for Feature-Model Defect Analysis. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery.
75. N. Mani, M. Helfert, and C. Pahl. A Domain-specific Rule Generation Using Model-Driven Architecture in Controlled Variability Model. *Procedia Computer Science*, 112:2354–2362, 2017.
76. A. G. Márquez, Á. J. Varela-Vaca, M. T. G. López, J. A. Galindo, and D. Benavides. Vulnerability impact analysis in software project dependencies based on satisfiability modulo theories (smt). *Computers & Security*, 139:103669, 2024.
77. R. Medeiros, O. Díaz, and D. Benavides. Unleashing the power of implicit feedback in software product lines: Benefits ahead. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 113–121, 2023.
78. M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.
79. D. Munoz, M. Pinto, and L. Fuentes. Quality-Aware Analysis and Optimisation of Virtual Network Functions. In *26th ACM International Systems and Software Product Line Conference - Volume A*, pages 210–221. ACM, 2022.
80. R. L. na, M. Ballarin, and C. Cetina. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *20th International Systems and Software Product Line Conference*, pages 194–203. ACM, 2016.
81. J. Navarro and J. Chavarriaga. Using Microsoft Solver Foundation to Analyse Feature Models and Configurations. In *8th Euro American Conference on Telematics and Information Systems (EATIS)*, pages 1–8, 2016.
82. N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *13th Intl. Conference on Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543. Springer, 2007.

83. T. Niederer, D. Schloss, and N. Christensen. Designing Context-Aware Chatbots for Product Configuration. In A. Følstad, T. Araujo, S. Papadopoulos, E. Law, E. Luger, M. Goodwin, and P. Brandtzaeg, editors, *Chatbot Research and Design*, pages 190–210, Cham, 2023. Springer.
84. E. OliveiraJr and D. Benavides. Principles of software product lines. In *UML-Based Software Product Line Engineering with SMarty*, pages 3–26. Springer, 2022.
85. J. Pereira, M. Acher, H. Martin, J. Jézéquel, G. Botterweck, and A. Ventresque. Learning Software Configuration Spaces: A Systematic Literature Review. *Journal of Systems and Software*, 182:111044, 2021.
86. J. Pereira, S. Krieter, J. Meinicke, R. Schröter, G. Saake, and T. Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *15th International Conference on Software Reuse: Bridging with Social-Awareness*, volume 9679, pages 397–401, Berlin, Heidelberg, 2016. Springer.
87. J. Pereira, P. Matuszyk, S. Krieter, S. piliopoulou, and G. Saake. Personalized Recommender Systems for Product-Line Configuration Processes. *Comput. Lang. Syst. Struct.*, 54(C):451–471, 2018.
88. J. Pereira, P. Matuszyk, S. Krieter, M. Spiliopoulou, and G. Saake. A Feature-Based Personalized Recommender System for Product-Line Configuration. *SIGPLAN Not.*, 52(3):120–131, 2016.
89. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering*, volume 10. Springer, 2005.
90. A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V. Le, K. Pilsl, M. Enzelsberger, and T. Tran. An Overview of Machine Learning Techniques in Constraint Solving. *Journal of Intelligent Inf. Sys.*, 58(1):91–118, 2022.
91. C. Prud'homme and J. Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):1–6, 2022.
92. M. Raatikainen, J. Tiihonen, T. Männistö, A. Felfernig, M. Stettinger, and R. Samer. Using a Feature Model Configurator for Release Planning. In *22nd International Systems and Software Product Line Conference - Volume 2*, pages 29–33. ACM, 2018.
93. R. Reiter. A Theory of Diagnosis From First Principles. *AI Journal*, 32(1):57–95, 1987.
94. J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, and D. Benavides. Selection of Software Product Line Implementation Components Using Recommender Systems: An Application to Wordpress. *IEEE Access*, 7:69226–69245, 2019.
95. D. Romero-Organvidez, D. Benavides, J.-M. Horcas, and M. T. Gómez-López. Variability in data transformation: towards data migration product lines. In *18th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 83–92, 2024.
96. D. Romero-Organvidez, J. Galindo, C. Sundermann, J. Horcas, and D. Benavides. UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles. Technical report, Elsevier, 2023.
97. V. Rothberg, N. Dintzner, A. Ziegler, and D. Lohmann. Feature Models in Linux: From Symbols to Semantics. In *VaMoS '16*, pages 65–72. ACM, 2016.
98. D. Sabin and R. Weigel. Product Configuration Frameworks – A Survey. *IEEE Intelligent Systems and their Applications*, 13(4):42–49, 1998.
99. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14:477–495, 2012.
100. S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, page 63–71, New York, NY, USA, 2012. Association for Computing Machinery.
101. J. Sincero and W. Schröder-Preikschat. The Linux Kernel Configurator as a Feature Modeling Tool. In *21st Workshop on Analyses of Software Product Lines (ASPL'08)*, pages pages 257–260, 2008.
102. M. Stumptner. An Overview of Knowledge-Based Configuration. *AICom*, 10(2):111–125, 1997.

103. C. Sundermann, S. Vill, T. Thüm, K. Feichtinger, P. Agarwal, R. Rabiser, J. A. Galindo, and D. Benavides. Uvlparser: Extending uvl with language levels and conversion strategies. In *27th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '23, pages 39–42, New York, NY, USA, 2023. Association for Computing Machinery.
104. P. Temple, J. Galindo, M. Acher, and J. Jézéquel. Using Machine Learning to Infer Constraints for Product Lines. In *20th International Systems and Software Product Line Conference*, pages 209–218, New York, NY, USA, 2016. Association for Computing Machinery.
105. T. Thüm. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. ACM, 2020.
106. T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, Jan. 2014.
107. T. Tran, M. Atas, A. Felfernig, V. Le, R. Samer, and M. Stettinger. Towards Social Choice-Based Explanations in Group Recommender Systems. In *UMAP '19*, pages 13–21, New York, NY, USA, 2019. Association for Computing Machinery.
108. T. Tran, A. Felfernig, and V. Le. An overview of consensus models for group decision-making and group recommender systems. *User Model User-Adap Inter*, 2023.
109. T. Tran, A. Felfernig, V. Le, M. Atas, M. Stettinger, and R. Samer. User Interfaces for Counteracting Decision Manipulation in Group Recommender Systems. In *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization*, pages 93–98, New York, NY, USA, 2019. ACM.
110. T. Tran, A. Felfernig, V. Le, T. Chau, and T. Mai. User Needs for Explanations of Recommendations: In-Depth Analyses of the Role of Item Domain and Personal Characteristics. In *31st ACM Conference on User Modeling, Adaptation and Personalization*, pages 54–65. ACM, 2023.
111. P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008. Agile Product Line Engineering.
112. T. Ulz, M. Schwarz, A. Felfernig, S. Haas, A. Shehadeh, S. Reiterer, and M. Stettinger. Human Computation for Constraint-Based Recommenders. *J. Intell. Inf. Syst.*, 49(1):37–57, 2017.
113. M. Uta, A. Felfernig, D. Helic, and V. Le. Accuracy- and Consistency-Aware Recommendation of Configurations. In *26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, page 79–84. ACM, 2022.
114. M. Uta, A. Felfernig, V. Le, T. Tran, D. Garber, S. Lubos, and T. Burgstaller. Knowledge-based Recommender Systems: Overview and Research Directions. *Frontiers in Big Data*, 7:1–30, 2024.
115. M. Weckesser, M. Lochau, and M. R. A. Schürr. Mathematical Programming for Anomaly Analysis of Clafer Models. In *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 34–44, New York, NY, USA, 2018. ACM.
116. J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. Automated Diagnosis of Feature Model Configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010. SPLC 2008.
117. L. Wozniak and P. Clements. How Automotive Engineering is Taking Product Line Engineering to the Extreme. In *19th International Conference on Software Product Lines*, pages 327–336, New York, NY, USA, 2015. ACM.
118. H. Xu, S. Baarir, T. Ziadi, L. Hillah, S. Essodaigui, and Y. Bossu. Optimization of the Product Configuration System of Renault. In *38th ACM/SIGAPP Symposium on Applied Computing*, pages 1486–1489. ACM, 2023.

Index