Proceedings

19th International Working Conference on Variability Modelling of Software-Intensive Systems

– VaMoS 2025 –

February 4–6, 2025
Rennes, France

VaMoS 2025 General Chair
Mathieu Acher, University of Rennes, France

VaMoS 2025 Program Committee Chairs
Juliana Alves Pereira, PUC-Rio, Brazil
Clément Quinton, University of Lille, France

Association for
Computing Machinery

Advancing Computing as a Science & Profession

The Association for Computing Machinery
1601 Broadway, 10th Floor
New York, New York 10019, USA

ACM ISBN: 979-8-4007-1441-2

Foreword

This volume contains the proceedings of the 19th International Working Conference on Variability Modelling of Software-intensive Systems (VaMoS'25), which took place in Rennes, France, February 3-6, 2025.

The previous editions of the event were held in Bern (2024), Odense (2023), Florence (2022, virtually), Krems (2021, virtually), Magdeburg (2020), Leuven (2019), Madrid (2018), Eindhoven (2017), Salvador (2016), Hildesheim (2015), Nice (2014), Pisa (2013), Leipzig (2012), Namur (2011), Linz (2010), Sevilla (2009), Essen (2008), and Limerick (2007).

Supported by its strong community, the 18 previous editions of the VaMoS workshop successfully bootstrapped research on modelling and managing variability of software systems, as witnessed by the many related breakthroughs published in top-tier conferences and journals. VaMoS aims to bring together researchers and practitioners to share ideas, results and experiences about their quest for mastering variability.

VaMoS 2025 started with the 8th International Workshop on Languages for Modelling Variability (MODEVAR 2025). Similar to previous years, VaMoS 2025 invited different types of contributions with the aim of expanding the community and further stimulating discussions: The *Technical track* for original research, the *Variability-in-Practice track* for real-world problems and solutions, and the *New and Controversial Ideas track* for short talks on new or positional topics.

VaMoS 2025 continued the artifact track initiated in the previous edition. This track aims to foster reusability in the variability modelling field. Through this track, researchers can actively contribute to open science in software engineering research. Authors of accepted papers were encouraged to submit their artifacts for evaluation in this track. The considered artifacts included (but were not limited to) any dataset, tool, script, experimental protocol, codebook, or other executable or non-executable object produced by or used. For this second edition, the track again relied on the badges defined by the ACM: Available, Functional, Reusable.

Most of today's software is made variable to allow for more adaptability and economies of scale, while many development practices (such as DevOps, A/B testing, parameter tuning, or continuous integration) support this goal of engineering software variants. VaMoS is the ideal venue to explore the underlying problems (such as automation, traceability, or combinatorial explosion) and their solutions. Moreover, variability is prominent in many systems – not only in software. For these reasons, we welcomed contributions from related areas (such as configuration, configurable systems, product lines, adaptive systems, or generators) as well as from new and emerging domains where techniques to manage complexity introduced by variability are applied (such as IoT systems, 3D printing, autonomous and deep learning systems, software-defined networks, security, generative art, games, or research software). Topics of interest included, but were not limited to:

- Variability Modeling and Realization
- Variability in Emerging Domains/Technologies
- Automated Reasoning on Variability
- Variability in AI Methods
- AI for Variability (Generative AI, Large Language Models)
- Variability for Sustainability
- Sustainable Variability
- Variability Across the Software Lifecycle

- Variability in Adaptive Systems
- Test and Verification of Variable Systems
- Configuration Management
- Evolution of Variability-Intensive Systems
- Runtime Variability
- Variability Mining
- Visualization Techniques for Variability
- Reverse-Engineering of Variability
- Economic Aspects of Variability
- Variability and Quality Requirements
- Industrial Development of Variable Systems
- Experience Reports from Managing Variability in Practice
- Variability for Security

VaMoS 2025 received 31 submissions overall across its paper tracks. These underwent a rigorous peer review process with each submission receiving reviews from at least 3 reviewers. After a discussion phase, the Program Committee decided to accept 10 Research Papers, 7 Tool Papers, and 2 New Ideas Papers, covering a wide variety of topics. In the Artifact Evaluation track, 8 artifacts were submitted, and all 8 were accepted for evaluation. Among the accepted artifacts, 8 received the Available badge, 5 received the Functional badge, and 3 received the Reusable badge. We thank the Program Committee members and all additional reviewers for providing detailed feedback and for participating in the discussions.

The VaMoS 2025 conference featured two keynote talks. We thank our speakers for accepting our invitation and for sharing their expertise at the conference:

- Camille Maumet, Variability in Brain Imaging Studies Across Different Analysis Pipelines
- Hugo Guillermo Chale-Gongora, Model-Based PLE and Variability Modeling – Which is Yin and Which is Yang?

This year, the paper “Code Smells Revisited: A Variability Perspective” by Wolfram Fenske (University of Magdeburg, Germany) and Sandro Schulze (University of Magdeburg, Germany) was selected as the Most Influential Paper. The paper was originally presented at VaMoS 2015.

VaMoS 2025 also recognized outstanding contributions with the following awards:

- Best Research Paper: “Teach Variability! A Modern University Course on Software Product Lines” by Elias Kuiter, Thomas Thüm and Timo Kehrer.
- Best Paper Reviewer: Sandra Greiner (University of Southern Denmark & University of Regensburg, Germany).
- Best Artifact Evaluation Reviewer: Philippe Collet (University of Nice Sophia Antipolis, France).

Like previous VaMoS editions, this year's event was a highly interactive event. Each session provided moderated discussions and also involved the paper presenters as discussants. We are grateful to the Local Arrangements chairs Djamel Eddine Khelladi and Paul Temple, the Publicity chair Olivier Zendra, and the local organization team for their help in organizing VaMoS 2025. Finally, we thank the Steering Committee that entrusted us with the organization of the conference.

Mathieu Acher
University of Rennes, France

Juliana Alves Pereira
PUC-Rio, Brazil
Clément Quinton
University of Lille, France

Organization Committee

General Chair

- Mathieu Acher, University of Rennes, France

Program Chairs

- Juliana Alves Pereira, PUC-Rio, Brazil
- Clément Quinton, University of Lille, France

Local Arrangements Chairs

- Djamel Eddine Khelladi, CNRS, France
- Paul Temple, University of Rennes, France

Publicity Chair

- Olivier Zendra, Inria, France

Local Organization

- Charly Reux, PhD student, University of Rennes
- Heraldo Pimenta Borges Filho, postdoc, University of Rennes
- Jolan Philippe, postdoc, University of Rennes
- Romain Lefeuvre, PhD student, University of Rennes

Program Committee (Research Track)

- Alexander Felfernig – TU Graz, Austria
- Camille Salinesi – CRI, Université de Paris 1 Panthéon-Sorbonne, France
- Goetz Botterweck – Trinity College Dublin, Lero, Ireland
- José A. Galindo – University of Sevilla, Spain
- Klaus Schmid – University of Hildesheim, Germany
- Sophie Fortz – PReCISE, NaDI, Université de Namur, Belgium
- Tewfik Ziadi – Sorbonne Université-CNRS 7606, LIP6, France
- Timo Kehrer – University of Bern, Switzerland
- Xhevahire Tërnava – Télécom Paris, IP Paris, France
- Sébastien Mosser – McMaster University, Canada
- Leopoldo Teixeira – Federal University of Pernambuco, Brazil
- Wesley K. G. Assunção – North Carolina State University, USA
- David Benavides – University of Sevilla, Spain
- Jihyun Lee – Jeonbuk National University, South Korea
- Jacob Krüger – Eindhoven University of Technology, Netherlands
- Gilles Perrouin – Université de Namur, Belgium
- Ivan Machado – Federal University of Bahia, Brazil
- Vander Alves – University of Brasília, Brazil
- Jabier Martinez – Tecnalia, France
- Sandra Greiner – University of Southern Denmark & University of Regensburg, Germany
- Christoph Elsner – Siemens AG
- Mónica Pinto – ITIS Software, University of Málaga, Spain
- Jacopo Mauro – University of Southern Denmark, Denmark
- Roberto Lopez-Herrejon – École de Technologie Supérieure de Montreal, Canada
- Deepak Dhungana – IMC University of Applied Sciences, Austria
- Djamel A. Seriai – University of Montpellier, France
- Philippe Collet – University of Nice Sophia Antipolis, France

Program Committee (Artifact Evaluation)

- Vander Alves – University of Brasília, Brazil
- Alexander Boll – University of Bern, Switzerland
- Philippe Collet – University of Nice Sophia Antipolis, France
- Samuel Dubuisson – University of Lille, France
- José A. Galindo – University of Sevilla, Spain
- Sandra Greiner – University of Southern Denmark & University of Regensburg, Germany
- Jihyun Lee – Jeonbuk National University, South Korea
- Gilles Perrouin – Université de Namur, Belgium
- Mónica Pinto – ITIS Software, University of Málaga, Spain
- Leopoldo Teixeira – Federal University of Pernambuco, Brazil
- Xhevahire Tërnava – Télécom Paris, IP Paris, France
- Nada Zine – University of Lille, France

Table of Contents

Technical Papers

Tool, Demo, and Experience Report Papers

New and Controversial Ideas

Keynotes

Variability in Brain Imaging Studies Across Different Analysis Pipelines

Camille Maumet (Inria Rennes Bretagne Atlantique / IRISA)
February 4, 2025

Abstract. Neuroimaging studies are characterized by a very large analysis space, and practitioners usually have to choose between different software, software versions, algorithms, parameters, etc. For many years, these choices have been regarded as implementation details, but it is becoming increasingly clear that the exact choices of analytical strategy can lead to different and sometimes contradictory results. This keynote discusses reproducibility in the field of brain imaging and our recent efforts to better understand and manage the different sources of this analytical variability.

Bio. Camille Maumet is a research scientist in neuroinformatics at the Empenn team, Inria Rennes Bretagne Atlantique / IRISA. She studies neuroimaging reproducibility with a focus on variability of analytical pipelines and its impact on our ability to reuse brain imaging datasets. She is an open science advocate and participates in international communities including Brainhack, INCF, and OHBM Open Science SIG.

Model-Based PLE and Variability Modeling – Which is Yin and Which is Yang?

Hugo Guillermo Chale-Gongora (Airbus)
February 5, 2025

Abstract. Variability management and modelling (VM) play a central role in feature-based and model-based product line engineering (MBPLE), but VM alone is not enough to guarantee successful PLE in large organizations. This keynote explores the long journey of PLE adoption at Airbus, offering lessons learned and a holistic approach to MBPLE that aligns marketing, business analysis, architecting, design, and engineering.

Bio. Hugo Guillermo Chale-Gongora is the Head of PLE & Multidisciplinary Analysis and Optimisation at Airbus. He brings practical insights from deploying model-based PLE in large-scale industrial contexts.

Teach Variability! A Modern University Course on Software Product Lines

Elias Kuiter
University of Magdeburg
Magdeburg, Germany
kuiter@ovgu.de

Thomas Thüm
TU Braunschweig
Braunschweig, Germany
t.thuem@tu-braunschweig.de

Timo Kehrer
University of Bern
Bern, Switzerland
timo.kehrer@inf.unibe.ch

Abstract

Teaching software product lines to university students is key in disseminating knowledge about software variability. In particular, education is needed to train new researchers and practitioners and, thus, sustain further research on software product lines. However, preparing appropriate teaching material is difficult and time-consuming, even when relying on existing literature. Thus, clone-and-own is a common practice among educators, with all its associated issues. Moreover, there is a lack of full-semester, open courses on software product lines. In this paper, we report on our experience of architecting and designing such a course from scratch, avoiding clone-and-own entirely. In addition, we perform a literature review of influential books on software product lines and which topics they cover. We position our course in terms of these topics, discuss how it compares to existing courses, and justify relevant design decisions. With our course, we aim to strengthen the positive interactions between research, industry, and education. So far, our course has already been held seven times across five universities. A preliminary evaluation of our course indicates that our course is mostly well-received by students.

CCS Concepts

• Social and professional topics → Software engineering education; **• Software and its engineering → Software product lines**.

Keywords

open educational resources, software product lines

ACM Reference Format:
Elias Kuiter, Thomas Thüm, and Timo Kehrer. 2025. Teach Variability! A Modern University Course on Software Product Lines. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3715340.3715441

1 Introduction

Software variability is ubiquitous, as many software systems are configurable [11, 12, 45, 65]. *Software product lines (SPLs)* [7, 20, 56, 67] are a well-known paradigm in research and practice [5, 10, 19, 26, 30, 54, 59] that apply systematic reuse to manage such variability. SPLs promise reduced costs for development and maintenance, faster time-to-market, and improved quality [7, 36, 40, 67].

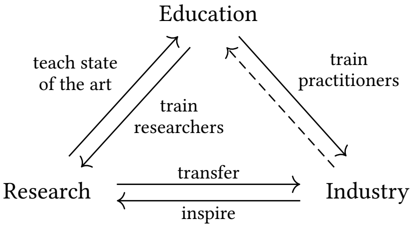


Figure 1: Interactions of research, industry, and education.

Education at universities plays an important role in disseminating knowledge about SPL engineering [2]. First, it gives students the required tools to identify opportunities for beneficial software reuse in industry. Second, it is instrumental in teaching the next generation of SPL researchers. Third, education interacts positively with research, as students can participate in experiments and research projects, the results of which might then feed back into education. In Figure 1, we visualize these interactions between research, industry, and education as they can be observed in the context of SPLs. One arrow is less pronounced: This is because the industry typically has limited direct influence on education, besides occasional guest lectures. Thus, the responsibility to teach students appropriate skills mostly lies with educators (often researchers [2]), who need to balance the interests of both research and industry.

To achieve this balancing act, having appropriate teaching material is key. However, preparing such material is difficult and time-consuming, even when relying on existing books [2, 51]. Moreover, educators perceive that there is a lack of recognition for such efforts [60]. Thus, a common strategy among educators has been to use existing teaching material and only slightly adapt it [2] (e.g., by adding topics of their own). While such a clone-and-own [6, 24] strategy is tempting, over time it might lead to outdated or incorrect information, scope creep, and licensing issues. As long-time maintainers of clone-and-own teaching material, we can anecdotally confirm that such issues have repeatedly come up over the years.

Besides clone-and-own issues, creators of SPL teaching material seem to struggle with completeness and openness. To illustrate these problems, we collect SPL teaching material that is publicly available, and we show all complete English courses that we find in Table 1. In particular, this table is quite short with five entries, which is likely due to two reasons: First, much SPL teaching material only consists of one or two lectures in the context of a larger

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715441

Table 1: Publicly available complete English courses on SPLs.

Authors	Year	University	Literature	Open?
Acher and Heymans[1]	2011	Namur	–	○
Kästner and Apel[2]	2015	Pittsburgh	[7]	○
Lopez-Herrejon and Rabiser[3]	2016	Linz	[20, 56, 67]	○
Donohoe and Northrop[4]	2020	Pittsburgh	[20]	◐
Gay and Berger[5]	2022	Gothenburg	[7, 67]	●
Thüm, Kehrer, and Kuiter[7]	2024	*(6 universities)*	[7, 50]	●

○ License unclear, no sources ◐ Open license ● Open license, sources available

We consider all material that is available in the online repository[6] of Acher et al. [3]. In addition, we consider material we found with the following Google search: `"software␣" ("product␣line" | "variability") ("␣course" | "␣slides")` We only include complete English courses that are mostly concerned with SPL topics. A more detailed version is available in our online appendix.[7]

course (e.g., as a cross-cutting concern in a course on software engineering). While individual lectures can teach the basics [21, 52], we believe that full-semester courses on SPLs are necessary to positively contribute to the interactions depicted in Figure 1. Second, many courses on SPLs are never released publicly [18, 61] (e.g., due to clone-and-own issues). Even for publicly released courses, their license may be unclear or their sources not available, as we observe for some courses[1−3] in Table 1. This severely limits the material's potential for adaptation and reuse. While the next course[4] is openly licensed (CC-BY-4.0), its sources are not available, so no custom adaptations can be made. Indeed, we are only aware of one complete course on SPLs[5] with published and openly-licensed (CC-BY-SA-4.0) sources available. However, this course has only been released in PowerPoint format (originally Google Slides). While these tools allow for collaboration to some degree, they still invite problematic clone-and-own practices for performing adaptions, and therefore limits the course's reusability. We contacted an author of each course in Table 1 and asked them whether they are aware of their course being adapted to other universities (not considering affiliation changes of the course authors). We found that only one course[3] had been adapted to another university (i.e., Karlsruhe).

Evidently, the SPL community is missing a collaborative effort to create a new, modern full-semester university course on SPLs, which should be openly licensed and widely applicable. In this paper, we strive to fill this gap by reporting on our experience of architecting and designing such a course. Our aim is twofold: First, we want to avoid the above-mentioned problems of clone-and-own with existing courses. Second, we want to offer a modern SPL curriculum based on practical topics and recent research results. Thus, we decided to create a new course from scratch (i.e., proactively [41]). Since September 2022, our course has already been held seven times across five universities (i.e., in Bern, Ulm, Wernigerode, Magdeburg, and Paderborn). Currently, three new iterations of the course are ongoing, one of them at a new, sixth university (i.e., in Braunschweig) with more than one hundred course participants.

In particular, we contribute the following:

- We publish a course on SPLs with slides for 12 new 90-minute lectures, which we created and refined over the last three years.[7] Our slides are released under the permissive CC-BY-SA-4.0 license[8] and can be easily adapted to other universities with LaTeX. We also publish video recordings of each lecture under this license.[9]
- We describe and justify the scope and goals of our course (cf. Section 2), how we align its architecture (cf. Section 3) and lecture design (cf. Section 4) accordingly, and how we address several potential adoption challenges (cf. Section 5).
- We perform a literature review of topics covered in influential books on SPLs. Based on this review, we discuss how our course addresses each topic, and where we deviate from the books. Using our review, educators can choose suitable literature and avoid redundancies when they create new SPL teaching material. We also compare the contents of our course to existing courses (cf. Table 1), so educators can easily decide which material to use.
- We perform a preliminary evaluation of our course, which is based on feedback from 64 students across six teaching evaluations. We find that our course is mostly well-received by students and that it performs favorably in comparison to other courses.

With this, we build on a previous survey of SPL teaching practices by Acher et al. [2], who identified the need for such a curriculum.

2 Course Scope and Goals

What constitutes a modern course on SPLs? In the following, we define four goals ($\mathbf{G}_{1-4}$) that narrow the format and scope of our course. We justify why we deem the chosen goals to be reasonable premises to build our course on. We begin with generic goals and get more specific, such that each goal is derived from previous ones.

$\mathbf{G}_1$ Connect Research, Industry, and Education As we show in Figure 1, education plays a key role in connecting research and industry. To support and strengthen this connection, we aim to:

- describe the state of the art, prioritizing recent sources [18]
- include insights from recent research, pointers for further reading, and references for claims
- discuss open challenges, which lay a foundation for lectures on research (to train researchers) or industry (to train practitioners)

As we are researchers, our perspective is bound to be rather research-oriented. While we largely embrace this perspective, we also aim to recount experiences from industry for motivation (e.g., based on our collaborations with industry partners).

$\mathbf{G}_2$ Invite Contributions We aim to release our course in form of *open educational resources.* UNESCO [66] defines these as "learning, teaching and research materials in any format and medium that reside in the public domain or are under copyright that have been released under an open license, that permit no-cost access, re-use, re-purpose, adaptation and redistribution by others." This promises several benefits over a non-distributed, closed-source course [51]:

- It enables students to participate who are not enrolled at a university or whose university does not offer a course on SPLs.

[1] http://teaching.variability.io/namur.html
[2] http://www.cs.cmu.edu/~ckaestne/17708
[3] http://teaching.variability.io/jku2016.html
[4] https://insights.sei.cmu.edu/library/introduction-to-software-product-lines-course
[5] https://greg4cr.github.io/courses/fall22tda594
[6] https://teaching.variability.io
[7] Artifact with online appendix: https://doi.org/10.5281/zenodo.14417094
Repository: https://github.com/SoftVarE-Group/Course-on-Software-Product-Lines
[8] https://creativecommons.org/licenses/by-sa/4.0/
[9] https://www.youtube.com/playlist?list=PL4hJhdKDPIxha8So7muX2zfNUU8NBoiu3

- It invites fellow educators and researchers to reuse and adapt our course or even make new contributions (e.g., new lectures).
- It allows practitioners to teach SPL concepts to industrial stakeholders (e.g., domain experts) without any licensing issues.
- It holds us and other contributors publicly accountable, creating an incentive to commit high-quality and up-to-date material.

Moreover, each of these benefits contributes to goal $\mathbf{G}_1$ by either making our course more attractive to some party in Figure 1 or establishing accountability and, thus, confidence in the material.

$\mathbf{G}_3$ Address a Broad Audience Preparing new teaching material is a difficult and time-consuming endeavor. Consequently, it makes sense for us to try to address a broad audience of students, so our course is widely applicable in practice. Thus, we aim to:

- choose a course format that fits a typical university schedule [2]
- apply modern teaching methods that attract students [18]
- architect an inductive course structure that emphasizes the natural discovery of concepts over their definitions [57]

We believe these efforts will help to attract educators, making it way easier to teach SPLs at more universities (as per $\mathbf{G}_2$).

$\mathbf{G}_4$ Focus on Practical Skills SPLs have many facets [2, 59], not all of which can be realistically taught in a single university course. Consequently, we must choose a particular subset of topics and skills that is both likely to be relevant for training new researchers and practitioners (as per $\mathbf{G}_1$) and to appeal to students (as per $\mathbf{G}_3$). For this reason, we opt for a practical, hands-on approach and focus mostly on topics like modeling, implementation, and analysis of variability. In particular, we put less emphasis on management and organizational topics. While these topics are no less important, they are also more abstract and difficult to grasp for students, which typically have limited industrial experience. Overall, we therefore aim to create a rather technical course that can be accompanied by a suitable exercise class with, for example, programming tasks.

Building on goal $\mathbf{G}_1$–$\mathbf{G}_4$, we describe and justify the high-level (cf. Section 3) and low-level (cf. Section 4) design of our course.

3 Course Architecture

Initially, the authors of this paper spent $\approx$ 11 months only discussing the course format and structure, and which SPL topics to cover in which depth. In the following, we briefly describe the relevant design choices we made, which concern our course as a whole.

3.1 Format

Our course consists of 12 English lectures ($\mathbf{L}_{1-12}$), with $\approx$ 40 slides each. This format fits a typical weekly (under-)graduate university course of 3–4 months with 90-minute lectures. We create an entire series of lectures to close the existing gap in complete, open courses on SPLs (cf. Table 1). We use the English language and an established course format for improved applicability ($\mathbf{G}_3$). The schedule with 12 lectures leaves enough room for guest lectures ($\mathbf{G}_1$). For example, we typically host one additional lecture with conference talks as well as an industrial guest lecture (e.g., by pure-systems GmbH).

3.2 Literature Review

Similar to other courses [2] (e.g., those shown in Table 1), we loosely base our lecture slides on existing books. We do this for several reasons: First, it saves time and effort, as several books already prepare a curriculum with a well-laid-out common thread. Second, given such a curriculum, we can decide more easily where to deviate from it (cf. Section 4), for example to include recent research or correct outdated information ($\mathbf{G}_1$). Third, referring to books gives interested students material for further reading ($\mathbf{G}_1$).

While searching for appropriate literature (as per $\mathbf{G}_4$), we became aware that a comprehensive overview of educational books on SPLs is missing. Moreover, we are not aware of any detailed review of topics that are relevant for teaching SPLs, and how books cover these topics. To fill this gap, we perform a literature review of influential (i.e., well-known and often-cited) books on SPLs. We also review relevant SPL topics and the degree to which each book covers them. We show both the methodology for this review and its results in Table 2. A more in-depth version is available in our online appendix,[7] which also includes justifications and a list of excluded books. Our literature review serves as a first reference to help educators (such as ourselves) choose suitable literature for new courses and pointers for further reading.

For our course, we choose Apel et al. [7] and Meinicke et al. [50] as accompanying literature, due to several reasons: First, both books are well-known and comparably up-to-date ($\mathbf{G}_1$) as well as mostly oriented towards practical topics ($\mathbf{G}_3$). Second, they include both theoretical [7] and practical [50] exercises ($\mathbf{G}_3$) and are likely accessible to university students (e.g., via SpringerLink). The practical exercises, in particular, rely on the tool FeatureIDE [38], which is free and open-source software ($\mathbf{G}_2$) and already used in teaching [2]. Third, our literature review in Table 2 shows that of all considered books, the chosen two focus most on our topics of interest, such as variability modeling, implementation, and analysis ($\mathbf{G}_4$).

3.3 Structure

We divide our course into three parts, which we show on the top in Figure 2. In Part I ($\mathbf{L}_{1-3}$), we begin with ad-hoc approaches for software variability, which many students will be intuitively familiar with. Then, in Part II ($\mathbf{L}_{4-8}$), we introduce feature modeling [7, 34], implementation techniques [7, 64], and the development process [7, 56]. Finally, we discuss measures for quality assurance and advanced topics in Part III ($\mathbf{L}_{9-12}$). This structure is inductive (i.e., it builds concepts up from basic principles, $\mathbf{G}_3$) and emphasizes practicality ($\mathbf{G}_4$). In particular, we introduce feature models (in $\mathbf{L}_4$) and the SPL development process (in $\mathbf{L}_8$) much later than Apel et al. [7], when students already know how to apply them.

4 Lecture Design

After settling on the course architecture (cf. Section 3), we spent four months creating initial versions of most lectures. Below, we discuss the detailed lecture design, relating it to our goals (cf. Section 2).

4.1 Structure

We divide each lecture into three blocks, which we show exemplary for the introduction ($\mathbf{L}_1$) on the bottom in Figure 2. Each block is designed to take 20–25 minutes of lecturing time, followed by an optional interaction with the audience of 5–10 minutes. In each block, we describe and discuss one distinct and cohesive topic related to SPLs. This discussion is then concluded by a summary of

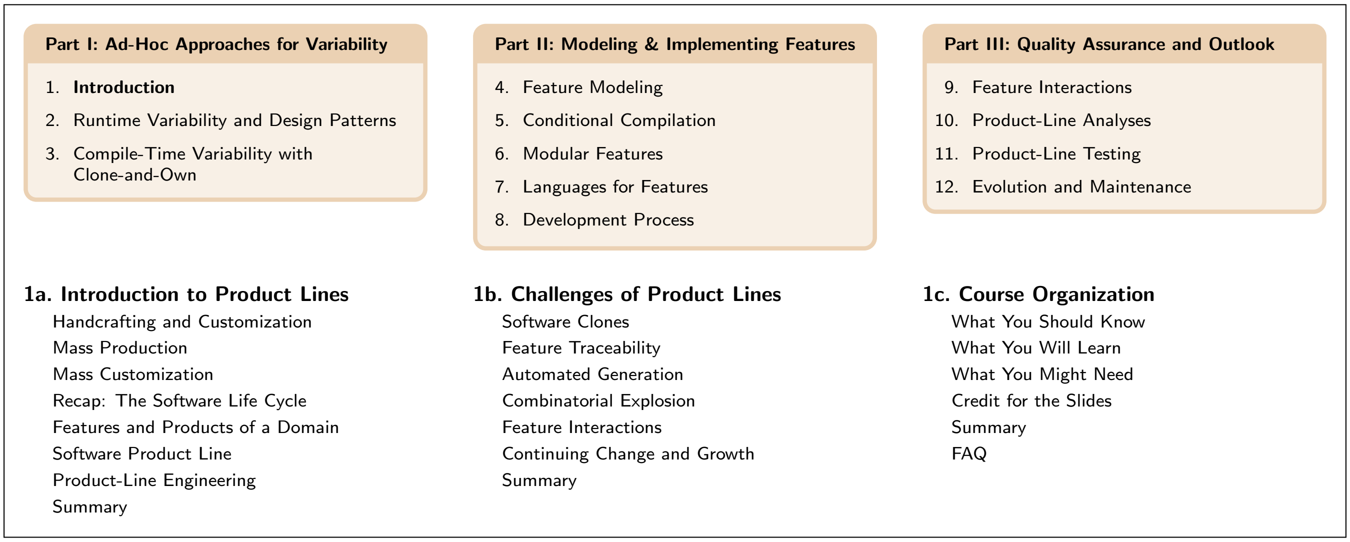
Table 2: Literature review of existing books and courses on SPLs, which topics they cover, and how they compare to our course.

Topics This list of topics is based on the books' and courses' tables of contents, the session titles of recent conferences (SPLC [8], VaMoS [35], and FOSD), and also on related work [2, 59]. In line with our practical focus (**G**$_4$), we break down topics related to modeling, implementation, and analysis in more detail and put less emphasis on management issues.	**Books**								**Courses**					
	Czarnecki and Eisenecker [23]	Bosch [15]	Clements and Northrop [20]	Gomaa [31]	Pohl, Böckle, and van der Linden [56]	van der Linden, Schmid and Rommes [67]	Apel, Batory, Kästner, and Saake [7]	Meinicke, Thüm, Schröter, Benduhn, Leich, and Saake [50]	Acher and Heymans[1]	Kästner and Apel[2]	Lopez-Herrejon and Rabiser[3]	Donohoe and Northrop[4]	Gay and Berger[5]	Thüm, Kehrer, and Kuiter[7]
Fundamentals														
Motivation, Goals, Context, History	●	●	●	●	●	●	●	●	●	●	●	●	●	●
SPL Definition, Delineation	◐	●	●	●	●	●	●	◐	●	●	●	●	●	●
SPL Engineering	○	●	●	●	●	●	●	◐	●	●	●	●	●	●
Modeling and Configuration														
Feature Modeling	●	○	◐	●	●	◐	●	●	●	◐	●	○	●	●
Decision Modeling	○	○	○	○	○	◐	○	○	○	◐	●	○	○	○
Product Configuration	○	●	●	●	◐	●	●	●	●	◐	●	●	●	●
Requirements Engineering	○	●	●	●	●	●	◐	○	○	●	◐	●	◐	◐
Scoping, Variability Reduction	●	●	●	●	●	●	◐	○	◐	●	●	●	◐	◐
Variability-Model Representations, Transformations	○	○	○	◐	○	○	●	◐	●	◐	◐	○	◐	●
Model-Driven Engineering, Domain-Specific Languages	●	○	○	◐	◐	◐	○	○	●	●	◐	○	○	○
Design and Implementation														
Product Derivation, Automation, Generative Programming	●	◐	◐	○	◐	◐	●	●	●	●	●	◐	●	●
Feature Mapping, Traceability, Location	●	○	○	○	◐	◐	◐	◐	◐	◐	◐	○	◐	◐
Runtime Variability, Design Patterns	●	◐	◐	●	◐	◐	●	●	◐	●	◐	◐	●	●
Clone-and-Own, Version Control Systems	○	◐	◐	○	○	○	◐	◐	○	●	◐	○	○	●
Preprocessors	◐	○	○	◐	◐	◐	●	●	◐	●	◐	◐	●	●
Build Systems	○	○	○	○	◐	◐	◐	○	○	◐	○	○	◐	●
Components, Services	◐	◐	◐	●	◐	◐	●	◐	○	◐	○	◐	●	●
Frameworks, Plug-ins	◐	●	○	○	●	◐	●	●	◐	●	◐	○	●	●
Feature-Oriented Programming	●	○	○	○	○	○	●	●	○	●	●	○	●	●
Aspect-Oriented Programming, Cross-Cutting Concerns	●	◐	◐	○	◐	○	●	●	○	●	●	◐	●	●
Quality Assurance														
Feature-Model Analysis, Satisfiability Solving, Model Counting	○	○	○	○	○	○	◐	◐	●	◐	◐	○	◐	●
Feature-Mapping Analysis, Presence Conditions	○	○	○	○	○	○	●	●	○	●	○	○	○	●
Solution-Space Analysis	○	○	○	○	○	○	●	◐	○	●	○	○	○	●
Feature Interactions	○	○	○	○	○	○	●	◐	○	●	○	○	◐	●
Testing, Sampling	◐	◐	◐	◐	◐	◐	◐	◐	○	●	●	◐	●	●
Formal Methods, Theory, Algebra	●	○	○	○	○	○	◐	○	◐	○	○	○	○	○
Validation, Certification, Specification	◐	◐	○	○	◐	○	○	◐	○	◐	○	○	○	○
Management														
Process Models, Development Life Cycle	●	●	●	●	●	●	◐	◐	◐	◐	◐	●	◐	◐
Organization, Roles, Business Cases	○	●	●	○	●	●	○	○	○	○	○	●	○	○
Financing, Economics, Cost Estimation	○	◐	●	○	●	●	○	○	○	○	○	●	○	○
Human Factors, Cognition, Knowledge	○	○	◐	○	○	○	○	○	○	○	○	◐	○	○
Transfer														
Adoption Strategies	○	●	◐	●	●	●	●	○	●	●	○	◐	○	●
Reengineering, Reverse Engineering, Refactoring	◐	○	○	◐	○	◐	◐	◐	◐	◐	●	○	○	◐
Evolution, Maintenance	○	●	○	●	◐	●	◐	◐	○	◐	●	○	◐	●
Release, Deployment, Operation	○	◐	◐	◐	○	◐	○	○	○	○	○	◐	○	○
Case Studies, Industrial Applications	◐	●	●	●	●	●	◐	◐	◐	◐	●	●	◐	◐
Tool Support	○	○	◐	○	○	○	●	●	◐	●	●	○	●	●
Exercises, Instructions, Training	○	○	●	○	◐	○	●	●	●	●	●	◐	●	●
Miscellaneous														
Multi SPLs, Software Ecosystems	○	○	○	○	◐	○	◐	○	○	◐	○	○	○	○
Dynamic SPLs, Adaptive Systems	◐	○	○	○	○	○	◐	○	○	○	○	○	○	◐
Computational Complexity, Limits, Combinatorial Explosion	○	○	○	○	◐	◐	○	○	◐	◐	◐	○	○	●

○ Not or barely mentioned ◐ Discussed partially or superficially ● Discussed in breadth or depth

Remark on Included Books We aim to include books that are concerned with SPLs, well-known in research or industry, and suitable for teaching (e.g., aimed at students or creators of educational resources). To this end, we consider SPL books in our inventory and our literature databases. In addition, we search Google, Amazon, eBay, and Google Scholar for the phrases: `"software␣" ("product␣line" | "variability") "␣book"`. We only include English books with available full-text. We exclude conference proceedings, dissertations, edited anthologies or collections, and books with less than 100 citations (according to Google Scholar). A more detailed version is available in our online appendix.[7]

Remark on Included Courses We include all publicly available complete English courses on SPLs that we identify in Table 1.



Part I: Ad-Hoc Approaches for Variability

1. **Introduction**
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

1a. Introduction to Product Lines

Handcrafting and Customization
Mass Production
Mass Customization
Recap: The Software Life Cycle
Features and Products of a Domain
Software Product Line
Product-Line Engineering
Summary

1b. Challenges of Product Lines

Software Clones
Feature Traceability
Automated Generation
Combinatorial Explosion
Feature Interactions
Continuing Change and Growth
Summary

1c. Course Organization

What You Should Know
What You Will Learn
What You Might Need
Credit for the Slides
Summary
FAQ

Figure 2: The first handout slide of our course on software product lines. We show the overall structure of our course on the top (including all 12 lectures) and the structure of the individual lecture on the bottom (in this case, the introductory lecture).

learned lessons, selected opportunities for further reading ($\mathbf{G}_1$), and an optional exercise that students can discuss in groups ($\mathbf{G}_3$). This three-block structure has several advantages: First, it allows educators and students to easily navigate the lecture slides. Second, it encourages students to pay attention, so they are able to participate in the interaction. This interaction can then immediately reinforce the learned concepts ($\mathbf{G}_3$). Third, it implements the *sandwich principle* [33], which mandates alternating periods of active listening and audience participation [16]. Recent evidence suggests that this principle may improve students' critical thinking and self-learning ability as well as their satisfaction with the course [13, 17].

Inside each block, we typically follow an inductive structure ($\mathbf{G}_3$), just as with the course at large. For instance, in $\mathbf{L}_{1a}$ (cf. Figure 2), we begin with examples of customization and slowly build up towards the concept of an SPL, finally asking the students to name their own examples of SPLs as part of the interaction. In addition, we occasionally use recaps to recapitulate a previous lecture, and even memes (e.g., XKCD comics)[10] to keep students engaged ($\mathbf{G}_3$). Where applicable, we also refer to relevant research publications in order to substantiate our claims and animate students to make themselves familiar with the literature ($\mathbf{G}_1$). Finally, we conclude each lecture with a list of frequently asked questions (FAQs), which students can use to check their grasp on each block's topic.

4.2 Topics

In Table 2 (in the column *our course* on the far right), we give an overview of the concrete topics that we cover in our course. To facilitate a comparison with existing courses (cf. Table 1), we also extend the literature review in Table 2 with an overview of the topics covered in these five courses. For instance, we can use this overview for a deeper comparison of our course to the only other complete, open course created by Gay and Berger.[5] This comparison shows that our course covers both more topics (e.g., clone-and-own) and several topics in more depth or breadth (e.g., feature-model analysis). Our extended review also shows how each course (including our own) relates to the topics covered in influential books on SPLs (cf. Section 3). As our course is loosely based on Apel et al. [7] and Meinicke et al. [50], it mostly covers similar topics as these books. However, we sometimes deviate from both books, for instance by reordering, omitting, or adding topics. In the following, we discuss some notable changes we make compared to these books, focusing on practical challenges, running examples, added topics, and original contributions.

Challenges In the first lecture ($\mathbf{L}_{1a}$), we name some promises of SPLs (e.g., cost reduction). However, we also want to ensure that students are immediately confronted with some potential drawbacks of SPLs ($\mathbf{G}_1$). Thus, in $\mathbf{L}_{1b}$, we identify six practical challenges of SPLs that relate to the topics of our course ($\mathbf{G}_4$). These challenges include (cf. Figure 2): software clones ($\mathbf{L}_{2b,3,6a}$), feature traceability ($\mathbf{L}_{2b,5c,6,7}$), automated generation ($\mathbf{L}_{2,5,6c,7}$), combinatorial explosion ($\mathbf{L}_{2c,3a,4,10,11}$), feature interactions ($\mathbf{L}_{9,10a,11b}$), and continuing change and growth ($\mathbf{L}_{8,12}$). Besides raising awareness, having these challenges serves as a common thread throughout the course ($\mathbf{G}_3$).

Examples Analogously to these challenges, we use several running examples throughout the course ($\mathbf{G}_3$). Some of these examples include: the graph product line (GPL) [47, 69] ($\mathbf{L}_{2,3,5-10}$), the Linux kernel [65] ($\mathbf{L}_{1,5,8,10-12}$), and the PigNap case study [43] ($\mathbf{L}_5$). Besides these examples from research and industry, we also contribute our own examples for configuring databases ($\mathbf{L}_{4,10,11}$), ordering waffles ($\mathbf{L}_4$), and assembling Lego minifigures ($\mathbf{L}_{1,10}$). Having such recurring examples, students can easily recognize and use them, for example to compare implementation techniques ($\mathbf{G}_3$).

Additions To account for recent research insights ($\mathbf{G}_1$), we include several practical topics in our course ($\mathbf{G}_4$), none of which is covered by any book in Table 2. Some of these topics include: the

[10] https://xkcd.com/

universal variability language (UVL) [27, 62] ($\mathbf{L}_{4b}$), model counting and enumeration [29, 63] ($\mathbf{L}_{4c}$), the KConfig language and tooling [22, 25, 53] ($\mathbf{L}_{5a}$), microservices [9, 55] ($\mathbf{L}_{6b}$), the promote-pl development process [41] ($\mathbf{L}_{8c}$), combinatorial interaction testing [4, 48] ($\mathbf{L}_{11b}$), and solution-space sampling [39, 68] ($\mathbf{L}_{11c}$).

Contributions Furthermore, we even make some original contributions in the preparation of this course. For example, we compute recent statistics regarding the evolution of the Linux kernel ($\mathbf{L}_{1b,8a}$), we distinguish build systems for clone-and-own and conditional compilation ($\mathbf{L}_{3c,5a}$), and we critically reflect on the complexity of SPLs and feature models ($\mathbf{L}_{4c,10c}$) [42]. These contributions demonstrate how education can positively interact with research ($\mathbf{G}_1$).

5 Adoption Challenges

In the following, we discuss several challenges related to the practical adoption of our course and how we address them.

Version Control We publish our course and its entire history [18] in a public Git repository[7] with > 600 commits by 7 contributors. Besides easing backup and distribution, this ensures transparency and, thus, accountability for any contributions made by us and others ($\mathbf{G}_2$). Potential concerns can be raised in our issue tracker.

No Clone-and-Own We proactively create a new course on SPLs to avoid typical issues of clone-and-own (cf. Section 1). To prevent such issues in the future as well, we discourage diverging forks by inviting other educators to integrate their contributions into the main Git repository, for example with pull requests ($\mathbf{G}_2$). This is possible because we create all our slides with LaTeX, which allows for textual diffs that are easy to review and merge (e.g., compared to PowerPoint or similar visually oriented tools).

Customization By relying on LaTeX, we can easily adapt our slides to new universities ($\mathbf{G}_2$) and satisfy needs for further customization (e.g., handout and dark-mode slides). To this end, we currently use an annotative technique (i.e., the `\ifuniversity{}` command, a LaTeX equivalent of C's `#ifdef` [44]). However, we almost exclusively use this technique for customizing appearance in order to avoid unneeded variability [1]. In particular, we deliberately avoid creating an "SPL of SPL courses" [3], as this would introduce a significant amount of additional complexity, which might be a source for inconsistencies and obstacle for future extensions. Using this technique, we adapted our course to six universities (cf. Table 3).

Underrepresented Topics With our course, we attempt to flesh out the SPL baseline curriculum proposed by Acher et al. [2]. However, we are aware that some particular SPL topics (cf. Table 2) are currently underrepresented in our curriculum. Management topics, in particular, are covered much more extensively by Donohoe and Northrop.[4] We envision two ways to close such gaps: First, educators with the respective expertise may contribute additional guest lectures on these topics to our course. Second, we encourage and look forward to other educators creating completely different courses on SPLs, in which these topics may be covered in more depth. To this end, the results of our literature review (cf. Table 2) can be used to identify gaps for future books and courses on SPLs.

Exercise Sheets We currently accompany the lecture with an exercise class, in which students solve and discuss in-depth exercises on each lecture's topic. Because our primary focus is on the lecture slides, we currently use work-in-progress exercise sheets based on previous courses. In the near future, we aim to revise these exercise sheets to match well with the lecture slides, so we can release them.

6 Preliminary Evaluation

Creating new lecture slides from scratch is not an easy task, but getting them actually adopted in practice is yet another challenge. In order to investigate whether our course is worth adopting, we aim to evaluate how it is received by our students with regard to several quality criteria. Thus, we seek to complement the discussion of our intentions (cf. Section 2) and their implementation (cf. Section 3 and 4) with real feedback from students. In particular, we aim to address the following research questions:

$\mathbf{RQ_1}$ How well do students receive our course in general?
$\mathbf{RQ_2}$ How well do students receive our course compared to …
$\mathbf{RQ_{2.1}}$ … other courses at the same faculty?
$\mathbf{RQ_{2.2}}$ … a previous course on SPLs at the same faculty?

Performing comprehensive, meaningful evaluations of university courses is known to be methodologically challenging [28, 37, 49]. Nonetheless, we collect feedback from several *student evaluations of teaching (SETs)* [32], which are conducted at many universities. We believe this feedback is a valuable first step towards assessing the quality of our course and whether it is suitable for adoption.

6.1 $\mathbf{RQ_1}$: General Reception

First, we aim to determine how students receive our course in general (e.g., aspects they like or dislike, and room for improvement).

Methodology Over the last two years, we have conducted six SETs at five universities, in which we collected anonymized feedback from 64 students. The feedback is both quantitative and qualitative, summarized in Table 3 and 4, respectively. For quantitative feedback, all universities use some kind of Likert scale to indicate (dis-)agreement. Still, we need to unify the questionnaires used by different universities. Thus, we aggregate relevant questions into a set of recurring quality criteria, and we normalize all scores to the same scale (details indicated in Table 3). We leave gaps in Table 3 when a SET does not cover a given quality criterion at all.

Results The results we show for RQ_1 in Table 3 indicate scores from 78.3 to 97.0 points for course-related quality criteria. In the mean (computed over all universities), students rate the structure of our course with 88.5, its quality with 88.9, and its difficulty and pacing with 91.7 out of 100 points. As for the self-assessment of our students, they rate their own motivation for the course with 69.3 points, their gain in knowledge with 83.7, and their overall satisfaction with 85.4 points in the mean. To put these numbers into context, we also show aggregated qualitative student feedback in Table 4, which reveals several points of praise and criticism.

Discussion The feedback suggests that our course was well-received by the majority of students. In particular, students seem to appreciate the structure of our course (i.e., as outlined in section 3 and 4), its material (i.e., the lecture slides and exercise sheets), and its difficulty and pacing (i.e., it is not too hard/fast and not too easy/slow). However, it is of course not possible to satisfy everyone—for instance, there will always be students who are challenged too

Table 3: Student feedback from Likert-scale questions (RQ_1, $RQ_{2.1}$, $RQ_{2.2}$).

	University of Bern				Ulm University			Harz	Magdeburg		Paderborn	Aggregated	
	WT 2022/23		WT 2023/24		ST 2023			ST 2023	WT 2023/24		ST 2024	Total or Mean	
Quality Criteria	RQ_1	$RQ_{2.1}$	RQ_1	$RQ_{2.1}$	RQ_1	$RQ_{2.1}$	$RQ_{2.2}$	RQ_1	RQ_1	$RQ_{2.1}$	RQ_1	RQ_1	$RQ_{2.1}$
# Participants	8	4110	8	2608	13	856	9	8	11	471	16	64	8045
Course													
Structure	90	76 +19% .00∗	90	80 +13% .03∗	95	81 +17% .00∗	91 +4% .37		88	80 +10% .24	80	89	79 +15%
Material	88	73 +20% .03∗	90	80 +13% .10	93	80 +17% .00∗	85 +9% .25				85	89	77 +17%
Difficulty, Pacing	97	82 +18% .11	97	91 +6% .01∗	91	78 +17% .01∗	87 +5% .46		78	75 +4% .67	95	92	82 +11%
Self-Assessment													
Motivation	75	68 +11% .43	79	70 +12% .08	86	77 +12% .05	86 +0% .99	44	63	73 −14% .19		69	72 +5%
Gain in Knowledge	86	74 +16% .05∗	89	77 +16% .02∗	91	74 +23% .00∗	90 +2% .80		73	70 +5% .65	79	84	74 +15%
Satisfaction	94	78 +21% .00∗	94	82 +14% .02∗	96	80 +20% .00∗	90 +7% .27	75	73	70 +5% .60	80	85	77 +15%

Harz University of Applied Studies (Wernigerode) University of **Magdeburg** **Paderborn** University TU Braunschweig (not yet evaluated)

WT Winter term **ST** Summer term **Score Scale** 0 = worst, 100 = best $+x\%$ $-x\%$ Comparison relative to RQ_1 p ∗ Statistically significant ($p < 0.05$)

Quantitative feedback aggregated from six questionnaires across five universities with 64 student participants in total. We exclude one unevaluated and three ongoing iterations of our course. We identify six key categories of quality criteria that most questionnaires cover. We calculate each score as the mean of the constituent questions, normalized to a scale from 0 to 100 for easy comparison. We omit questions that only relate to the exercises and the performance of lecturers. A more detailed version is available in our online appendix.[7]

Table 4: Student feedback from open questions (RQ_1).

Praise
Detailed course structure with a clear overview and a common thread Many images with good visualizations, appropriate information depth Many practical, recurring examples Interactions, FAQs, opportunities for asking questions and discussions Video recordings, dark-mode slides, guest lectures
Criticism
Having English slides was challenging for some non-native speakers Too theoretical for some students, too practical and coding-focused for others Some lectures are too long (e.g., feature modeling) and too fast towards the end

Qualitative feedback aggregated from the same responses that we consider in Table 3.

much or too little by the material. Notably, the student's own motivation seems comparably low in the mean, which is due to the low rating at the Harz University of Applied Studies. This is a vocational university with less emphasis on research, which might explain the lower self-assessment at this university. Another point of criticism we aim to address is that, despite the good overall pacing, some individual lectures have too much content and could be tightened.

6.2 $RQ_{2.1}$: Comparison to Faculty

While RQ_1 indicates generally positive feedback, the results for our course become more meaningful when we compare them to a suitable baseline. For $RQ_{2.1}$, we aim to compare our course to other courses that have been held at the same university.

Methodology Four of the six SETs we conducted (i.e., in Bern, Ulm, and Magdeburg) also include data on many other courses that have been evaluated at the same faculty. By aggregating and normalizing the other courses' scores analogously to RQ_1, we can compare them to our course and assess where it performs better or worse. We show these results in Table 3 in the column for $RQ_{2.1}$, including the differences to our course's results as percentages. To calculate these percentages, we only consider the universities for which data on other courses is available. This way, the all-university percentage in the last column of Table 3 is computed correctly, although the absolute mean values for RQ_1 and $RQ_{2.1}$ are not directly comparable. To test the differences' statistical significance, we perform a two-tailed Welch's t-test with a significance level of $\alpha = 0.05$. This test is more robust than Student's t-test for unequal variances and unequal sample sizes, which apply here. We indicate p-values and significant results ($*$) in Table 3 and omit variances for brevity.

Results The results for $RQ_{2.1}$ show that, in the mean, students rate our course as better than other courses at the same faculty in almost all quality criteria. In particular, our course scores 11% to 17% more points than other courses in course-related quality criteria in the mean. Regarding self-assessment, we find that, in the mean, students award our course 5% to 15% more points than other courses. Only at the University of Magdeburg do we find a mean reduction in motivation by 14% compared to other courses. Regarding statistical significance, we find that for almost all quality criteria, the differences are significant for at least two distinct universities. Again, the students' motivation is the single exception, having no statistically significant differences ($0.05 \le p \le 0.43$).

Discussion These results suggest that students generally regard our course as better than a typical course at the same faculty. While we consider this a very positive result, we must interpret it carefully. In particular, not all differences are statistically significant, and for these we cannot completely rule out random effects. However, for most quality criteria, we find significant differences for at least two distinct universities, which is promising. The single exception is student motivation, which does not seem to significantly differ from other courses at the same faculty.

6.3 $RQ_{2.2}$: Comparison to Previous Course

Finally, as an alternative baseline, we aim to compare our course to a previous course on SPLs that has been held at the same faculty.

Methodology At two of the five universities considered in Table 3 (i.e., Ulm and Magdeburg), another course on SPLs (C_{old}) used to be taught before our course (C_{new}) was introduced. The course C_{old} had completely different, unpublished slides, which were adapted and slightly updated from year to year with clone-and-own. To reduce the influence of confounding factors, we aim to only compare an SET of C_{new} to an SET of C_{old} if both courses were held at the same university, by the same lecturer, and in a similar timeframe (i.e., within two years). Unfortunately, these criteria exclude most SETs we have for C_{old}, as they are simply too old to allow for meaningful comparison. Thus, we can only compare C_{new} (held in 2023) to one iteration of C_{old} (held in 2022) at Ulm University. We show the results for this comparison in Table 3 in the column for $RQ_{2.2}$. We test for statistical significance analogous to $RQ_{2.1}$.

Results The results for $RQ_{2.2}$ show that, in the mean, students at Ulm University rate C_{new} slightly better than C_{old}. In particular, C_{new} scores 4% to 9% more points than C_{old} in course-related quality criteria in the mean. Regarding self-assessment, these differences range from 0% to 7% more points for C_{new}. We find no statistically significant differences between C_{old} and C_{new} for any quality criterion ($0.25 \leq p \leq 0.99$).

Discussion Unfortunately, the results for $RQ_{2.2}$ remain inconclusive due to a lack of statistical significance, which maybe could have been reached with more regular SETs before introducing our course. Nonetheless, they do not contradict the favorable results we found for RQ_1 and $RQ_{2.1}$, which is reassuring.

6.4 Threats to Validity

Evaluations of teaching material can be challenging due to a variety of biases and confounding factors [28, 37, 49]. Thus, we aim to openly discuss potential threats to the validity of our conclusions.

Internal Validity Several biases and confounding factors might distort the results of our SETs compared to reality, some of which we discuss in the following. First, only students with very positive or negative experiences may choose to fill out questionnaires. To avoid this issue, we compare our course to other courses at the same faculty ($RQ_{2.1}$), which would suffer from the same bias. Second, students might simply like the topic and would also rate any other SPL course better than faculty average. We partially mitigate this by comparing our course to a previous course on SPLs ($RQ_{2.2}$). Third, students might be impacted in their answers by the lecturer's performance (e.g., are they experienced/enthusiastic?), exercise class (e.g., is it too labor-intensive/strict?) and other external factors not related to the course (e.g., organizational structures). We reduce this threat by collecting SETs over several distinct lecturers at multiple universities. To improve comparability, we unify the questionnaires from different universities and aggregate their questions into relevant quality criteria. Also, we ignore questions that only relate to teaching personnel and exercise classes. This way, we can focus on the actual course material and the students' self-assessment.

External Validity The external validity of our conclusions might be limited by the size of our dataset (e.g., regarding the number of universities, SETs, and participants). While more evidence is always welcome, our dataset already demonstrates how our course can be successfully introduced to new universities, which is the main goal of our evaluation. For the comparison to other courses at the same faculty ($RQ_{2.1}$), we only have data for four universities. However, at these universities, a total of 8045 students participated in SETs, which makes this a suitable baseline for comparison nonetheless.

7 Related Work

To the best of our knowledge, we are the first to create a new course on SPLs from scratch, describe our approach in detail, and publish it as open educational resources. We are also not aware of another attempt at evaluating a course on SPLs by collecting both quantitative and qualitative feedback from SETs. However, there are several case studies and experience reports on teaching SPLs [18, 21, 46, 52, 58, 61] published either at CSEE&T [14] or the SPLTEA workshop [3]. Some of these publications only discuss course goals superficially [21, 46, 52, 58], do not discuss a complete course [21, 52, 58], or target other audiences than computer science students [52, 61]. Other publications have a different focus from ours, for example on course evolution [18, 21] or specific pedagogical approaches [18, 61]. In particular, none of these publications review the literature on SPLs or perform a quantitative evaluation, as we do.

Acher et al. [2] performed two surveys and a workshop to capture a snapshot of current practices and challenges in teaching SPLs. In contrast to our work, they do not focus on a specific course. Our work can be regarded as a continuation of their work, which identified the need for a baseline curriculum, such as ours.

8 Conclusion

In this paper, we shared our experiences in architecting and designing a new university course on SPLs from scratch. We publish our course in form of open educational resources. Thus, we hope to ignite a collaborative effort to create and continuously improve SPL teaching material. Ideally, this will contribute to SPL education growing more mature along with research and industry.

In the future, we aim to add new lectures on underrepresented topics, release exercise sheets, and introduce an open call for scientific talks to attract guest lecturers and, thus, potentially extend the network of universities participating in this project. We also intend to regularly reevaluate our course and consider new feedback.

Acknowledgments

This project would not have been possible without the work and support of several of our colleagues: First, we want to thank Sebastian Krieter, Paul Maximilian Bittner, Chico Sundermann, Benno Hölz, and Florian Sihler for their contributions to the lecture slides and slide template. Second, we want to thank Sandra Greiner, Chico, Paul, and Sebastian for their contributions to the exercise sheets, which we hope to publicly release in the future. Third, we want to thank Gunter Saake, Thomas Leich, Sandy Grawe, and Roman Macháček for their organizational support throughout this project. Fourth, we want to thank our students for participating in the course and giving valuable feedback. Finally, special thanks go out to Sandra, who was very involved in discussing and developing new and improved exercises for our course.

Student Testimonials “I have not heard the term SPL before the course at all, and I think I have acquired a pretty good understanding of it now.” — “The slides were of good quality, the lecture always interactive and never boring.” — “It's really a pity that not more students have attended this high quality lecture, but I would recommend it to any other computer science student.”

References

[1] Mathieu Acher, Luc Lesoil, Georges Aaron Randrianaina, Xhevahire Tërnava, and Olivier Zendra. 2023. A Call for Removing Variability. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 82–84.

[2] Mathieu Acher, Roberto E. Lopez-Herrejon, and Rick Rabiser. 2017. Teaching Software Product Lines: A Snapshot of Current Practices and Challenges. *ACM Trans. on Computing Education (TOCE)* 18, 1, Article 2 (2017), 2:1–2:31 pages.

[3] Mathieu Acher, Rick Rabiser, and Roberto E. Lopez-Herrejon (Eds.). 2019. *Fourth International Workshop on Software Product Line Teaching (SPLTea 2019)*. ACM.

[4] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 144–155.

[5] Vander Alves, Nan Niu, Carina Alves, and George Valença. 2010. Requirements Engineering for Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 52, 8 (2010), 806–820.

[6] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering With a Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 532–535.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[8] Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Miguel R. Luaces, Christa Schwanninger, Shaukat Ali, Mahsa Varshosaz, Angelo Gargantini, Stefania Gnesi, Malte Lochau, Laura Semini, and Hironori Washizaki (Eds.). 2023. *SPLC '23: Proceedings of the 27th ACM International Systems and Software Product Line Conference*. ACM.

[9] Wesley K. G. Assunção, Jacob Krüger, and Willian D. F. Mendonça. 2020. Variability Management Meets Microservices: Six Challenges of Re-Engineering Microservice-Based Webshops. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 22.

[10] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.

[11] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Software Engineering (TSE)* 39, 12 (2013), 1611–1640.

[12] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in the Real: A Perspective From the Operating Systems Domain. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 73–82.

[13] Anna Bock, Bianca Idzko-Siekermann, Martin Lemos, Kristian Kniha, Stephan Christian Möhlhenrich, Florian Peters, Frank Hölzle, and Ali Modabber. 2020. The Sandwich Principle: Assessing the Didactic Effect in Lectures on "Cleft Lips and Palates". *BMC medical education* 20 (2020), 1–7.

[14] Andreas Bollin, Ivana Bosnić, Jennifer Brings, Marian Daun, and Meenakshi Manjunath (Eds.). 2024. *36th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE. https://doi.org/10.1109/CSEET62301.2024.10663010

[15] Jan Bosch. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Pearson Education.

[16] Diane M. Bunce, Elizabeth A. Flens, and Kelly Y. Neiles. 2010. How Long Can Students Pay Attention in Class? A Study of Student Attention Decline Using Clickers. *Journal of Chemical Education* 87, 12 (2010), 1438–1443. https://doi.org/10.1021/ed100409p

[17] Xiaoyan Cai, Mingmei Peng, Jieying Qin, Kebing Zhou, Zhiying Li, Shuai Yang, and Fengxia Yan. 2022. Sandwich Teaching Improved Students' Critical Thinking, Self-Learning Ability, And Course Experience in the Community Nursing Course: A Quasi-Experimental Study. *Frontiers in Psychology* 13 (2022), 11 pages. https://doi.org/10.3389/fpsyg.2022.957652

[18] Jaime Chavarriaga, Rubby Casallas, Carlos Parra, Martha Cecilia Henao-Mejía, and Carlos Ricardo Calle-Archila. 2019. Nine Years of Courses on Software Product Lines at Universidad de los Andes, Colombia. In *Proc. Int'l Workshop on Software Product Line Teaching (SPLTea)*. ACM, 130–133.

[19] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *J. Information and Software Technology (IST)* 53, 4 (2011), 344–362.

[20] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[21] Philippe Collet, Sébastien Mosser, Simon Urli, Mireille Blay-Fornarino, and Philippe Lahire. 2014. Experiences in Teaching Variability Modeling and Model-Driven Generative Techniques. In *Proc. Int'l Workshop on Software Product Line Teaching (SPLTea)*. ACM, 26—-29.

[22] The Kernel Development Community. 2018. KConfig Language. Website: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html. Accessed: 2024-01-30.

[23] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley.

[24] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[25] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 45–54.

[26] Emelie Engström and Per Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *J. Information and Software Technology (IST)* 53 (2011), 2–13. Issue 1.

[27] Kevin Feichtinger, Chico Sundermann, Thomas Thüm, and Rick Rabiser. 2022. It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 67–78.

[28] Daniela Feistauer and Tobias Richter. 2017. How Reliable Are Students' Evaluations of Teaching Quality? A Variance Components Approach. *Assessment & Evaluation in Higher Education* 42, 8 (2017), 1263–1279.

[29] José A Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. 2016. Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 74–78.

[30] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2013. Variability in Software Systems-a Systematic Literature Review. *IEEE Trans. on Software Engineering (TSE)* 40, 3 (2013), 282–306.

[31] Hassan Gomaa. 2004. *Designing Software Product Lines With UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley.

[32] Robert L Isaacson, Wilbert J McKeachie, John E Milholland, Yi G Lin, Margaret Hofeller, and Karl L Zinn. 1964. Dimensions of Student Evaluations of Teaching. *Journal of Educational Psychology* 55, 6 (1964), 344.

[33] Martina Kadmon, Veronika Strittmatter-Haubold, Rainer Greifeneder, Fadja Ehlail, and Maria Lammerding-Köppel. 2008. The Sandwich Principle – Introduction to Learner-centred Teaching/Learning Methods in Medicine. *Zeitschrift für Evidenz, Fortbildung und Qualität im Gesundheitswesen* 102, 10 (2008), 628–633. https://doi.org/10.1016/j.zefq.2008.11.018 Professionalisierung der medizinischen Ausbildung.

[34] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[35] Timo Kehrer, Marianne Huchard, Leopoldo Teixeira, and Christian Birchler (Eds.). 2024. *VaMoS '24: Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM.

[36] Peter Knauber, Jesús Bermejo Muñoz, Günter Böckle, Julio Cesar Sampaio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, and David M. Weiss. 2001. Quantifying Product Line Benefits. In *Proc. Int'l Workshop on Software Product-Family Engineering (PFE)*. Springer, 155–163.

[37] Rebecca J. Kreitzer and Jennie Sweet-Cushman. 2021. Evaluating Student Evaluations of Teaching: A Review of Measurement and Equity Bias in Sets and Recommendations for Ethical Reform. *Journal of Academic Ethics* 20, 1 (2021), 73–84. https://doi.org/10.1007/s10805-021-09400-w

[38] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 42–45.

[39] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Sebastian Ruland, Malte Lochau, Gunter Saake, and Thomas Leich. 2022. *T-Wise Presence Condition Coverage and Sampling for Configurable Systems*. Technical Report arXiv:2205.15180. Cornell University Library.

[40] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 432–444.

[41] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. 2020. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–12.

[42] Elias Kuiter, Tobias Heß, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Gunter Saake. 2024. How Easy Is SAT-Based Analysis of a Feature Model?. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 149–151.

[43] Elias Kuiter, Jacob Krüger, and Gunter Saake. 2021. Iterative Development and Changing Requirements: Drivers of Variability in an Industrial System for Veterinary Anesthesia. In *Proc. Int'l Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)*. ACM, 113–122.

[44] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Proc. Int'l Symposium on Visual*

Languages and Human-Centric Computing (VL/HCC). IEEE, 143–150.

[45] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 105–114.

[46] Liana Barachisio Lisboa, Leandro Marques Nascimento, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2008. A Case Study in Software Product Lines: An Educational Experience. In *Proc. IEEE Conf. on Software Engineering Education and Training (CSEE&T)*. IEEE, 155–162.

[47] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Conf. on Generative and Component-Based Software Engineering (GCSE)*. Springer, 10–24.

[48] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proc. Int'l Workshop on Combinatorial Testing (IWCT)*. IEEE, 1–10.

[49] Mark Davies Mark Shevlin, Philip Banyard and Mark Griffiths. 2000. The Validity of Student Evaluation of Teaching in Higher Education: Love Me, Love My Lectures? *Assessment & Evaluation in Higher Education* 25, 4 (2000), 397–405. https://doi.org/10.1080/713611436

[50] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability With FeatureIDE*. Springer.

[51] Tomohiro Nagashima and Susan Harch. 2021. Motivating Factors Among University Faculty for Adopting Open Educational Resources: Incentives Matter. *Journal of Interactive Media in Education* 2021, 1 (2021), 10 pages.

[52] Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda. 2018. Teaching Software Product Lines as a Paradigm to Engineers: An Experience Report in Education Programs and Seminars for Senior Engineers in Japan. In *Proc. Int'l Workshop on Software Product Line Teaching (SPLTea)*. ACM, 46–47.

[53] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 893–905.

[54] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. 2015. A Systematic Literature Review of Software Product Line Management Tools. In *Proc. Int'l Conf. on Software Reuse (ICSR)*. Springer, 73–89.

[55] Marcus Pinnecke. 2021. Product-Lining the Elinvar Wealthtech Microservice Platform. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 60–68.

[56] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[57] Michael J. Prince and Richard M. Felder. 2006. Inductive Teaching and Learning Methods: Definitions, Comparisons, And Research Bases. *Journal of Engineering Education* 95, 2 (2006), 123–138. https://doi.org/10.1002/j.2168-9830.2006.tb00884.x

[58] Clément Quinton. 2018. Giving Students a Glimpse of the SPL Lifecycle in Six Hours: Challenge Accepted!. In *Proc. Int'l Workshop on Software Product Line Teaching (SPLTea)*. ACM, 42–43.

[59] Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. 2018. A Study and Comparison of Industrial vs. Academic Software Product Line Research Published at SPLC. (2018), 14–24.

[60] Zaynab Sabagh and Alenoush Saroyan. 2014. Professors' Perceived Barriers and Incentives for Teaching Improvement. *International Education Research* 2, 3 (2014), 18–40.

[61] Christoph Seidl and Irena Domachowska. 2014. Teaching Variability Engineering to Cognitive Psychologists. In *Proc. Int'l Workshop on Software Product Line Teaching (SPLTea)*. ACM, 16—-23.

[62] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet Another Textual Variability Language? A Community Effort Towards a Unified Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 136–147.

[63] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering (EMSE)* 28, 29 (2023), 38.

[64] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A Taxonomy of Variability Realization Techniques: Research Articles. *Software: Practice and Experience* 35, 8 (2005), 705–754.

[65] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*. ACM, 47–60.

[66] Scientific United Nations Educational and Cultural Organization (UNESCO). 2019. Recommendation on Open Educational Resources (OER). https://www.unesco.org/en/legal-affairs/recommendation-open-educational-resources-oer. Accessed: 2024-03-18.

[67] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

[68] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–13.

[69] Hillel Wayne. 2024. The Hunt for the Missing Data Type. https://www.hillelwayne.com/post/graph-types. Accessed: 2024-03-22.

When to Sample from Feature Diagrams?

Nikolai Käfer
TU Dresden
Dresden, Germany
nikolai.kaefer@tu-dresden.de

Sven Apel
Saarland University
Saarbrücken, Germany
apel@cs.uni-saarland.de

Christel Baier
TU Dresden
Dresden, Germany
baier@tcs.inf.tu-dresden.de

Clemens Dubslaff
Eindhoven University of Technology
Eindhoven, Netherlands
c.dubslaff@tue.nl

Holger Hermanns
Saarland University
Saarbrücken, Germany
hermanns@cs.uni-saarland.de

Abstract

Uniform random sampling (URS) has many applications in configurable systems analysis. Usually, feature models consisting of a hierarchical feature diagram and additional side constraints specify the space of valid configurations to be sampled from. However, URS has predominately been applied on feature models translated *a priori* into conjunctive normal form (CNF). In this work, we study URS approaches that instead operate directly on feature diagrams and provide a comparative evaluation of their performance against well-established URS tools for CNF representations. Our findings suggest that translating feature models to CNF offers advantages, even in the presence of only few side constraints.

CCS Concepts

• **Software and its engineering → Software product lines**.

Keywords

Uniform Random Sampling, Feature Models, BDDs

ACM Reference Format:
Nikolai Käfer, Sven Apel, Christel Baier, Clemens Dubslaff, and Holger Hermanns. 2025. When to Sample from Feature Diagrams?. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3715340.3715442

Acknowledgments

This work was partially supported by the NWO through Veni grant VI.Veni.222.431 and by the DFG under the projects EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy) and TRR 248 (see https://perspicuous-computing.science, project ID 389792660).

1 Introduction

Generating a representative set of instances of a complex system domain is a major challenge across computer science. For instance, in the area of data-driven AI, the input space is exponential in the number of features. This renders machine-learning algorithms only feasible when applied to a small subset of the entire input-output relation. The accuracy of ML classifiers then crucially depends on how representative that subset is with respect to the entire input-output relation [8]. Similarly, in software product line engineering, the space of software configurations is exponential, rendering test case generation by means of system configurations a challenging task. Further, the distribution of bugs or other properties of interest is a priori not known and is likely to depend on cross-cutting feature interactions [5, 10]. *Uniform random sampling* (URS) is a key concept to derive representative sample sets with limited information [1, 29] by ensuring that every element of the sample space has the same probability to be sampled. The latter avoids bias in the samples and may allow for establishing at least a certain degree of statistical guarantees.

In this paper we focus on the challenge of URS on configuration spaces of variability-aware software. Configuration spaces are usually given through *feature models* over boolean configuration options, so-called *features* [4, 13]. Most commonly, a feature model comprises a *feature diagram* and additional *side constraints* [4, 23]. A feature diagram is a directed tree over features with certain boolean operators as inner nodes, while side constraints are given as propositional logic formulas or *cross-tree-constraints* annotating the feature diagram. The standard semantics of a feature diagram then induces a propositional logic formula [32]. So, the main benefit of a feature diagram lies in exposing the hierarchical dependencies of features that commonly are reflected in separation of concerns and structured software development processes. While feature diagrams hence carry much structural domain knowledge, the standard approach to URS over feature models starts off with translating feature diagram and side constraints into a single propositional logic formula in *conjunctive normal form* (CNF), followed by subsequent application of context-agnostic but highly optimized URS approaches for CNFs [1, 19, 29].

URS Approaches and Tools. Due to its broad applications, a variety of tools for URS have been proposed in the literature, covering a range of distinct methods. For instance, *knowledge compilation* approaches translate the input CNF into representations that support efficient sampling [14]. Although these methods involve an initial compilation step that can be time- and resource-intensive, the subsequent sampling tends to be fast. Examples include KUS [33], which leverages d-DNNF (deterministic decomposable negation normal

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715442

form) compilation [14], and BDDSampler [19], which utilizes *reduced ordered binary decision diagrams* (BDDs) [9] as a target for compilation and URS [25, 42]. Hashing-based methods, such as the UniGen family [11, 12, 35], combine random hash functions with satisfiability (SAT) solvers to produce near-uniform samples. Model counting (#SAT) asks for the exact count of satisfying assignments of a CNF, a computational problem that is closely related to URS [22]. For example, SPUR [3] adapts the DPLL procedure of many #SAT solvers [7] to produce uniform samples during model counting, while Smarch [30] constructs a bijection between solutions and integers within the model count range to achieve perfect uniform sampling. Heradio et al. [19] recently conducted a comparative evaluation of various URS tools applied on feature models given in CNF, but without taking into account the resource-intensive knowledge compilation steps towards generating BDDs or d-DNNFs. As can be expected, this perspective makes knowledge compilation approaches clearly outperform #SAT-based approaches, but leaves open the research question whether and for which sample sizes the resource investments for knowledge compilation pay off:

(RQ1) Which URS approaches and tools perform best on feature models for various sample sizes, also taking knowledge compilation resources into account?

URS on Feature Diagrams. As illustrated above, commonly used URS tools are not specifically tailored to the feature model context. Therefore, to sample from feature models, one must extract constraints from the feature diagram, incorporate any side constraints, and convert the resulting propositional formula to CNF. However, this hinders the exploitation of structural domain knowledge on the configuration space given in the feature diagram. Notably, both model counting [40] and URS [39] can be performed in linear time on feature diagrams (i.e., feature models without side constraints). In the following, we denote by *URS-FD* this URS approach on feature diagrams [39]. While this suggests that *URS-FD* clearly outperforms classical URS via CNFs, which is computational intractable in the worst case [22], it has, to the best of our knowledge, never been implemented and evaluated empirically on real-world feature diagrams. Due to the manifold heuristics and optimizations in state-of-the-art URS tools, this directly raises the question whether the linear-time URS-FD approach is actually as performant as it could be expected from its superior worst-case time complexity.

(RQ2) How competitive is URS-FD compared to URS tools on feature models without side constraints?

Rejection Sampling. This standard method [41] generates samples of a random variable over one dimension by uniformly sampling at random over a product sample space and applying a selector criterion to filter the second dimension. Samples that do not satisfy the criterion are discarded, typically followed by resampling. Rejection sampling is especially useful when incorporating the selector criterion directly into the sampling process is difficult.

URS-FD does a priori not support feature models with side constraints. However, a straight-forward extension of the algorithm can be established through rejection sampling, where the feature diagram serves as the first and the side constraints as second sample dimension. After sampling from the feature diagram, we check if the sample satisfies the side constraints of the feature model. If it does, we keep the sample; if not, we discard it and sample again. This approach, which we abbreviate *URS-FD*$^+$, guarantees uniform samples for the entire feature model, since the *sample space*—containing all configurations valid for the diagram—is a superset of the *solution space* which contains all configurations valid for both the diagram and the side constraints.

To the best of our knowledge, rejection sampling has not yet been applied in the area of URS for feature models. This method adds a further dimension to the URS portfolio for feature models and opens the question whether certain characteristics of feature models render specific URS methods to be superior.

(RQ3) What factors determine which sampling methods perform best for a given feature model?

1.1 Contribution

This paper gives answers to all three research questions stated above, providing the following three main contributions:

- We empirically compare state-of-the-art URS tools on feature models with non-trivial feature diagrams.
- We implement the URS-FD algorithm [39] as well as the new URS-FD$^+$ algorithm and compare them empirically.
- We establish phase transition points on the complexity of the feature models where knowledge-compilation and #SAT-based approaches start to outperform URS-FD$^+$.

We implemented URS-FD and URS-FD$^+$ in a tool called Zampler,[1] together with BDD-based knowledge compilation on side constraints to optimize the rejection sampling process within URS-FD$^+$. Our empirical evaluation is performed on commonly used benchmarks that comprise non-trivial feature diagrams [36], excluding feature diagrams where all information resides in side constraints (inducing a trivial diagram consisting of a single AND-node with all features as its children).

Our experiments towards answering RQ1 reveal that, when including the resources for knowledge compilation, the URS tool SPUR is the fastest sampler across all feature models in CNF, whereas thus far KUS and BDDSampler have been considered to be the fastest according to the literature [19]. In response to RQ2, we report that URS-FD indeed outperforms all state-of-the-art URS tools when not taking side constraint of the feature model into account. The efficiency of our tool Zampler—and even its feasibility—is directly related to the size ratio of sampling space vs. solution space, which we reflect by introducing the notion of *hit rate*. This echoes how strongly the side constraints restrict the solution space. Addressing RQ3, we find that for most models with many side constraints, hit rates are too low for rejection sampling to be feasible, and we identify the tipping points where feasibility breaks down.

1.2 Related Work

Benchmarking activities for URS on feature models include BURST [1, 2], a platform integrating many solvers and uniformity checks, or specific comparisons, such as [43], comparing SPUR and UniGen3. Heß et al. investigate URS as a means to achieve high t-wise coverage [20]. They consider QuickSampler, Smarch, and SPUR on

[1] The tool and evaluation results are publicly available at https://doi.org/10.5281/zenodo.14536082

a set of 49 feature models, as well as the t-wise samplers Baital and YASA. The introduction of BDDSampler [19] comes with a comparison across existing samplers KUS, Smarch, SPUR, and UniGen2 (and QuickSampler) on a set of 9 feature models. The measurements reported, however, do not include model construction times for BDDSampler and KUS, thus putting the findings partially into question. The remaining results see SPUR orders-of-magnitude faster than Smarch and UniGen2, the latter two timing out for most models after 1 hour. Earlier work [18, 31] only looked at Smarch, SPUR, and UniGen2 (and QuickSampler). To the best of our knowledge, we are the first to investigate and benchmark the performance of URS with a focus on feature models equipped with non-trivial diagrams and feature models not provided in CNF.

Our paper focuses on URS methods that provide formal guarantees on the uniformity of the distribution of generated samples. Several other approaches instead trade these guarantees for speed, resulting in approximate URS tools such as QuickSampler [16] and CMSGen [17]. URS is tightly connected to #SAT, an observation made in the seminal article by Jerrum et al. [22]. In [37], #SAT solvers were evaluated on industrial-size feature models. URS appears as a motivating application in need of efficient #SAT solvers, pointing particularly at the algorithm used in Smarch [30]. In [28], CNF transformations like the Tseitin transformation and their impact on feature model analysis have been investigated. While this is crucial for the transformation of feature models into CNF we used in our experiments, their work does not investigate URS.

2 Background

2.1 Feature Models

A *feature diagram* (FD) over a finite set of features $\mathcal{V}$ is a tuple $\mathcal{D} = \langle \mathcal{V}, \mathcal{E}, \mathcal{M}, op \rangle$ where

- $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is the set of edges defining a directed tree,
- $\mathcal{M} \subseteq \mathcal{V}$ is the subset of features marked as *mandatory*, and
- $op : \mathcal{V} \to \{\textsc{and}, \textsc{or}, \textsc{xor}, \textsc{leaf}\}$ assigns an operator to each node.

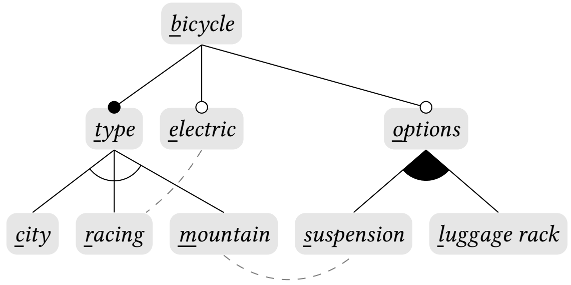
We call a feature diagram $\mathcal{D}$ together with a set C of propositional side constraints over $\mathcal{V}$ a *feature model* (FM).

Let ${}^{\rightarrow}x := \{y \colon (y, x) \in \mathcal{E}\}$ and $x^{\rightarrow} := \{y \colon (x, y) \in \mathcal{E}\}$ denote, respectively, the set of parents and children of x. A feature diagram $\mathcal{D}$ is *well-formed* if the following conditions hold:

(1) The edge relation $\mathcal{E}$ defines a tree, i.e., there is a designated root node $r \in \mathcal{V}$ s.t. ${}^{\rightarrow}r = \emptyset$, and for all other nodes $x \in \mathcal{V} \setminus \{r\}$, we have $|{}^{\rightarrow}x| = 1$.
(2) The operator cardinality is respected for each $x \in \mathcal{V}$:
   - If $op(x) = \textsc{leaf}$, then $|x^{\rightarrow}| = 0$.
   - If $op(x) = \textsc{and}$, then $|x^{\rightarrow}| \geq 1$.
   - If $op(x) \in \{\textsc{or}, \textsc{xor}\}$, then $|x^{\rightarrow}| \geq 2$.
(3) Mandatory nodes may only appear as children of AND nodes, i.e., for all $x \in \mathcal{V}$ where $op(x) \neq \textsc{and}$, we have $x^{\rightarrow} \cap \mathcal{M} = \emptyset$.

In the following, we only consider well-formed feature diagrams. The semantics of a feature diagram $\mathcal{D}$ is given as a set of constraints $C_{\mathcal{D}}$ in propositional logic over $\mathcal{V}$ as follows [32]:

- The root r needs to hold: $r \in C_{\mathcal{D}}$.
- Children imply their parents, so for every $x \in \mathcal{V}$ and $c \in x^{\rightarrow}$, we have $(c \to x) \in C_{\mathcal{D}}$.
- AND nodes imply their mandatory children: For $x \in \mathcal{V}$ with $op(x) = \textsc{and}$ and every $c \in x^{\rightarrow} \cap \mathcal{M}$, we have $(x \to c) \in C_{\mathcal{D}}$.
- OR nodes imply at least one of their children: For $x \in \mathcal{V}$ with $op(x) = \textsc{or}$, we have $\left(x \to \bigvee_{c \in x^{\rightarrow}} c\right) \in C_{\mathcal{D}}$.
- XOR nodes imply exactly one of their children: For $x \in \mathcal{V}$ with $op(x) = \textsc{xor}$, we have $\left(x \to \bigvee_{c \in x^{\rightarrow}} c\right) \in C_{\mathcal{D}}$ and $\left(x \to \neg(c \wedge c')\right) \in C_{\mathcal{D}}$ for $c, c' \in x^{\rightarrow}$ s.t. $c \neq c'$.



$$C = \{e \to \neg r, m \to s\}$$

$$\begin{aligned} C_{\mathcal{D}} = \{&b, b \to t, t \to b, e \to b, o \to b, \\ &t \to (c \vee r \vee m) \wedge \neg(c \wedge r) \wedge \neg(r \wedge m) \wedge \neg(c \wedge m), \\ &c \to t, r \to t, m \to t, o \to (s \vee l), s \to o, l \to o\} \end{aligned}$$

Figure 1: A feature model for a simple bicycle product line (feature diagram $\mathcal{D}$ at the top, set of side constraints C below). The underlined letters are used to abbreviate the feature names. $C_{\mathcal{D}}$ lists the constraints encoded in the diagram.

An example of a feature model is shown in Figure 1, where the root *bicycle* is an AND node, *type* is an XOR node, *options* is an OR node, and all other features are LEAF nodes. Below the AND node *bicycle*, only the child *type* is marked as mandatory. There are two side constraints, highlighted by the dashed lines in the diagram. Intuitively, $e \to \neg r$ states electric bicycles cannot be of type racing, while $m \to s$ assures mountain bikes always come with suspension.

Configurations. A *configuration* $\theta \colon \mathcal{V} \to \mathbb{B}$ is an assignment of each feature variable to either `true` or `false`. The set of all $2^{|\mathcal{V}|}$ possible assignments is denoted by Θ. For a propositional logic formula ϕ, its *valid configurations* are collected in the set $[\![\phi]\!] := \{\theta \colon \theta \models \phi\}$ of all configurations under which ϕ is true. We extend the notation to sets of formulas C in the intuitive way ($[\![C]\!] := \bigcap_{\phi \in C} [\![\phi]\!]$), and—in a slight abuse of notation—to feature diagrams: $[\![\mathcal{D}]\!] := [\![C_{\mathcal{D}}]\!]$. The valid configurations of $\mathcal{F}$ are given by $[\![\mathcal{F}]\!] := [\![\mathcal{D}]\!] \cap [\![C]\!]$. Finally, the *model count* of a structure $\mathcal{S}$ with configuration semantics is defined as the total number of satisfying configurations, i.e., $|[\![\mathcal{S}]\!]|$.

2.2 Model Counting in FDs

The number of valid configurations for a feature diagram $\mathcal{D}$ can be computed recursively with a single pass over $\mathcal{D}$ [40]. For each node $x \in \mathcal{V}$, we recursively define $\#(x)$ as:

$$\#(x) := \begin{cases} 1 & \text{for } op(x) = \textsc{leaf} \\ \prod_{c\in x^{\rightarrow}\cap\mathcal{M}} \#(c) \cdot \prod_{c\in x^{\rightarrow}\setminus\mathcal{M}} \big(\#(c)+1\big) & \text{for } op(x) = \textsc{and} \\ \sum_{c\in x^{\rightarrow}} \#(c) & \text{for } op(x) = \textsc{xor} \\ \big[\prod_{c\in x^{\rightarrow}} \big(\#(c)+1\big)\big] - 1 & \text{for } op(x) = \textsc{or} \end{cases}$$

Then $\#(x)$ is the model count of the sub-feature diagram rooted in node x. For a feature diagram $\mathcal{D}$ with root node r, we have $\#(r) = |[\![\mathcal{D}]\!]|$.

2.3 Uniform Random Sampling

Let $\Pr[X]$ denote the probability of event X. A *random sampler* $S_{\mathcal{T}}$ for a structure $\mathcal{T}$ is a probabilistic algorithm which produces a random sample $\theta \in [\![\mathcal{T}]\!]$ as output. $S_{\mathcal{T}}$ is *uniform* if for all configurations $\theta \in [\![\mathcal{T}]\!]$, we have

$$\Pr[S_{\mathcal{T}} = \theta] \;=\; \frac{1}{|[\![\mathcal{T}]\!]|}.$$

In [39], a recursive algorithm URS-FD has been provided to dedicatedly sample in feature diagrams uniformly at random. Algorithm 1 shows an adaption of this algorithm, iterating in breadth-first order over the diagram tree to construct a configuration θ. We assume the model count $\#(x)$ has been precomputed for every feature $x \in \mathcal{V}$. $\mathcal{B}_x$ is a boolean random variable with $\Pr[\mathcal{B}_x = \texttt{true}] \;=\; \frac{\#(x)}{\#(x)+1}$. `weighted_choice(`$\mathcal{W}$`)` is a probabilistic subroutine that takes a set $\mathcal{W} \subseteq \mathcal{V} \times [0,1]$ of elements and their respective weights and randomly selects one element with probability proportional to its weight. That is, for $(x, w) \in \mathcal{W}$, element x is chosen with probability $\frac{w}{w_{\text{total}}}$ where $w_{\text{total}} = \sum_{(x',w')\in\mathcal{W}} w'$. Note that in line 11, the algorithm loops until at least one child $c \in x^{\rightarrow}$ is selected (which happens almost-surely). This is not an issue in practice: since $\Pr[\mathcal{B}_c = \texttt{true}] \geq \frac{1}{2}$ for every c, for each loop iteration, the probability of selecting no child and having to repeat the loop is at most $(\frac{1}{2})^n$ for $n = |x^{\rightarrow}|$ the number of children.

3 Rejection Sampling

In this section, we extend the feature diagram sampling algorithm URS-FD to take a feature model's side constraints into account. To this end, we use *rejection sampling*, a generally applicable approach to uniform random sampling, and call the resulting algorithm URS-FD^{+}. Given a uniform random sampler $S_{\mathcal{T}}$ which allows sampling from a *sampling space* $[\![\mathcal{T}]\!]$, we can construct a *rejection sampler* $S_{\mathcal{T}\triangleright\mathcal{T}'}$ for every *solution space* $[\![\mathcal{T}']\!] \subseteq [\![\mathcal{T}]\!]$ as follows: draw candidate samples θ from $S_{\mathcal{T}}$ and return the first satisfying $\theta \in [\![\mathcal{T}']\!]$. Then $S_{\mathcal{T}'}$ is guaranteed to be uniform as well.

The practicality of rejection sampling hinges on two factors: how quickly can we generate and test the candidate samples, and how many candidates are rejected until a valid sample is found. We call the former quantity the *throughput* (*tp*), and define it as the product of the base sampler's sampling rate (*sr*) and the average time t_{check} to test a candidate sample:

$$tp(S_{\mathcal{T}\triangleright\mathcal{T}'}) := sr(S_{\mathcal{T}}) \cdot t_{\text{check}}.$$

Algorithm 1: `urs-fd`($\mathcal{D}$)

```
   input  : feature diagram $\mathcal{D} = \langle \mathcal{V}, \mathcal{E}, \mathcal{M}, op \rangle$
   output : uniform random sample $\theta\colon \mathcal{V} \to \mathbb{B}$
1  $\theta(root)$ := true
2  for $x \in \mathcal{V}$ in breadth-first order do
3      if $\theta(x)$ = false then
4          for $c \in x^{\rightarrow}$ do $\theta(c)$ := false
5      else
6          if $op(x) = \textit{AND}$ then
7              for $c \in x^{\rightarrow} \cap \mathcal{M}$ do $\theta(c)$ := true
8              for $c \in x^{\rightarrow} \setminus \mathcal{M}$ do $\theta(c) := \mathcal{B}_c$
9          if $op(x) = \textit{OR}$ then
10             $\mathcal{S} := \emptyset$
11             while $\mathcal{S} = \emptyset$ do
12                 $\mathcal{S} := \{c \in x^{\rightarrow} : \mathcal{B}_c = \texttt{true}\}$
13             for $c \in \mathcal{S}$ do $\theta(c)$ := true
14             for $c \in x^{\rightarrow} \setminus \mathcal{S}$ do $\theta(c)$ := false
15         if $op(x) = \textit{XOR}$ then
16             $c'$ := weighted_choice($\{(c, \#(c)) : c \in x^{\rightarrow}\}$)
17             $\theta(c')$ := true
18             for $c \in x^{\rightarrow} \setminus \{c'\}$ do $\theta(c)$ := false
19 return $\theta$
```

The second quantity is captured by the *hit rate*, the quotient of the size of solution space and the size of the sample space:

$$hit(S_{\mathcal{T}\triangleright\mathcal{T}'}) := \frac{|[\![\mathcal{T}']\!]|}{|[\![\mathcal{T}]\!]|}.$$

Multiplied together, throughput and hit rate define the *effective sampling rate* (*esr*) of a rejection sampler $S_{\mathcal{T}\triangleright\mathcal{T}'}$:

$$esr(S_{\mathcal{T}\triangleright\mathcal{T}'}) := hit(S_{\mathcal{T}\triangleright\mathcal{T}'}) \cdot tp(S_{\mathcal{T}\triangleright\mathcal{T}'}). \tag{1}$$

In the context of feature models, our solution space is the set $[\![\mathcal{F}]\!]$ of all valid configurations for the feature model $\mathcal{F}$. An immediate option for a sampling space is the set of all configurations Θ: Clearly, $[\![\mathcal{F}]\!] \subseteq \Theta$ holds, and we get a uniform random sampler S_{Θ} by constructing assignments with a 50/50 chance of assigning `true` or `false` for every feature. The hit rate for the resulting sampler $S_{\Theta\triangleright\mathcal{F}}$ is $hit(S_{\Theta\triangleright\mathcal{F}}) = |[\![\mathcal{F}]\!]|/2^{|\mathcal{V}|}$. Due to the exponent in the denominator, we expect hit rates to be low for most constrained systems.

The feature diagram sampler $S_{\mathcal{D}}$ from Section 2.3 is also suitable as a base sampler. Since $[\![\mathcal{F}]\!] = [\![\mathcal{D}]\!] \cap [\![C]\!]$, $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{D}]\!]$ holds, so we can construct the rejection sampler $S_{\mathcal{D}\triangleright\mathcal{F}}$. As the candidate samples already satisfy the diagram constraints by construction, checking the side constraints is sufficient for rejection. The hit rate for diagram-based rejection sampling is $hit(S_{\mathcal{D}\triangleright\mathcal{F}}) = |[\![\mathcal{F}]\!]|/|[\![\mathcal{D}]\!]|$.

Reversing the roles is imaginable as well, namely sampling from the side constraints and rejecting samples that are not valid for the feature diagram. While this still requires a full-fledged constraint sampler, there might be advantages as only a subset of all features usually appears in the side constraints. For example, a BDD-based sampler might be unable to construct a BDD for the whole model within the given memory and time limits, but could manage to construct a BDD for just the side constraints. Yet, the candidate samples still need to cover all features $\mathcal{V}$. Let $\mathcal{V}_C$ denote the subset of features appearing in the side constraints in C, and $[\![C]\!]_{\mathcal{V}_C}$ the set of valid partial samples over the features in $\mathcal{V}_C$. Then all features in $\mathcal{V} \setminus \mathcal{V}_C$ can be freely assigned, which exponentially increases the size of the sample space $[\![C]\!]$. The hit rate of the side constraint-based rejection sampler $S_{C \triangleright \mathcal{F}}$ is thus given by

$$hit(S_{C \triangleright \mathcal{F}}) = \frac{|[\![\mathcal{F}]\!]|}{|[\![C]\!]|} = \frac{|[\![\mathcal{F}]\!]|}{|[\![C]\!]_{\mathcal{V}_C}| \cdot 2^{|\mathcal{V} \setminus \mathcal{V}_C|}}.$$

3.1 Implementation

We implemented two variants of the diagram-based rejection sampler $S_{\mathcal{D} \to \mathcal{F}}$ in Rust. The first variant, Zampler-RS, is straightforward: candidate samples θ are generated using Algorithm 1 and checked against the side constraints by evaluating each constraint $\phi \in C$ directly under the assignment θ.

The second variant, Zampler-RS-BDD, seeks to reduce the time needed for solution checking by constructing a binary decision diagram for the conjunction of all $\phi \in C$. This introduces initial overhead for BDD construction, but allows to check each sample in time linear in the number of features. Additionally, Algorithm 1 is adapted to perform satisfiability checks on partial samples during generation, allowing early rejection before samples are fully constructed. Ideally, if the BDD's variable ordering aligns with the feature diagram's breadth-first order, sample creation and checking proceed smoothly together: as Algorithm 1 traverses the diagram, each assignment of a feature involved in side constraints is mirrored in the BDD by following the high or low edge to the appropriate level. However, the BDD's variable ordering is critical for handling larger models efficiently. Preliminary experiments showed that using orderings consistent with the feature diagram's partial order does not scale well. Therefore, we use heuristics [15] and use a buffered approach for checking partial assignments.

4 Evaluation

4.1 Research Questions

The overarching goal of our experiments is to assess the performance of uniform random samplers on feature models with non-trivial feature diagrams, i.e., feature diagrams where the hierarchical structure contains structural domain knowledge. We hence aim to establish a baseline on existing URS approaches for feature models:

(RQ1) Which URS approaches and tools perform best on feature models for various sample sizes, also taking knowledge compilation resources into account?

Given the linear-time complexity of URS-FD, we ask whether URS-FD can meet or exceed the performance of state-of-the-art URS approaches on feature models without side constraints.

(RQ2) How competitive is URS-FD compared to URS tools on feature models without side constraints?

We anticipate URS-FD to outperform general-purpose tools. However, since real-world feature models without any side constraints are uncommon, answering this question primarily serves to estimate the baseline potential for URS-FD$^+$, extending URS-FD to also support side constraints.

Finally, we are looking for factors indicating that one sampler might be preferable over the others for a specific feature model:

(RQ3) What factors determine which sampling methods perform best for a given feature model?

4.2 Experiment Setup

Our experiment setup is driven by our research questions, selecting state-of-the-art URS tools for feature models in CNF to compare with our implementations of URS-FD and URS-FD$^+$ on community benchmarks comprising non-trivial feature diagrams.

URS Tools. We consider the following samplers from the literature, in their order of appearance: SPUR [3], KUS [33], and BDDSampler [19]. Each tool was introduced with claims of outperforming its predecessors. The URS tools Smarch [30] and UniGen3 [35] have shown great advancements when they have been introduced, but their performance has shown to be not competitive with the tools above (see, e.g., [19]). For a comparison with dedicated URS approaches on feature diagrams, we also consider the rejection samplers implementing URS-FD$^+$ described in Section 3.1, Zampler-RS and Zampler-RS-BDD. Additionally, we consider an alternative BDD-based sampler, Zampler-BDD, which constructs the BDD directly from the feature diagram and the side-constraints instead of a CNF. While BDDSampler uses CUDD [34] for BDD compilation, our implementation Zampler-BDD relies on the recently introduced OxiDD library [21].

Benchmark Set. We collected a set of 46 feature models from the literature [6, 24, 27], ranging in size from 12 to 18,616 features and 1 to 3,545 side constraints.[2] All models are provided in the XML format used by FeatureIDE [38] and consist of a non-trivial feature diagram and a collection of side constraints as propositional formulas. Our implementation Zampler operates directly on these XML files. SPUR and KUS require input as a single CNF formula in DIMACS format, which we generated with FeatureIDE for all models in the benchmark set. For BDDSampler, input is in the form of a VAR file listing all features and an EXP file for the constraints, which we extract from the DIMACS files using dimagic [15]. For RQ1, we used the same 46 models and stripped away their side constraints before re-exporting them as XML and DIMACS.

Operationalization. To evaluate competitiveness of the different URS approach, we consider both runtime and the capacity to generate large sample volumes. As part of RQ3, we investigate the rejection sampling approaches in reference to the hit rate and throughput defined in Section 3. All experiments were run on an Intel Core i9-10900K with 64 GB RAM running Ubuntu 22.04. Throughout, a timeout of 10 minutes (600 s) was enforced.

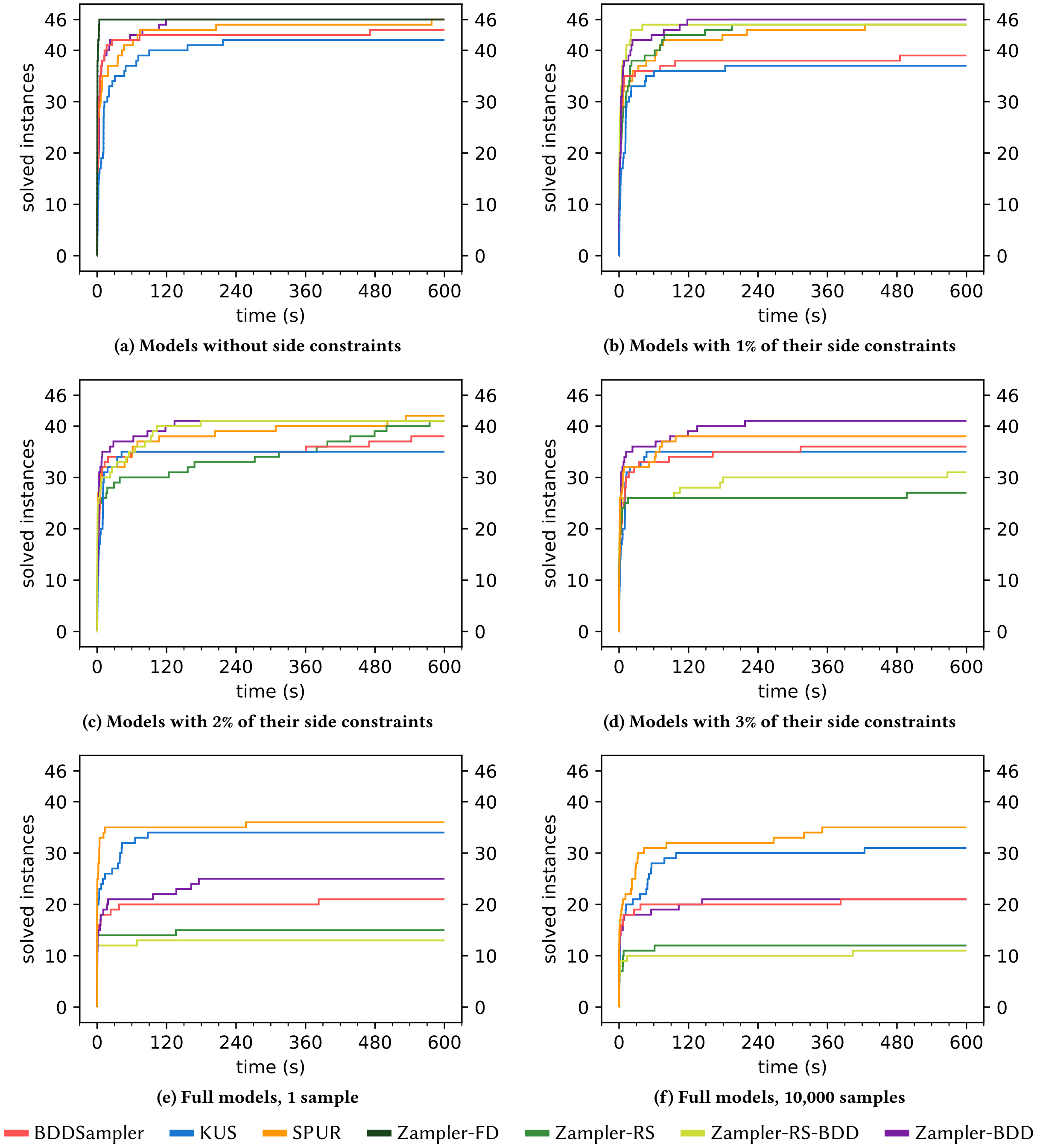
4.3 Results

Figure 2 shows cactus plots for different variants of the benchmark set, indicating for how many of the 46 models the sampling task finished within the time indicated on the x-axis. The task for each solver was to generate 10,000 samples, except for Figure 2e where times to generate a single sample are shown.

4.3.1 Research Question RQ1.

Figure 2f shows sampling times for the complete models, i.e., including all side constraints. Clearly,

[2] Including 12 artificially generated models from https://github.com/skrieter/MIG-Evaluation/tree/master/de.ovgu.featureide.fm.benchmark/models



(a) Models without side constraints

(b) Models with 1% of their side constraints

(c) Models with 2% of their side constraints

(d) Models with 3% of their side constraints

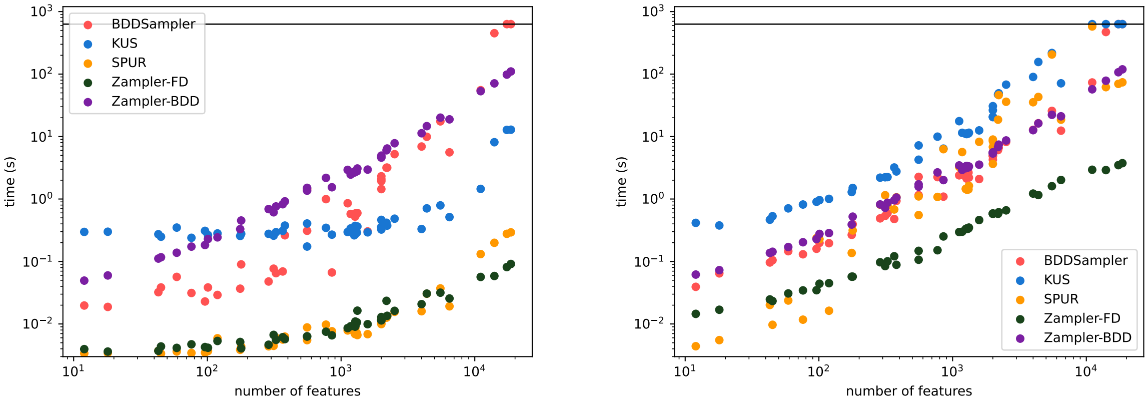
(e) Full models, 1 sample

(f) Full models, 10,000 samples

Figure 2: Runtime to generate 10,000 samples on models with varying percentage of randomly selected side constraints.

the two rejection sampling methods, Zampler-RS and Zampler-RS-BDD, are not competitive in this setting. Surprisingly, the simpler implementation, Zampler-RS, successfully handles one more model in total than the more optimized Zampler-RS-BDD. Our investigation in the following section, addressing RQ3, will explore possible reasons for this outcome.

Interestingly, KUS was shown in [33] to outperform SPUR across 1,425 benchmarks from various domains, although there is no overlap with our benchmark set. We find the roles reversed, with SPUR consistently in the lead. The same can be seen in [19] (cf. Table 3), where benchmarks partly overlap with ours. In that table, BDDSampler outperforms SPUR on most models, which contrasts sharply



(a) Times to generate one sample **(b) Times to generate 10,000 samples**

Figure 3: Sampling times in relation to the number of features for models without side constraints.

with our findings. This discrepancy is likely due to the use of precompiled BDDs as input for BDDSampler, precluding a fair comparison. The same applies to KUS, where reported timings exclude the d-DNNF compilation step.

Figure 2e shows times for generating a single sample. The results show that BDDSampler is largely unaffected by the number of requested samples, suggesting that BDD construction time dominates the total sampling time. For the other samplers, a few additional benchmarks are solvable when only a single sample is requested, although overall performance remains similar regardless of the sample count requested.

Answering **RQ1**, we find SPUR to be the fastest sampler on feature models with side constraints.

4.3.2 Research Question RQ2. Figure 2a presents results for sampling only the feature diagram without side constraints. As expected, Zampler-FD implementing Algorithm 1 outperforms all other solvers, generating 10,000 samples in less than 4 seconds for each model. SPUR succeeds for all benchmarks in 10 minutes, while BDDSampler and KUS time-out for 2 and 4 models, respectively.

We also investigated how model size and the number of requested samples impact performance. The scatter plots in Figure 3 show sampling times for all models, with size on the x-axis and sampling time on the y-axis. The left plot represents single-sample times, while the right plot shows times for generating 10,000 samples. Both axes use a logarithmic scale. We observe that the diagram-based approach Zampler-FD and SPUR are comparably fast when generating a single sample, but with Zampler-FD in the lead for larger sample sizes. Notably, our BDD-based reference implementation Zampler-BDD looks almost indifferent to the number of requested samples. To explore the implementations' limits, we also requested 10^6 samples per model. Zampler-FD is able to generate a million samples for each feature diagram, taking less than 7 minutes for the largest model `Automotive02_V4`. In contrast, KUS and SPUR achieve this only for 18 and 8 of the smaller models, respectively.

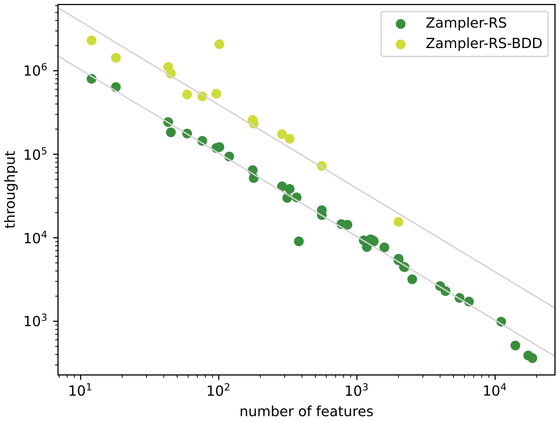


Figure 4: Throughput measures for full models.

At this scale, file sizes become substantial enough that I/O performance may influence results, making even larger sample sets less practical to test. Summarizing, we obtain the expected result:

In answer to **RQ2**, we find that Zampler-FD outperforms state-of-the-art URS tools on feature models without side constraints.

4.3.3 Research Question RQ3. Given a feature model, which tool should we use to sample from it? From the discussion of the previous two research questions, we know that SPUR is likely a good overall choice, though Zampler-RS is preferable when our model contains no side constraints and we need many samples. What if we have only a small number of side constraints? Figures 2b to 2d show variants of the benchmark set where only 1%, 2%, or 3% of the models' side constraint were randomly selected and all others removed. While for all solvers the number of solvable benchmarks generally decreases the more side constraints are included, the drop is especially pronounced for the rejection-sampling approaches Zampler-RS and Zampler-RS-BDD.

benchmarks	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{V}_\mathcal{C}\rvert$	$\frac{\lvert\mathcal{V}_\mathcal{C}\rvert}{\lvert\mathcal{V}\rvert}$	$\log_{10}(\cdot)$				$\log_{10}(hit(\cdot))$			**Zampler-RS**			**Zampler-RS-BDD**		
					$\lvert\Theta\rvert$	$\lvert[\![\mathcal{F}]\!]\rvert$	$\lvert[\![\mathcal{D}]\!]\rvert$	$\lvert[\![\mathcal{C}]\!]\rvert$	$S_{\mathcal{D}\triangleright\mathcal{F}}$	$S_{\mathcal{C}\triangleright\mathcal{F}}$	$S_{\Theta\triangleright\mathcal{F}}$	*tp*	*hit**	**out.**	*tp*	*hit**	**out.**
pizzas	12	1	2	16.7%	3.6	1.6	1.7	3.2	-0.1	-1.6	-2.0	800,453.2	-8.7	TP	2,304,322.4	-9.3	TP
GPL	18	13	12	66.7%	5.4	2.6	3.7	4.0	-1.1	-1.3	-2.8	637,867.5	-8.5	TP	1,429,037.2	-9.2	TP
mobilemedia2	43	3	5	11.6%	12.9	6.3	6.7	12.3	-0.3	-6.0	-6.6	242,955.7	-8.2	TP	1,112,852.7	-8.8	TP
jHipster	45	13	25	55.6%	13.5	4.4	5.6	12.1	-1.2	-7.7	-9.1	183,003.7	-8.1	TP	923,299.6	-8.8	TP
aafms_framework	59	14	30	50.8%	17.8	11.1	13.1	15.6	-2.0	-4.4	-6.6	176,967.2	-8.0	TP	519,073.4	-8.7	TP
berkeleyDB1	76	20	32	42.1%	22.9	9.6	14.0	18.1	-4.4	-8.5	-13.3	144,556.3	-7.9	TP	493,043.5	-8.5	TP
KConfig_axTLS	96	14	30	31.3%	28.9	11.9	14.8	26.1	-2.8	-14.2	-17.0	118,824.5	-7.8	TP	530,379.0	-8.4	TP
violet	101	27	67	66.3%	30.4	18.8	26.2	24.5	-7.4	-5.7	-11.6	121,891.0	-7.8	TP	2,074,763.7	-8.4	TP
berkeleyDB2	119	68	97	81.5%	35.8	9.9	29.5	15.8	-19.5	-5.9	-25.9	93,954.3	-7.7	TN		-8.3	TN
BankingSoftware	176	4	5	2.8%	53.0	31.7	31.9	52.4	-0.2	-20.7	-21.3	64,490.4	-7.5	TP	256,989.4	-8.2	TP
WeaFQAs	179	7	13	7.3%	53.9	24.5	25.0	52.8	-0.5	-28.3	-29.4	51,895.4	-7.5	TP	232,485.3	-8.2	TP
eShopSplot	287	21	34	11.8%	86.4	49.4	50.8	83.9	-1.4	-34.5	-37.0	41,223.6	-7.3	TP	173,894.9	-8.0	TP
KConfig_uClibc	313	56	136	43.5%	94.2	40.2	54.3	84.0	-14.1	-43.8	-54.0	30,005.4	-7.3	TN		-7.9	TN
eShopFIDE	326	21	34	10.4%	98.1	49.4	50.8	95.6	-1.4	-46.2	-48.8	38,480.2	-7.3	TP	153,267.2	-7.9	TP
DMIE	366	192	342	93.4%	110.2	97.6	99.0	108.5	-1.3	-10.9	-12.6	30,437.5	-7.2	TP		-7.9	**FP**
KConfig_uClinux-base	380	3,455	299	78.7%	114.4	22.4	22.4	60.0	0.0	-37.6	-92.0	9,056.1	-7.2	TP		-7.8	**FP**
FinancialServices_2017	557	1,001	507	91.0%	167.7	2.6	15.6	134.8	-13.0	-132.2	-165.0	21,366.8	-7.0	TN		-7.7	TN
splot505	557	50	47	8.4%	167.7	65.0	69.2	164.5	-4.2	-99.5	-102.7	18,687.1	-7.0	TP	72,439.7	-7.7	TP
KConfig_busybox-1.18.0	854	123	225	26.3%	257.1	201.3	212.9	243.8	-11.6	-42.5	-55.8	14,316.5	-6.9	TN		-7.5	TN
splot1001	1,120	100	95	8.5%	337.2	125.2	133.1	331.2	-7.9	-205.9	-211.9	9,314.0	-6.7	TN		-7.4	TN
KConfig_embtoolkit	1,179	323	598	50.7%	354.9	96.7	185.8	293.3	-89.1	-196.5	-258.2	7,741.6	-6.7	TN		-7.3	TN
CDL_adder	1,285	890	1,086	84.5%	386.8	124.9	214.1	297.9	-89.2	-173.0	-261.9	9,409.2	-6.7	TN		-7.3	TN
CDL_rattler	1,324	894	1,087	82.1%	398.6	136.5	226.5	309.4	-90.0	-172.9	-262.1	9,066.9	-6.7	TN		-7.3	TN
KConfig_uClinux-distr.	1,580	197	294	18.6%	475.6	409.6	419.5	460.9	-9.8	-51.3	-66.0	7,667.2	-6.6	TN		-7.2	TN
Gen2000-3CNF-1_100	2,000	100	287	14.4%	602.1	329.4	337.0	595.9	-7.6	-266.6	-272.7	5,445.9	-6.5	**FN**	15,494.6	-7.1	**FN**
Automotive_small	2,513	2,833	1,280	50.9%	756.5	210.7	303.4	615.0	-92.7	-404.3	-545.8	3,185.8	-6.4	TN		-7.0	TN
KConfig_linux-2.6.33.3	6,467	3,545	3,994	61.8%	1,946.8	timeout	1,672.6	timeout				1,722.1	-6.0			-6.6	
Automotive02_V1	14,010	666	805	5.7%	4,217.4	1,260.7	1,300.7	4,152.0	-40.0	-2891.3	-2956.7	512.6	-5.6	TN		-6.3	TN
Automotive02_V4	18,616	1,369	1,501	8.1%	5,604.0	1,534.2	1,596.2	5,494.0	-62.0	-3959.8	-4069.7	362.2	-5.5	TN		-6.2	TN

Table 1: Properties, model counts, and hit rate prediction for selected feature models from the benchmark set.

Taking a step back, we look into the feasibility of the rejection-sampling approaches described in Section 3 in light of the properties of our benchmark models: the feature-based sampler $S_{\Theta\triangleright\mathcal{F}}$, diagram-based sampler $S_{\mathcal{D}\triangleright\mathcal{F}}$, and side constraint-based sampler $S_{\mathcal{C}\triangleright\mathcal{F}}$. Table 1 shows a selection of feature models from the benchmark set along with their number of features $|\mathcal{V}|$, number of side constraints $|\mathcal{C}|$, and the percentage of features appearing in the side constraints $|\mathcal{V}_\mathcal{C}|/|\mathcal{V}|$. We note that the latter highly varies, ranging from less than 3% to over 90% of features in the side constraints, and seemingly unrelated to the overall number of features.

To compare the hit rates of the three approaches, we used the model counter SharpSAT-TD [26] to compute the size of the solution space $|[\![\mathcal{F}]\!]|$ and the sample space $|[\![\mathcal{C}]\!]|$, and the algorithm shown in Section 2.2 for $|[\![\mathcal{D}]\!]|$. Recall that $|\Theta| = 2^{|\mathcal{V}|}$. The resulting hit rates, computed as defined in Section 3, are also given in Table 1. To allow comparison, all values are given in $\log_{10}$ scale. As expected, $hit(S_{\Theta\triangleright\mathcal{F}})$ is extremely low for all but the smallest models. While much better in comparison, the hit rates for the diagram-based rejection sampler $S_{\mathcal{D}\triangleright\mathcal{F}}$ are still very low for the larger models, providing a plausible reason for the poor performance of the USR-FD$^+$ approaches on the full models. Hit rates of the side-constraint-based rejection sampler $S_{\mathcal{C}\triangleright\mathcal{F}}$ fall in between the other two, except for the two models `violet` and `berkeleyDB2`. This indicates $S_{\mathcal{D}\triangleright\mathcal{F}}$ has mostly greater chances to succeed than $S_{\mathcal{C}\triangleright\mathcal{F}}$, but averse model characteristics may exist that tip the scale in favor of the latter.

One more outlier sticks out: `KConfig_uClinux-base` has a hit rate of 1 for the diagram sampler $S_{\mathcal{D}\triangleright\mathcal{F}}$, because the sample space $[\![\mathcal{D}]\!]$ and solution space $[\![\mathcal{F}]\!]$ have the same size. As $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{D}]\!]$ holds, this entails that $[\![\mathcal{F}]\!] = [\![\mathcal{D}]\!]$. In other words, the model's 3,455 side constraints only encode constraints that are already implied by the diagram, so $S_{\mathcal{D}\triangleright\mathcal{F}}$ never has to reject a sample.[3]

We measured the throughput of Zampler-RS and Zampler-RS-BDD running on the full feature models, i.e., the number of candidate samples generated and tested per second. The respective columns in Table 1 display the average throughput per second after running for 1 minute, with the blank cells for Zampler-RS-BDD indicating the BDD construction timed out after 10 minutes. We find that the early rejection optimization implemented in Zampler-RS-BDD significantly increases throughput, though the applicability is limited to the models where a BDD for the side constraints can be constructed. Figure 4 plots the throughput in relation to the size of each model. An inverse linear relation $tp \sim \frac{1}{|\mathcal{V}|}$ is clearly visible, allowing us to approximate $tp(S_{\mathcal{D}\triangleright\mathcal{F}}) \approx \frac{\alpha}{|\mathcal{V}|}$ with some constant α. Computing the geometric mean of the measured throughput multiplied by the number of features for each model, we get $\alpha = 1.03 \cdot 10^7$ for Zampler-RS and $\alpha = 3.9 \cdot 10^7$ for Zampler-RS-BDD, plotted as lines in Figure 4. With this estimate at hand, we can calculate the necessary hit rate which still guarantees a given effective sample rate. Rearranging Equation (1), we get a predicted *minimal hit rate*

[3] This explains why `KConfig_uClinux-base` is one of the two models for which Zampler-RS-BDD fails while Zampler-RS succeeds: for the former, the BDD construction times out.

hit* as follows:

$$hit^*(S_{\mathcal{D}\triangleright\mathcal{F}}) = \frac{esr(S_{\mathcal{D}\triangleright\mathcal{F}})}{tr(S_{\mathcal{D}\triangleright\mathcal{F}})} = esr(S_{\mathcal{D}\triangleright\mathcal{F}})\frac{|\mathcal{V}|}{\alpha}.$$

As an example, assume we have a feature model $\mathcal{F}$ with $|\mathcal{V}| = 2000$ features and want to be able to generate 1,000 samples in less than three hours. Then $esr(S_{\mathcal{D}\triangleright\mathcal{F}}) \leq \frac{1000}{3\cdot 60\cdot 60}$, and we get $hit(S_{\mathcal{D}\triangleright\mathcal{F}}) \geq \frac{1000\cdot 2000}{3\cdot 60\cdot 60\cdot\alpha}$. Using Zampler-RS, the hit rate then has to be at least $1.8 \cdot 10^5$, and more than $4.7 \cdot 10^6$ for Zampler-RS-BDD. Finally, to check if a URS-FD$^+$ approach is possible for $\mathcal{F}$, we can compute $|[\![\mathcal{F}]\!]|$ with a model counter and check if $|[\![\mathcal{F}]\!]| \geq hit^*(S_{\mathcal{D}\triangleright\mathcal{F}})\cdot|[\![\mathcal{D}]\!]|$ holds. The *hit** columns in Table 1 give the predicted minimal hit rates to achieve an effective sample rate of at least one sample in 10 minutes, corresponding to the benchmarks in Figure 2e. The cells are colored green if $hit^* < hit(S_{\mathcal{D}\triangleright\mathcal{F}})$, and orange otherwise. The columns labeled *out* report the outcome of the prediction: true positive/negative (TP/TN) or false positive/negative (FP/FN). For almost all models, the predictions hold up. The false positives for Zampler-RS-BDD only appear in cases where the BDD construction timed out.

Answering **RQ3**, we find that the applicability of URS-FD$^+$ approaches can be predicted from size of the feature model and the overall model count. With few side-constraints, URS-FD$^+$ is superior to diagram-agnostic URS tools.

5 Threats to Validity

Internal Validity. We mitigated performance-influencing side-effects in our experiments by running each experiment on a single core, also leaving one core unoccupied for garbage collection and maintenance threads. Further, we ran the experiments slightly longer and introduced margin on memory granted. Approaches that involve knowledge compilation and SAT solvers usually involve randomized optimizations. We hence employed state-of-the-art heuristics, e.g., relying on tree decompositions [33] for d-DNNF compilation or hypergraph cutting for BDD construction [15]. Internal risks however could be further reduced by running randomized parts of the experiments multiple times, averaging results.

External Validity. Our findings might not hold for all kinds of feature models. However, we chose from a wide range of feature models from different areas that have been shown to be representative in the field [27]. Due to the clear relation of distribution of number of features and results with only few outliers (see Figures 3 and 4) we expect our results to smoothly extend to other feature models not contained in our benchmark set. There are several methods to obtain CNF representations from feature models, possibly leading to different results also for the considered URS tools [28]. We chose the standard method well-accepted in the community and shown to be suitable in most of the cases by using FeatureIDE [38].

6 Conclusion

This paper presents the first comparative evaluation of URS approaches applied to non-trivial feature diagrams. Our findings highlight SPUR as the most effective sampler, closely followed by KUS. For feature models without side constraints, sampling directly from the feature diagram is not only theoretically superior, but also practically as demonstrated by our implementation. However, generalizing this method to handle side constraints through rejection sampling is worthwhile only for instances with very high hit rates. When the total number of valid configurations is known, we propose a method to estimate the expected hit rate and assess whether the desired number of samples can be produced within a specified time budget. In practice, we find only small feature models to have sufficiently high hit rates. For larger models commonly encountered in the literature, not enough constraints are encoded in the diagram to make diagram-based approaches viable.

References

[1] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. 2021. BURST: A Benchmarking Platform for Uniform Random Sampling Techniques. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*. ACM, Leicester United Kindom, 36–40. https://doi.org/10.1145/3461002.3473070

[2] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. 2023. BURST: Benchmarking Uniform Random Sampling Techniques. *Science of Computer Programming* 226 (March 2023), 102914. https://doi.org/10.1016/j.scico.2022.102914

[3] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *Theory and Applications of Satisfiability Testing – SAT 2018*. Vol. 10929. Springer International Publishing, Cham, 135–147. https://doi.org/10.1007/978-3-319-94144-8_9

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7

[5] Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th Workshop on Feature-Oriented Software Development (FOSD)*. 1–8.

[6] Danilo Beuche. 2008. Modeling and Building Software Product Lines with Pure::Variants. In *2008 12th International Software Product Line Conference*. IEEE, Limerick, Ireland, 358–358. https://doi.org/10.1109/SPLC.2008.53

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press. https://doi.org/10.3233/FAIA336

[8] Tomas Borovicka, Marcel Jirina Jr., Pavel Kordik, and Marcel Jirina. 2012. Selecting Representative Data Sets. In *Advances in Data Mining Knowledge Discovery and Applications*. IntechOpen, Rijeka, Chapter 2. https://doi.org/10.5772/50787

[9] Randal E Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[10] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41 (2003), 115–141.

[11] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 9035. Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319. https://doi.org/10.1007/978-3-662-46681-0_25

[12] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *Computer Aided Verification*. Vol. 8044. Springer Berlin Heidelberg, Berlin, Heidelberg, 608–623. https://doi.org/10.1007/978-3-642-39799-8_40

[13] Paul Clements and Linda Northrop. 2001. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional.

[14] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *J. Artif. Int. Res.* 17, 1 (Sept. 2002), 229–264.

[15] Clemens Dubslaff, Nils Husung, and Nikolai Käfer. 2024. Configuring BDD Compilation Techniques for Feature Models. In *28th ACM International Systems and Software Product Line Conference*. ACM, Dommeldange Luxembourg, 209–216. https://doi.org/10.1145/3646548.3676538

[16] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 549–559. https://doi.org/10.1145/3180155.3180248

[17] Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. 2021. Designing Samplers Is Easy: The Boon of Testers. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. TU Wien. https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_31

[18] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, and David Benavides. 2020. 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. ACM, Montreal Quebec Canada, 1–11. https://doi.org/10.1145/3382025.3414951

[19] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, David Benavides, and Don Batory. 2022. Uniform and Scalable Sampling of Highly Configurable Systems. *Empirical Software Engineering* 27, 2 (March 2022), 44. https://doi.org/10.1007/s10664-021-10102-5

[20] Tobias Heß, Tim Jannik Schmidt, Lukas Ostheimer, Sebastian Krieter, and Thomas Thüm. 2024. UnWise: High T-Wise Coverage from Uniform Sampling. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, Bern Switzerland, 37–45. https://doi.org/10.1145/3634713.3634716

[21] Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl. 2024. OxiDD: A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust. In *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 14572. Springer Nature Switzerland, Cham, 255–275. https://doi.org/10.1007/978-3-031-57256-2_13

[22] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science* 43 (1986), 169–188. https://doi.org/10.1016/0304-3975(86)90174-X

[23] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

[24] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch between Real-World Feature Models and Product-Line Research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 291–302. https://doi.org/10.1145/3106237.3106252

[25] Donald E. Knuth. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams* (12th ed.). Addison-Wesley Professional.

[26] Tuukka Korhonen and Matti Järvisalo. 2023. SharpSAT-TD in Model Counting Competitions 2021-2023. https://doi.org/10.48550/ARXIV.2308.15819

[27] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 898–909. https://doi.org/10.1145/3180155.3180159

[28] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or Not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–13. https://doi.org/10.1145/3551349.3556938

[29] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 61–71. https://doi.org/10.1145/3106237.3106273

[30] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. University of Texas at Austin, Department of Computer Science.

[31] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 240–251. https://doi.org/10.1109/ICST.2019.00032

[32] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*. 139–148. https://doi.org/10.1109/RE.2006.23

[33] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 620–602. https://doi.org/10.29007/h4p9

[34] Fabio Somenzi. 2015. *CUDD: CU Decision Diagram Package*. Technical Report. University of Colorado at Boulder.

[35] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. 2020. Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling. In *Computer Aided Verification*. Vol. 12224. Springer International Publishing, Cham, 463–484. https://doi.org/10.1007/978-3-030-53288-8_22

[36] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. 2024. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *28th ACM International Systems and Software Product Line Conference*. ACM, Dommeldange Luxembourg, 54–65. https://doi.org/10.1145/3646548.3672590

[37] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art # SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering* 28, 2 (March 2023), 29. https://doi.org/10.1007/s10664-022-10265-9

[38] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (Jan. 2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[39] Mathieu Vavrille, Erwan Meunier, Charlotte Truchet, and Charles Prud'Homme. 2023. *Linear Time Computation of Variation Degree and Commonalities on Feature Diagrams*. Technical Report RR-2023-01-DAPI. Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, LS2N, UMR 6004, F-44000 Nantes, France.

[40] Thomas Von Der Maßen and Horst Lichter. 2005. Determining the Variation Degree of Feature Models. In *Software Product Lines*. Vol. 3714. Springer Berlin Heidelberg, Berlin, Heidelberg, 82–88. https://doi.org/10.1007/11554844_9

[41] John von Neumann. 1951. Various techniques used in connection with random digits. In *Monte Carlo Method*. National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 36–38.

[42] Jun Yuan, K. Albin, A. Aziz, and C. Pixley. 2002. Simplifying Boolean constraint solving for random simulation-vector generation. In *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002*. 123–127. https://doi.org/10.1109/ICCAD.2002.1167523

[43] Olivier Zeyen, Maxime Cordy, Gilles Perrouin, and Mathieu Acher. 2024. Preprocessing Is What You Need: Understanding and Predicting the Complexity of SAT-based Uniform Random Sampling. In *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)*. ACM, Lisbon Portugal, 23–32. https://doi.org/10.1145/3644033.3644371

Covering T-Wise Interactions of Deployed Configurations

Rahel Sundermann
University of Ulm
Ulm, Germany
rahel.sundermann@uni-ulm.de

Sabrina Böhm
University of Ulm
Ulm, Germany
sabrina.boehm@uni-ulm.de

Sebastian Krieter
TU Braunschweig, Germany
Braunschweig, Germany
sebastian.krieter@tu-braunschweig.de

Malte Lochau
University of Siegen
Siegen, Germany
malte.lochau@uni-siegen.de

Thomas Thüm
TU Braunschweig, Germany
Braunschweig, Germany
thomas.thuem@tu-braunschweig.de

Abstract

Software product lines are a common way to develop individually configurable products. In practice, already deployed products have to undergo frequent changes throughout the whole life-cycle due to unavoidable updates (e.g., to close critical vulnerabilities). Such software updates require additional testing effort to reduce the risk of unwanted behavior of deployed functionality. However, existing sampling techniques are ignorant of deployed configurations coming from different points in a system's evolution. Furthermore, available techniques aim to cover all configurations instead of focussing on deployed configurations. Exploiting this knowledge can help to improve efficiency and effectiveness of sample-based quality assurance of product updates. To address this gap, we introduce a new sampling technique that guarantees coverage of all t-wise interactions on a given set of deployed configurations (i.e., field configurations). As our evaluation shows, state-of-the-art sampling approaches that do no exploit this additional knowledge are not able to cover all of these interactions. Incorporating knowledge about field configurations improves the effectiveness of testing while ensuring full coverage of interactions in deployed products.

CCS Concepts

• **Software and its engineering → Software product lines**; **Feature interaction**; Software testing and debugging.

Keywords

product line, field configuration, sampling

ACM Reference Format:
Rahel Sundermann, Sabrina Böhm, Sebastian Krieter, Malte Lochau, and Thomas Thüm. 2025. Covering T-Wise Interactions of Deployed Configurations. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3715340.3715432

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715432

1 Introduction

To support configurability of software systems, many companies develop their products as a product line [10, 48]. Different features are offered, where the selection of features is constrained by a feature model [5, 8]. A customer can choose which of the features to in- or exclude, as long as their choices do not contradict the constraints of the feature model [5, 9]. This gives the customer the possibility to configure a product to individually meet their needs.

One challenge that arises when developing a product line is the additional testing effort compared to a single system, as each valid combination of features should behave correctly [1, 18, 30]. Because of the large number of possible products, testing all of them is infeasible. To reduce the testing effort, t-wise sampling has been proposed, which aims to create a set of configurations (i.e., a sample) that is as small as possible while containing all valid combinations between t features [1, 19, 30, 37, 49].

Deployed products also have to undergo a testing process each time they receive a software update, which might be unavoidable for, e.g., critical security updates. While t-wise sampling has been well researched in the last years [33, 35, 49], creating efficient and effective t-wise samples for deployed configurations (i.e., field configurations) has not yet been investigated. Taking field configurations into account is nevertheless important, because updates can create new possible unwanted feature interactions in the field configurations, leading to a fault in the deployed product [4, 20].

Existing t-wise sampling techniques are not sufficient for ensuring the quality of updated field configurations because of three important reasons. First, existing field configurations are usually derived from older versions of the feature model, as they are built and bought at different points in time and real-world product lines evolve [34, 47]. This results in configurations in the field that may be invalid for the current feature model of the product line, leading to a discrepancy between field configurations and configurations supported from the current feature model. However, ignoring the existing field configurations leads to a lack of effectiveness in the resulting sample. Second, the t-wise sample of a feature model aims to cover all interactions, whereas many of them might not actually appear in the field configurations, because not every possible feature combination has necessarily been chosen by a customer. This increase in sample size would decrease efficiency. Third, in real-world product lines, there might be a lack of information regarding,

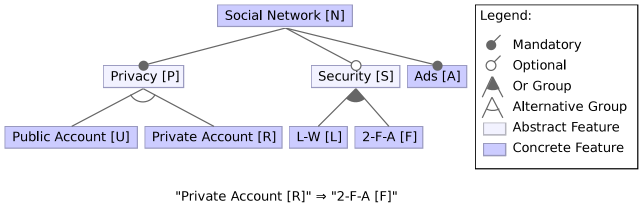


Figure 1: FM Running Example

for instance, past feature model versions [10]. Most existing sampling techniques require a feature model as input [35] and are not developed to consider deprecated configurations.

In this paper, we propose an algorithm for creating a sample covering t-wise interactions of a given set of field configurations from different points in time. Instead of a feature model, our algorithm takes a set of configurations as input for creating a t-wise sample and creates a subset of the given configurations. It is based on a greedy heuristic, because we aim to solve an instance of the NP-hard minimum set cover problem and greedy is the best known approximation algorithm. We present an evaluation of the algorithm to investigate the time needed to create the sample and the size of the sample (i.e., the number of contained configurations). We examine the impact of the fact that existing sampling techniques were not developed with the aim of doing quality assurance for field configurations. This includes comparing their sample size to the sample size of our approach as well as how well they cover the interactions from a given set of field configurations.

Overall, we make the following contributions:

(1) We investigate the weaknesses of existing sampling techniques when applied to field configurations.
(2) We present a greedy algorithm for covering t-wise interactions in field configurations, based on a reduction to the minimum set cover problem.
(3) We evaluate the algorithm regarding sampling efficiency and testing efficiency on real-world deployed products as well as generated configurations and compare the results to a state-of-the-art t-wise sampling technique.

2 Background

In this chapter, we briefly introduce the concepts of *configurable systems*, *feature interactions*, and *t-wise sampling*.

2.1 Feature Models and Configurations

A *feature* is a functionality that is contained in a product. A *product line* might have some base and some variable features that can be in- or excluded in a product [5, 16]. This way, a user can individually decide, which features their product should contain. We use *feature models* (FMs) to describe possible feature combinations [8].

Definition 2.1 (Feature Model). A feature model is a tuple FM = (F, R) with $F = \{f_1, ..., f_n\}$ being a non-empty set of n features and R being a set of constraints restricting possible combinations of features in a product configuration.

Feature models are often represented in a tree-like hierarchy by a *feature diagram* [45]. In Figure 1, we see a feature diagram, with `L-W` = Login-Warning and `2-F-A` = 2-Factor-Authentication. The feature model represents a product line of a social network platform. A user has to choose between a `Public Account` or a `Private Account`. They can also decide to include `Security`, where they choose to include `L-W`, `2-F-A`, or both. When choosing a `Private Account`, `2-F-A` has to be selected as well. `Ads` are always included.

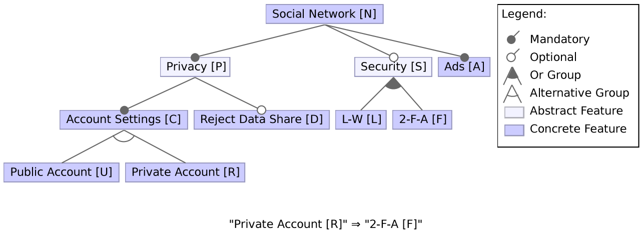


Figure 2: FM Running Example - Before Evolution Step

A *configuration* represents a selection of features from a feature model and can be used to derive a product of the product line. We define a configuration $C = \{l_1, ..., l_n\}$ as a set of literals over a set of features F, where a literal is positive, if the respective feature is selected, and negative if it is deselected. A feature can only be referred to by one literal for each configuration (i.e., $\forall f \in F : \{f, \neg f\} \not\subset C$). A configuration is *valid*, if it does not violate the constraints defined in the feature model. It is called *complete*, if it contains a literal for every feature (i.e., $|F| = |C|$). Otherwise, it is called *partial.* We always refer to a complete, valid configuration, if not stated otherwise. The *valid configuration space* is the set of all configurations that are valid for a given feature model.

2.1.1 *Product-Line Evolution.*

With product-line development, multiple factors have to be considered in evolution. Feature-model evolution refers to changes that are made in the feature model resulting in different feature model versions [13, 31, 34]. These changes can be, for example, features being added or removed. In Figure 2, we see the feature model from Figure 1 at an earlier point in time. As we see, two features (i.e., `Account Settings` and `Reject Data Sharing`) from the old model have been deleted in Figure 1. Evolution can also refer to changes in the underlying implementation which are not necessarily displayed in the underlying feature model.

2.1.2 *Field Configurations.*

A field configuration is a configuration describing a deployed product, which is based on a version of a feature model. Deployed products can receive updates, which can be in the form of security updates, or updating and adding functionality. Additionally, with updates, it is possible to keep software features of field configurations *stable* (i.e., updates ensure no difference in the implementation of the same feature in different field configurations). However, when a feature is not supported anymore, a field configuration is not valid on the current feature model any more even when its features are kept stable.

2.2 T-Wise Sampling

Examining all configurations of a product line is typically infeasible due to the exponential growth in configurations with an increasing number of features [6, 46]. Thus, in product-line testing, we often use a small, yet representative subset of all possible configurations

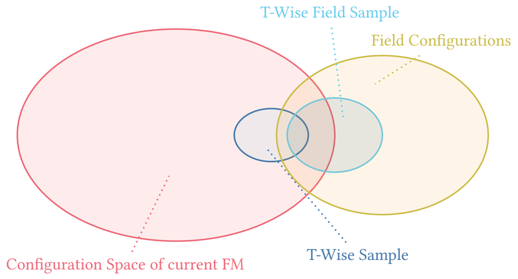


Figure 3: Venn diagram showcasing relationship between feature-model sample and field configurations

	R	¬R	U	¬U	L	¬L	F	¬F	D
C	🟨	⬜	⬜	🟨	⬜	🟨	🟨	⬜	🟨
D	🟨	⬜	⬜	🟨	⬜	🟨	🟨	⬜	
¬F	⬜	🟩	🟩	⬜	🟥	🟩			
F	🟩	🟩	🟩	🟩	🟩	🟩			
¬L	🟩	🟩	🟩	🟩					
L	🟥	🟩	🟩	🟥					
¬U	🟩	⬜							
U	⬜	🟩							

Table 1: Possible one- and pair-wise interactions. 🟥: in the feature model and *not* in field configurations; 🟨: in field configurations and *not* in the feature model; 🟩: in both

(i.e., a sample) [29]. With t-wise sampling algorithms, we generate a sample in which every valid t-wise feature interaction is contained in at least one configuration [29, 35, 42]. A t-wise feature interaction is the emergent behavior caused by t features [1, 19, 30, 36, 37]. For instance, there are four possible combinations for $t = 2$ (i.e., both features are selected, either one is selected, or neither selected) [5, 22]. We call a t-wise feature interaction valid, if (de-)selecting the t features forms a valid partial configuration. If not stated otherwise, we consider an interaction to be valid. In this paper, we disregard core and dead features for the t-wise coverage [11].

3 Problem Statement

In general, testing every product that is potentially affected from an update is infeasible. Consequentially, t-wise sampling is proposed to achieve test coverage over the interaction of features. In this section, we demonstrate, with a running example, the importance of taking field configurations into account for t-wise sampling by discussing three shortcomings of the current state of the art.

Missing Relevant Interactions. Simply using an existing t-wise sampling technique on the last feature model to create a sample for field configurations does not necessarily cover all important interactions. Field configurations have been derived from different versions of the feature model. As Figure 3 illustrates, the interactions that appear in the field configurations intersect with the configuration space of the current feature model but is not a subset. Therefore, a t-wise sample of the current feature model may only partially cover interactions of the field configurations.

Consider the following field configurations representing deployed products as an example. FC_1 to FC_6 are built based on the version of the feature model in Figure 1 and FC_7 on the version of the feature model in Figure 2:

FC_1 = {N, P, -R, U, -S, -L, -F, A} FC_2 = {N, P, R, -U, S, -L, F, A}
FC_3 = {N, P, -R, U, S, L, F, A} FC_4 = {N, P, -R, U, S, -L, F, A}
FC_5 = {N, P, -R, U, -S, -L, -F, A} FC_6 = {N, P, -R, U, S, -L, F, A}
FC_7 = {N, P, C, R, D, -U, S, -L, F, A}

Table 1 presents which one-wise and pair-wise interactions are covered by the pair-wise sample of the last feature model and the field configurations. As the yellow blocks show, overall nine pair-wise interactions do appear in the deployed configurations but are not covered by a pair-wise sample over the last model. Hence, creating a sample on the latest feature model misses nine interactions which then would not get tested for unwanted behavior under the update.

Covering Unused Interactions. As Figure 3 suggests, a t-wise sample on the last feature model also considers interactions that are not part of any field configuration. Covering the additional interactions may require testing more configurations even though we would not address any additional interactions from deployed configurations. As the red blocks in Table 1 indicate, the sample from the last feature model contains three interactions that are not represented in any deployed product (e.g., interaction R and L).

Covering Evolution. Typically, deployed configurations were derived from different points in time and, thus, possibly from different feature-model versions, such as Figure 2 and Figure 1. One way to address this evolution with available sampling algorithms is creating t-wise samples for every evolution step. However, t-wise sampling can result in large samples [49] even for a single feature model. Adding up all t-wise samples from all feature models leads to unnecessary many configurations. In our example, this would result in testing eleven configurations (i.e., five from Figure 1 and six from Figure 2). Additionally, as discussions with experts from the industry revealed, past feature-model versions are not necessarily available. Hence, requiring all prior model versions as input may not be applicable.

However, it is possible to achieve a pair-wise coverage of the example field configurations only by testing FC_1, FC_3, and FC_7. In the next sections, we introduce and evaluate a strategy on how to find such a suitable subset and cope with the problems of the low effectiveness and efficiency of existing t-wise sampling techniques with regard to field configurations.

4 Sampling on Field Configurations

In this section, we introduce the first t-wise sampling technique targeting field configurations. With our algorithm, we aim to reduce an existing set of field configurations to a subset that contains the same t-wise interactions. To this end, our goal is to keep the sample resulting from our technique as small as possible.

4.1 Field Coverage

Before introducing our algorithm, we formally define the coverage of t-wise interactions appearing in field configurations.

Definition 4.1 (T-Wise Field Sample). A t-wise field sample FS^t_{FC} is a set of configurations covering every potential t-wise interaction of a given set of field configurations FC.

Definition 4.2 (T-Wise Field Coverage). Let FC be the field configurations of a product line. The t-wise field coverage $FCov^t_{FC}(FS)$ of a t-wise field sample FS describes the ratio of the t-wise interactions that appear in FC (i.e., I^t_{FC}) and in FS (i.e., I^t_{FS}).

$$FCov^t_{FC}(FS) = \frac{|I^t_{FS} \cap I^t_{FC}|}{|I^t_{FC}|}$$

In the following, we propose a greedy-based strategy for creating a t-wise field sample by reducing a set of field configurations.

4.2 Generic Procedure

Now, we introduce the core idea of our algorithm. We aim to solve an instance of the minimum set cover problem, meaning we reduce the set of field configurations to a subset instead of creating new configurations as usual in t-wise sampling. Building a sample solely consisting of existing deployed configurations comes with multiple advantages. First, a configuration which results from sampling on a feature model might fail to cover some interactions, because specific features are not supported in the respective model, as shown in Section 3. Second, when using existing field configurations for t-wise sampling, a company can test on the deployed product or, if available, a digital twin. This way, they save the expense of building new configurations. Third, it is also an advantage to run a test on actual deployed products, because conventionally created samples may contain configurations that are never used in practice.

As input, the algorithm starts with a set of field configurations FC. To build the field sample, we start with an empty set FS^t_{FC}. First, we count for each interaction how often it appears and add all configurations containing unique interactions to our final sample. Second, we add, one after the other, a field configuration from the input sample to FS^t_{FC}, but only if the field configurations bring any new interactions to FS^t_{FC}. After each addition, we check whether 100% coverage is reached (i.e., $FCov^t_{FC}(FS^t_{FC}) = 1$). If that is the case, we stop adding more field configurations and return FS^t_{FC}. Picking promising configurations, w.r.t. effectively reducing sample size, is vital for the effectiveness of the algorithm. We discuss strategies for selecting the next configuration to add in the next sections.

We present an example where we create a pair-wise sample for the seven field configurations introduced in Section 3 ($FC_1 - FC_7$). For this example, we assume that the algorithm randomly chooses the configurations in their order, i.e., 1, 2, 3, 4, 5, 6, 7. Consequentially, Field Configuration 1 is added to FS^t_{FC} first. Then, Field Configuration 2 is added and the addition is valid because new interactions where added (e.g., {R, S}), increasing the t-wise field coverage of FS^t_{FC}. The same applies to Field Configuration 3, which also contains uncovered interactions (e.g., {-R, L}). Afterwards, Field Configurations 4, 5, and 6 are skipped, as all appearing interactions are already covered. Finally, Field Configuration 7 is added and the field sample reaches the desired t-wise field coverage.

As this example demonstrates, the resulting sample size relies on the order in which the configurations are added to FS^t_{FC}. In the example from above, we end up with a field sample containing four configurations (i.e., 1, 2, 3, and 7). However, pair-wise coverage can also be achieved with the smaller sample 1, 3, and 7. To prevent the algorithm from adding Field Configuration 2, Field Configuration 7 would have to be added first. To determine a way of adding the configurations in an order that potentially leads to a smaller and, thus, more efficient sample, we present two strategies for configuration selection in the next two subsections.

4.3 Reduction to Minimum Set Cover Problem

We can specify our problem setting as a minimum set cover problem. In the minimum set cover problem, we have a set X of n elements (X = $\{X_1, ..., X_n\}$) where each X_i contains m_i elements ($X_i = \{x_1, ..., x_{m_i}\}$) and search for a subset of X that contains every element contained in any X_i. Mapping it to our problem, the set of field configurations is X and the elements in X_i are the interactions between the features.

The specification as set cover problems yields the problem that we need to explicitly list and consider all interactions appearing in every configuration with conventional set cover techniques. With product lines in practice containing thousands of features [24, 28, 34] and, thus, potentially millions of interactions, considering every configuration as a set of appearing interactions instead of a set of features may induce scalability issues. To deal with the large numbers of interactions we apply the idea of a lazy greedy approach for the set cover problem [32] and adapt it to our technique. The core idea is:

(1) Track for every field configuration how many of its interactions are not yet covered by the final sample.
(2) Add the field configuration with the most uncovered interactions next.

We also integrated our strategy of finding unique interactions and adding the configurations containing them to the final sample before starting with the algorithm. Here, we only consider subsets of the overall interactions at a given time during our procedure. We present the approach and an optimization based on prioritizing promising configurations in Section 4.4.

4.4 Configuration Scores

To further specify the selection of the next configuration, we propose to use a scoring function to determine which configuration is added next to the field sample FS^t_{FC}. We aim to add configurations in a way that we reach a t-wise field coverage of 1 while trying to keep the sample size low. Hence, we prioritize configurations that cover interactions that occur in only few configurations by giving the configurations a score, which depends on the number of appearances of its interactions. The steps for the configuration selection are:

(1) we compute a score for each configuration with the formula shown in Equation 1 with c being a configuration, I being the t-wise interactions that appear in the configuration, and $counter_i$ being the number of appearances of that interaction in the remaining field configurations $FC \setminus FS^t_{FC}$,
(2) add the configuration with the highest score to our final sample,
(3) recalculate the scores in a way that interactions which are now covered by FS^t_{FC} are no longer considered for the score and repeat step 2 until every score is zero.

$$score_c = \sum_{i \in I_t} \frac{1}{counter_i} \tag{1}$$

	R	¬R	U	¬U	L	¬L	F	¬F	D
C	1			1		1	1		1
D	1			1			1		
¬F		2	2			2			
F	2	3	3	2	1	4			
¬L	2	4	4	2					
L		1	1	2					
¬U	2								
U		5							

Table 2: Count of interactions in field configurations

With our approach to calculate the score and prioritize configurations with high scores, we prioritize configurations containing more rare appearing interactions. We assume this leads to the algorithm quickly covering the rare interactions while covering interactions that often appear per default. When prioritizing often appearing interactions, we would end with multiple rare interactions for which we would need to add multiple configurations because the chance of them appearing in the same ones is low. This way, the sample size would be unnecessarily increased.

Again, we take our seven field configurations from above and walk through the algorithm to demonstrate how the scoring function works. In Table 2, we see for each interaction how often it appears amongst the field configurations. For readability, we leave out the features N, P, and A, since they appear in every field configuration as selected. The unique interactions are marked in orange.

First, the algorithm adds every configuration containing a unique interaction to FS. This results in adding the field configurations FC_3 and FC_7. Afterwards, the algorithm calculates the scores of the remaining configurations, only considering the interactions that are not covered by one of the two already added configurations. The left interactions are marked with a blue color in Table 2. This leads to the remaining field configurations receiving the following scores:

FC_1: score $= 7 \times \frac{1}{2} + 2 \times \frac{1}{4} = 4$ FC_2: score $= 0$

FC_4: score $= 2 \times \frac{1}{4} = 0.5$ FC_5: score $= 7 \times \frac{1}{2} + 2 \times \frac{1}{4} = 4$

FC_6 = score $= 2 \times \frac{1}{4} = 0.5$

Two field configurations have the same highest score (i.e., FC_1 and FC_5). In this case, the algorithm chooses to add one of them, e.g., FC_1. After the addition and the recalculation of every score, we remove the interactions that are now covered by FS from the equation. Now, every left configuration (i.e., FC_2, FC_4, FC_5, and FC_6) results with a score of zero because every interaction is now covered. This means, our algorithm returns the calculated sample, containing the field configurations FC_1, FC_3, and FC_7, and terminates.

4.5 Combined Approach with T-Wise Sampling

In this paper, we demonstrate the importance of being able to cover and test interactions in the field before running an update. We also show how existing t-wise testing techniques are not efficient to achieve t-wise field coverage when we have an evolving product line. Nevertheless, covering the interactions of potential future configurations by creating a t-wise sample on the current feature model is still beneficial in product-line development. The idea is not only considering interactions in already sold products but also products potentially sold in the future. For this reason, we include an optional step in our approach, aiming to create a unified sample for both, the field configurations and the latest feature model. The goal is to reduce the sample size compared to testing two samples separately while achieving the same coverage. Because some of the interactions are contained in both samples, we aim to remove configurations only containing duplicate interactions. Besides applying the algorithms on field configurations, the idea is to apply it on a combined sample. This sample consists of the set of field configurations and a t-wise sample for the feature model.

5 Experimental Evaluation

We performed an empirical evaluation comparing different reduction strategies. In detail, we compare three algorithms for sample reduction, *Set-Cover*, which uses a simple greedy heuristic for solving the Minimum Set Cover Problem [32] (cf. Section 4.3), *Scoring*, which uses a weighted greedy heuristic (cf. Section 4.4), and *Random*, which uses a random heuristic. In addition, we include *YASA* [26], a state-of-the-art t-wise sampling algorithm, as a baseline. We are interested in the scalability of the algorithms as well as the size and coverage of the computed samples. In particular, we aim to answer the following research questions:

RQ_1 What is the efficiency of *Scoring* and *Set-Cover*?

$RQ_{1.1}$ How much time is required by the different algorithms?

$RQ_{1.2}$ How large are the samples of the different algorithms?

RQ_2 What is the effectiveness of *Scoring* and *Set-Cover* in terms of t-wise interaction coverage and t-wise field coverage?

To answer RQ_1, we measure and compare the time to create a sample for all algorithms for different systems with differing numbers of field configurations. Further, we compare the size of the samples created with each algorithm.

For RQ_2, we measure the coverage of all samples created for the two metrics *t-wise interaction coverage* and *t-wise field coverage* (cf. Section 4) with $t = 1$ and $t = 2$, respectively.

5.1 Experiment Design

5.1.1 Data. We use three data sets from different sources, each containing multiple systems with a set of field configurations. In Table 3, we show an overview of all systems, including the number of features, the number of field configurations, and the number of feature-model versions per system. For each dataset, we also evaluated on the combined sample as introduced in Section 4.5.

The first data set[1] consists of real-world systems using the Kconfig language, collected by Pett et al. [44]. Although these systems each include a wide range of feature-model versions over time, there is no data on field configurations for these feature models. Thus, for each feature-model version, we use a randomly created sample by Pett et al. [44].

The second data set[2], provided by Pereira et al. [40], encompasses two real-world case studies, containing one feature-model version and a set of real field configurations for each system.

The third data set contains five systems from the automotive domain, provided by our industry partner. This data set contains multiple feature-model versions and real configurations over these

[1] https://github.com/orgs/TUBS-ISF/repositories?q=-case-study

[2] https://wwwiti.cs.uni-magdeburg.de/~jualves/PROFilE/

System Name	Source	#Versions	#Feat.	#Field Config.
Soletta	[44]	173	457	11,950
Toybox	[44]	62	334	1,039
Busybox	[44]	454	1,050	31,279
Fiasco	[44]	33	258	10,872
Uclibc	[44]	177	272	110,524
ERP	[40]	1	1,920	61
Agrib	[40]	1	2,238	5,749
Automotive08	Industry	unknown	221	233,309
Automotive09	Industry	unknown	279	247,780
Automotive10	Industry	unknown	279	45,387
Automotive11	Industry	unknown	218	27,808
Automotive12	Industry	unknown	147	129

Table 3: Systems used in the evaluation.

models. However, due to obfuscated feature names in the feature models and configurations, used to protect company secrets, we are not able to create a consistent mapping between features in the feature-model version and field configurations making the application of the combined approach unreasonable.

5.1.2 Process. Our experiments encompass multiple steps per system. First, since the data sets were created by different people, we convert feature models and field configurations from each system into a unified format. Second, we create a one-wise and a pair-wise interaction sample for the latest feature-model version of each system using YASA. Third, we create two new samples by combining the field configurations with the one-wise and the pair-wise sample, respectively. Fourth, we apply the reduction algorithms to each sample created so far using $t = 1$ and $t = 2$ as input for each algorithm, which creates six reduced samples for each existing sample. We repeat the execution of *Random* three times for each sample using different random seeds. For all measurements on randomly reduced samples, we use the median value. Finally, we compute the coverage of each sample using the two metrics *t-wise interaction coverage* and *t-wise field coverage* with $t = 1$ and $t = 2$, respectively. In addition, we measure the time required for Step 2 and Step 4.

5.1.3 Technical Setup. The evaluation was run on a server with two Intel(R) Xeon(R) CPU E5-2620v3 2.4 GHz and 256 GB RAM under Ubuntu 22.04. The time measurement was done in Java with `System.nanoTime()`. The YASA version that was used in our implementation is based on the last commit from April 6th, 2024.[3]

5.1.4 Data Availability. For reproducibility, we published an artifact containing our implementation of all algorithms described in Section 4, our evaluation setup, used data sets, and all raw data created by our experiments.[4] Note that we had to exclude data set three for reasons of confidentiality.

5.2 Results

We distinguish the results of our evaluation between efficiency and effectiveness measurements.

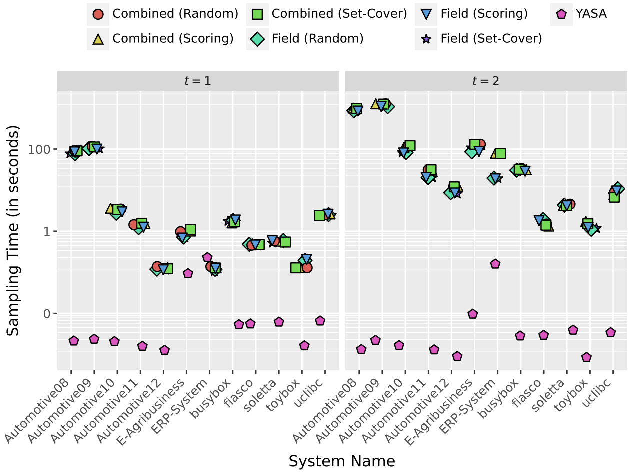


Figure 4: Sampling time per system and sampling algorithm

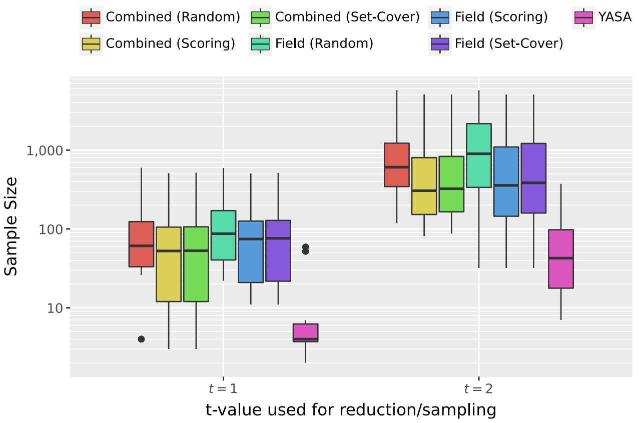


Figure 5: Sample size per sampling algorithm

Efficiency. In Figure 4, we show the time each algorithm needed to finish on a logarithmic scale for each system. The left plot presents the results for creating a one-wise sample and the right plot for the pair-wise sample. For every calculation, except ERP-System with $t = 1$, YASA was the fastest algorithm. Comparing the reduction of the samples, all algorithms need approximately the same time for $t = 1$ as well as $t = 2$. An exception is the feature model ERP-System, where reducing the combined sample for $t = 2$ needs noticeably more time than reducing only the field sample. Note that the displayed time of the combined sample only shows the calculation time of the reduction step. In order to create the combined sample, the YASA sample has to be created first, which requires additional computation time, as shown in the plots.

Figure 5 depicts the size (i.e., the number of configurations) of all samples. We show the sample size for all algorithms using a logarithmic scale. The boxes on the left-hand side indicate the size for the calculations of $t = 1$ and the boxes on the right-hand side for $t = 2$. We see that YASA computes much smaller samples than any other algorithm. Comparing the reduction algorithms with each

[3] https://github.com/FeatureIDE/FeatJAR-formula-analysis-sat4j/commit/20d4e7dead43ef908ff772892c30f2658ea6bdbf
[4] https://github.com/skrieter/rp-evaluation-sample-reducer

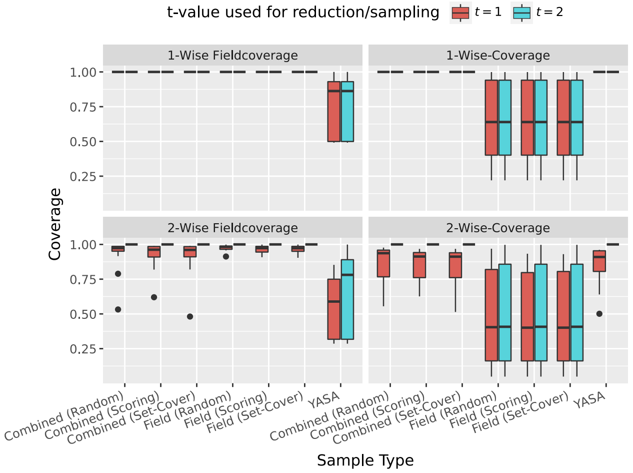


Figure 6: Coverage values achieved by all samples for each sampling algorithm and coverage metric

other, only *Random* performs worse than the other two for $t = 1$ and $t = 2$ on the combined as well as the normal sample. Comparing these sample sizes with the number of field configurations that were given as input (cf. Table 3), we are able to reduce the number of configurations by several orders of magnitude.

Effectiveness. In Figure 6, we display the coverage of all sampling algorithms. For each algorithm, we show a boxplot over all systems containing the coverage value for a specific coverage metric. In detail, we compare the reduction algorithms and YASA with $t = 1$ and $t = 2$, respectively. For each computed sample we display the one-wise as well as pair-wise coverage to inspect how many of the pair-wise interactions are covered by the one-wise sample and vice versa. We show values for four different metrics, t-wise field coverage (cf. Section 4) and t-wise interaction coverage with $t = 1$ and $t = 2$, respectively. While the reduction algorithms all reach a 100% t-wise field coverage if a sufficient t-value is chosen, they may cover less than a quarter of the t-wise interactions of the last feature model. Similarly, YASA is always able to achieve 100% t-wise interaction coverage, given a sufficient t-value, but on average stays below 80% of field coverage. Comparing the different reduction algorithms with each other, all of them reach similar values over all coverage metrics. Calculating a sample for the combined configurations achieves 100% coverage for all metrics given a sufficient value for t.

5.3 Discussion

Regarding $RQ_{1.1}$, we measured that YASA is always the fastest algorithm except for ERP, with the least amount of field configurations. Contrary to our expectations that the random heuristic would outperform the proposed algorithms because of its simplicity, all algorithms need approximately the same time for each calculation. We also measured no significant efficiency loss when reducing on the combined sample compared to reducing on the field configurations. This comes from the fact that the YASA samples are already very small and, thus, adding the t-wise sample to the field configurations only marginally increases the number of configurations. We conclude that the algorithms do not significantly differ in terms of computation time. Also, the combined sample can be computed with as much computational effort as the sample for field configurations, excluding the creation of the YASA sample. While we were not able to compute a reduced sample within 12 hours for Automotive08 and Automotive09 with t=2, we conclude that for all other systems in our evaluation the scoring algorithm and the combined approaches scale quite well.

Regarding $RQ_{1.2}$, we see similar results to the computational effort. YASA always produces the smallest samples. As expected, using the scoring function to determine configurations that are added next, results in a smaller sample size than just random addition. However, using the scoring function does not outperform the Set Cover algorithm we adapted from the literature. Thus, we conclude that it is better to use a strategy other than randomly selecting configurations and existing set cover approaches serve as a good base for the purpose of reducing configuration sets. In addition, we find that the sample resulting from the combined approach has a similar sample size than samples resulting from the field configurations.

Finally, answering RQ_2, we are able to confirm our assumption about existing sampling techniques not being sufficient to cover interactions appearing in field configurations. The other way around, calculating a sample for field configurations is not able to cover interactions of the last feature model, since it only covers those interactions that have been chosen by a customer. Both samples (i.e., the one for the last feature model and the one for the field configurations) are not able to result in a sample that gives a reliable coverage for the use case they are not made for. To this end, we also created the combined sample, which covers the interactions of the feature model and the interactions of the field configurations without loss in efficiency, as shown above.

In summary, the evaluation shows that existing sampling techniques are more efficient in terms of sample size and sampling time than our sampling on field configurations, but are not able to cover all interactions in the field. We show that it is promising to develop a more advanced technique to decrease the sample size by a large amount compared to a baseline of randomly selecting configurations. In addition, we see that using a combined approach which covers both, all t-wise field interactions and all t-wise interaction from the latest feature model is about as efficient as using the algorithms only on the field configurations.

5.4 Threats to Validity

Internal Validity. Each data set used in our evaluation had certain limitations we needed to consider. First, for data set one [44], only random samples instead of real-world field configurations were available. Furthermore, all samples contained the same number of configurations. Second, data set two [40] only contained a single version for each system. Third, due to obfuscated feature names in the industry data, we could not directly compare the number of interactions in the field configurations with those from the corresponding feature model, which also prevents us from using the combined approach on these systems. While each data set has its own limitations, we argue that by combining all three, we can still draw meaningful conclusions on the evaluated algorithms.

Regarding our implementation, we cannot guarantee correctness. We use the same base implementation for all reduction algorithms to enable a fair comparison between the evaluation strategies and use unit testing to mitigate the risk of bugs.

External Validity. For comparison with an existing sampling technique, we only considered YASA. However, we argue that YASA is a well established tool, which is representative for sampling techniques using greedy strategies. Furthermore, this comparison only serves to contextualize the sample sizes and computation time of t-wise field sampling to classical t-wise sampling approaches, which follow slightly different optimization goals.

In our algorithm, we assume that features with the same name have the same underlying implementation. That is not necessarily the case in real-world product lines. This assumption limits our approach to systems, where features are kept stable as long as they are supported in the current feature model. We assume that numerous companies synchronize software assets of field configurations with updates in the case of implementation changes. Furthermore, we argue that typically implementations of features are not completely replaced, but get modified, which means that our technique can still be useful in these cases.

6 Related Work

There have already been publications introducing different testing algorithms for product lines [2, 14, 15, 29, 33, 41] as well as different sampling techniques [7, 23, 25, 27, 35, 42]. However, none of the existing work considers updates on deployed configurations containing features which are not supported anymore. As we have shown in this paper, we do need to consider every deployed configuration for the use case of such an update. It follows that none of the existing work on sampling algorithms is sufficient to solve the problem of covering interactions in field configurations.

More similar to our work are existing sampling techniques including the consideration of configurations. Oster et al. [39] introduced an algorithm which allows involving existing products to the sampling algorithm. However, their algorithm is restricted to products for one feature model version and only developed for pair-wise interactions. Bombarda et al. [12] support the reuse of configurations from test suites of old feature models. These configurations are used for the creation of tests of new test suites after a feature model evolved. Contrary to our goal, they do not consider any subset of valid interactions but only aim to achieve t-wise coverage of the feature models. Al-Hajjaji et al. [3] take an already existing sample and apply configuration prioritization on it. The way they chose to calculate the prioritization of the configurations is similar to the scoring function we use in our algorithm. Anyway, additionally to some differences in the calculation, their goal is reordering a t-wise sample and prioritize by difference between configurations while our goal was to build a subset and prioritize by rare interactions.

We also consider publications that specifically investigate the impact of evolution on configurations as a similar research field of our work. Pett et al. [43] proposed continuous t-wise coverage, where they consider previous tested t-wise interactions of old feature models to reduce the interactions that have to be covered after an evolution step. Different to ours, their goal is achieving t-wise coverage of a specific feature model and only contains inclusion of old t-wise interactions that are still valid on the current model. Heider et al. [21] consider derived products in their work and analyze the impact of changes in the variability model on those products. Similar as Nieke et al. [38], who locks configurations to ensure them staying valid even if a feature model evolves, they have the goal to investigate the validity of the configurations regarding the newest feature model. Both works do not consider configurations that get an update receiving new functionality while only being valid on an old variability model. Demuth et al. [17] optimize existing products, including the evolution of the underlying software product line. However, the optimization does not include coverage of t-wise interactions of the products.

7 Conclusion and Future Work

In this paper, we introduce *t-wise field sampling*, a sampling metric and corresponding algorithm on a set of field configurations (i.e., configurations that have been deployed) for evolving software product lines. We enable covering t-wise interactions contained in a set of configurations without the need of having a corresponding feature model. To this end, we propose two algorithms, which adapted an existing Minimum Set Cover algorithm based on a greedy strategy and find a solution to cover interactions of field configurations. In addition, we propose an approach combining t-wise field sampling and classical t-wise interaction sampling to achieve an even better interaction coverage. We evaluate all algorithms in terms of their efficiency and effectiveness by comparing them to a state-of-the-art t-wise sampling technique.

Our evaluation shows that existing sampling techniques are more efficient than sampling on field configurations. However, existing sampling techniques also lack in covering interactions in the field. This disqualifies them for the use case of testing field configurations that have been build on different feature models. The evaluation also shows that our algorithm is able to build a sample that covers field configurations as well as potential future combinations from the last feature model with a similar efficiency than our t-wise field sampling approach.

In future work, we aim to optimize our current algorithm and to evaluate different variants of our greedy algorithm's scoring function. This way, we examine whether we can further decrease the sampling time and sample size. Also, we plan to include filtering of interactions in our algorithm. With the filtering, a user can decide to only cover specific interactions further reducing the size of the final sample. Another optimization would be possible if updates can be directly mapped to one or multiple features. In this case, only interactions with the updated features would have to be considered.

Acknowledgments

We want to thank Chico Sundermann for his support and many helpful discussions. We also want to thank Paul Bittner for his patience and help before the deadline.

References

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A

Qualitative Analysis. *Trans. on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 10:1–10:34.

[2] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730.

[3] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Software and System Modeling (SoSyM)* 18, 1 (2019), 499–521.

[4] Simon Alvarez. 2022. Lucid Air briefly “bricked” after failed over-the-air software update. https://www.teslarati.com/lucid-air-bricked-after-failed-ota-update/.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer.

[6] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions Using Feature-Aware Verification. In *Proc. Int’l Conf. on Automated Software Engineering (ASE).* IEEE, 372–375.

[7] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: An Adaptive Weighted Sampling Approach for Improved t-Wise Coverage. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM, 1114–1126.

[8] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* Springer, 7–20.

[9] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[10] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int’l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 7:1–7:8.

[11] Sabrina Boehm, Tim Schmidt, Sebastian Krieter, Tobias Pett, Thomas Thüm, and Malte Lochau. 2025. Coverage Metrics for T-Wise Feature Interactions. In *Proc. Int’l Conf. on Software Testing, Verification and Validation (ICST).* IEEE, Washington, DC, USA. To appear.

[12] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. 2023. On the Reuse of Existing Configurations for Testing Evolving Feature Models. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B.* 67–76.

[13] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of Software Product Lines. In *Evolving Software Systems.* Springer, 265–295.

[14] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 56, 10 (2014), 1183–1199.

[15] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2006. Coverage and Adequacy in Software Product Line Testing. In *Proc. Int’l Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA).* ACM, 53–63.

[16] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley.

[17] Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. 2014. Automatic and incremental product optimization for software product lines. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation.* IEEE, 31–40.

[18] Emelie Engström and Per Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *J. Information and Software Technology (IST)* 53 (2011), 2–13.

[19] B.J. Garvin and M.B. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *Proc. Int’l Symposium on Software Reliability Engineering (ISSRE).* 90–99.

[20] Jordan Golson. 2016. Many Lexus navigation systems bricked by over-the-air software update. https://www.theverge.com/2016/6/7/11879860/lexus-navigation-broken-software-update-bug.

[21] Wolfgang Heider, Rick Rabiser, Paul Grünbacher, and Daniela Lettner. 2012. Using Regression Testing to Analyze the Impact of Changes to Variability Models on Products. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* ACM, 196–205.

[22] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int’l Conf. on Model Driven Engineering Languages and Systems (MODELS).* Springer, 638–652.

[23] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays From Large Feature Models. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* ACM, 46–55.

[24] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM, 291–302.

[25] Sebastian Krieter. 2020. Large-Scale T-Wise Interaction Sampling Using YASA. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* ACM, 29:1–29:4.

[26] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *Proc. Int’l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM.

[27] Dominik Michael Krupke, Ahmad Moradi, Michael Perk, Phillip Keldenich, Gabriel Gehrke, Sebastian Krieter, Thomas Thüm, and Sándor Fekete. 2025. How Low Can We Go? Minimizing Interaction Samples for Configurable Systems. *Trans. on Software Engineering and Methodology (TOSEM)* (2025). To appear.

[28] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int’l Workshop on Logics for Component Configuration (LoCoCo).* Open Publishing Association, 44–53.

[29] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (1st ed.). Chapman & Hall/CRC.

[30] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. on Software Engineering (TSE)* 30, 6 (2004), 418–421.

[31] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming (SCP)* 78, 8 (2013), 1010–1034.

[32] Ching Lih Lim, Alistair Moffat, and Anthony Wirth. 2014. Lazy and eager approaches for the set cover problem. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference-Volume 147.* 19–27.

[33] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proc. Int’l Workshop on Combinatorial Testing (IWCT).* IEEE, 1–10.

[34] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* Springer, 136–150.

[35] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int’l Conf. on Software Engineering (ICSE).* ACM, 643–654.

[36] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proc. Int’l Conf. on Automated Software Engineering (ASE).* ACM, 483–494.

[37] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11:1–11:29.

[38] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proc. Int’l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 73–80.

[39] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* Springer, 196–210.

[40] Juliana Alves Pereira, Matuszyk Pawel, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2018. Personalized Recommender Systems for Product-Line Configuration Processes. *Comput. Lang. Syst. Struct.* 54 (2018), 451–471.

[41] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2012. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal (SQJ)* 20, 3-4 (2012), 605–643.

[42] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int’l Conf. on Software Testing, Verification and Validation (ICST).* IEEE, 459–468.

[43] Tobias Pett, Tobias Heß, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2023. Continuous T-Wise Coverage. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* ACM, 87–98.

[44] Tobias Pett, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. AutoSMP: An Evaluation Platform for Sampling Algorithms. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* ACM, 41–44.

[45] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *Proc. Int’l Conf. on Requirements Engineering (RE).* IEEE, 136–145.

[46] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering (EMSE)* 28 (2023).

[47] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *Proc. Int’l Conf. on Software Engineering (ICSE).* IEEE, 254–264.

[48] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer.

[49] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int’l Systems and Software Product Line Conf. (SPLC).* ACM, 1–13.

Asking Security Practitioners: Did You Find the Vulnerable (Mis)Configuration?

Richard May
Harz University of Applied Sciences
Wernigerode, Germany
rmay@hs-harz.de

Christian Biermann
msg services gmbh, Harz University of Applied Sciences
Hamburg, Germany
christian.biermann@msg.group

Jacob Krüger
Eindhoven University of Technology
Eindhoven, Netherlands
j.kruger@tue.nl

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

Abstract

With ever evolving software, reliability and quality assurance are facing growing complexity and security issues. Particularly, interconnected and configurable systems are threatened by (mis)configurations that can lead to exploitable vulnerabilities. Unfortunately, there is limited information on how such configuration vulnerabilities occur or how practitioners deal with these. To tackle this gap, we investigated the connections between (mis)configurations, vulnerabilities, and their treatment by conducting a survey with 41 security practitioners who have encountered configuration vulnerabilities in their work. More precisely, our objectives were to understand the causes, prevalence, severity, and treatments of such vulnerabilities. We found that configuration vulnerabilities are prevalent and severe in practice. They primarily stem from dependency issues, outdated software, and inconsistent (cross-)configurations; are typically influenced by human errors; and are either identified during testing or, in the worst case, during deployment and operation. Generally, configuration vulnerabilities are detected due to security incidents or through word-of-mouth, implying that more preventive security management is required—ideally at an early stage and as part of a holistic security-engineering process. Overall, we aim to enhance the understanding of researchers and practitioners regarding current practices related to handling configuration vulnerabilities as well as open challenges.

CCS Concepts

• **Software and its engineering**; • **Security and privacy** → **Vulnerability management**; **Software security engineering**;

Keywords

variability, misconfiguration, vulnerability management, security, practitioners, survey, questionnaire

ACM Reference Format:
Richard May, Christian Biermann, Jacob Krüger, and Thomas Leich. 2025. Asking Security Practitioners: Did You Find the Vulnerable (Mis)Configuration?. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3715340.3715439

1 Introduction

Ensuring high-quality and reliable software systems is becoming more and more challenging [76], due to the increasing complexity of such systems [20]. In recent years, security has become one of the most important quality attributes related to system reliability [56, 61, 62]. Concurrently, growing configurability [21], interconnections and dependencies between different features and systems [55], as well as system evolution [42] have increasingly contributed to the emergence of misconfigurations, unwanted feature interactions, unexpected bugs, or even system failures [48, 67, 92]. So, misconfigurations tend to cause security vulnerabilities that can be exploited by malicious attackers, usually leading to major violations of a system's confidentiality, integrity, and availability [44, 45, 63].

Typically, security experts are hired to evaluate software systems and their configurations regarding potential risks for vulnerabilities and malicious exploitation (i.e., attack likelihood and impact) [36, 69]. For example, penetration tests are used to identify and assess specific vulnerabilities [19] or attack trees are modeled to trace and understand attack scenarios [43, 46]. Due to the rising number of vulnerabilities and exploits, the experiences gained and required by security experts is growing fast. Consequently, surveying such experts offers a valuable opportunity to synthesize and benefit from their lessons learned and best practices.

Using this opportunity, we asked security practitioners to share their experiences on vulnerabilities caused by configuring (i.e., *configuration vulnerabilities*). Our research goal was to **understand the connections between configurations and vulnerabilities, 🔍 the main causes of these vulnerabilities, ⚠ how prevalent and severe they are, and, 🛡 how they can be treated.** In more detail, we conducted an online questionnaire among 41 security practitioners who already experienced configuration vulnerabilities. There has already been research studying such vulnerabilities (cf. Section 7), for example, May et al. [55] analyzed developer discussions on vulnerable system configurations. Loureiro [48] studied

VaMoS 2025, Rennes, France

ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715439

misconfiguration prevention strategies and Dietrich et al. [22] analyzed the human aspects of vulnerable misconfigurations from an operators' perspective. However, to the best of our knowledge, there is currently no comparable study (i.e., thematic focus, sample size) focusing on practitioners' voices on main causes, prevalence, and common mitigation practices.

In detail, our contributions are:

- A comprehensive and practice-oriented overview of configuration vulnerabilities and their properties.
- A discussion of the main causes, severity, and common practices for effectively treating configuration vulnerabilities.
- An open-access repository containing our questionnaire and the anonymous answers of all participants to enable replications and validations of our study.[1]

Through our findings, we aim to shed light on the connections between (mis)configuring and vulnerabilities from a practical perspective. Our results are intended to help researchers and practitioners, especially in the domains of variability and security, learn from the experience of security experts. Overall, our work bridges the gap between theoretical research and real-world practice on this topic.

2 Background

Next, we introduce key background information, specifically system configuring and vulnerability management.

2.1 System Configuring

Modern software has become increasingly configurable to make systems and applications adaptable to specific use cases and associated stakeholder requirements [2, 7]. Typically, configurable systems rely on a set of features (i.e., user-visible functions) that can be enabled, disabled, and combined to create a variety of customized system variants [34, 71, 77]. System configuring may involve different types of configurations, oriented towards specific areas like software configuration (e.g., application settings) [77, 87], network configuration (e.g., routing protocols) [11, 32], storage configuration (e.g., cloud and database system access patterns) [13, 23, 54], server configuration (e.g., load balancer) [8, 30], and security configuration (e.g. security measures) [14, 16]. Furthermore, these configurations have numerous relations and interdependencies. For example, a security configuration typically influences all other types of configurations, which have to ensure compliance with security standards [78].

Developers usually implement system configurations using certain variability mechanisms, such as preprocessor directives [49, 82], dependency injections [80], or feature-oriented programming [7, 15]. Such mechanisms represent variability at the implementation level, while the organization of features, their relationships, and dependencies are usually documented via feature models—the defacto standard at a conceptual level [18, 66, 79]. From a more technical perspective, configuration files store the allowed configurations at the implementation level (e.g., specifying constraints) [11, 75]. Particularly constraints are essential to document, since certain features may depend on each other while other features cannot co-exist (i.e., they are mutually exclusive) [59, 88]. Referring to the latter, faulty configurations (i.e., misconfigurations) may lead to non-trivial problems, ranging from vulnerabilities over bugs to system failures that might cause fatal consequences (e.g., in safety-critical systems) [55, 56]. Accordingly, verifying system configurations is key to ensure reliable and secure systems [1, 72, 83].

2.2 Vulnerability Management

The reliability of modern software systems is increasingly linked to IT security [35, 61]. IT security refers to the practices and measures used to protect all types of software systems, comprising a variety of ways to ensure confidentiality, integrity, and availability [37, 69, 74]. Security measures or patterns that involve concrete strategies to protect systems (e.g., security policies) are typically oriented towards the mitigation of threats and risks. Threats are events with a potential negative impact on a system [36, 37, 69], such as unwanted feature interactions or configuration errors (i.e., misconfigurations) [92]. Such events are triggered mainly by certain unsecured system conditions, typically in the context of vulnerabilities (i.e., system weaknesses) [36, 37]. Although not every vulnerability is necessarily critical and may be considered as an acceptable risk, exploiting vulnerabilities (e.g., via SQL injections [33]) often results in violated security objectives and in the associated security risks [36, 69]. The actual risk is specified based on the likelihood and impact of potential exploits [36, 38], and is usually listed in incident databases (e.g., National Vulnerability Database, Artificial Intelligence Vulnerability Database) that provide scales to classify and rate vulnerabilities [43, 57, 60].

Detecting and managing vulnerabilities is complex and has become more difficult, due to the increasing number of features and their respective configuration options [67, 81, 90]. To prevent vulnerabilities, a systematic security-management process should be performed during development and maintenance [37, 68]. In this context, features, products, and entire product lines should be systematically tested (i.e., verified) to minimize the risk of security-related incidents [54, 83].

3 Methodology

In the following section, we describe our study's goal, research questions, methodology, and conduct (cf. Figure 1).

3.1 Goal and Research Questions

Our main goal was to understand the relations between configuring and vulnerabilities. This included identifying and discussing properties and practices related to detecting and treating vulnerabilities. To accomplish this goal, we formulated the following three Research Questions (RQs):

RQ$_1$ 🔍 **What are causes for configuration vulnerabilities?**
First, we aimed to identify the primary causes that lead to vulnerabilities in the context of configuring a software system. Specifically, our focus was on collecting insights on the triggers to find common patterns and highlight the most critical causes for configuration vulnerabilities.

RQ$_2$ ⚠ **How prevalent and severe are configuration vulnerabilities in the real world?**
Second, we aimed to determine the frequency and severity of configuration vulnerabilities as experienced by security experts. Our goal was to collect data on how often these

[1] https://doi.org/10.5281/zenodo.14007881

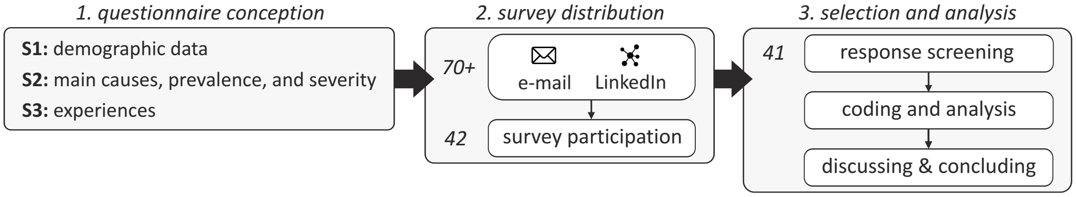


Figure 1: Overview of our research method, with numbers indicating the amount of practitioners involved.

vulnerabilities occur in practice and the impact they can have on systems and operations.

RQ$_3$ 🛡 What are common practices to treat and prevent configuration vulnerabilities?
Lastly, we aimed to explore and discuss strategies and practices employed by experts to address and prevent configuration vulnerabilities.

Through our work, we aim to provide a comprehensive overview of the connections between software configuring and vulnerabilities by building on real-world experiences of security experts. Through this overview, we contribute insights for practitioners and researchers that can help to increase their awareness for such vulnerabilities and related practices; providing a helpful means for them to safeguard their software systems and develop new techniques to prevent vulnerabilities.

3.2 Questionnaire Design

We developed our questionnaire based on established guidelines for conducting (online) questionnaires in software engineering [28, 64]. The first and second authors created the questionnaire using Microsoft Forms to build and host the online questionnaire. Iteratively, the third author independently reviewed the questionnaire two times. Besides a general introduction into the topic, we provided a description of how the participants' data was used as well as a consent form. Participants could only take part in the questionnaire if they gave their informed consent and voluntarily. We collected all data in a way that no conclusions could be drawn about the participants or their companies (i.e., anonymized data collection). Furthermore, all data was stored in an encrypted form on a server in Germany.

Overall, our questionnaire involved 20 questions in English and took about 10–15 minutes. The research questions served as basis for the questions and their ordering. In most cases, we used closed-ended questions requiring participants to choose specific items. For these, we typically provided an additional free-text option to add items not listed. Moreover, in some cases, we added the option not to answer a question (i.e., *prefer not to say*) in case participants may have had privacy concerns or constraints by their company. The closed-ended questions involved questions in which we explicitly asked for opinions based on Likert scales (e.g., ranging from *strongly agree* to *strongly disagree*) or number-based scales (e.g., ranging from *1–very rarely* to *10–very frequently*). In addition, we included questions to rank certain items according to their perceived relevance. We further relied on a few open-ended (i.e., free text) questions, typically to obtain more elaborate answers. All but one question were mandatory (i.e., one optional open-ended question for further thoughts on triggers).

Scoping. The target group of our study were IT security practitioners who already experienced configuration vulnerabilities in their work. Besides this background, we did not define any further restrictions. We combined three different distribution channels to promote our online questionnaire, aiming to reach as many participants as possible. These channels included 1) the personal networks of the authors, 2) the LinkedIn network of the first author, and 3) sharing of the invitation through interested participants (i.e., snowballing).

Structure. Our questionnaire involved 20 questions structured into three sections: *demographics* (6 questions), *main causes, prevalence, and severity* (9 questions), and *experiences* (5 questions).

Section 1 (demographics). In the first section, we asked for demographic data, i.e., questions on practical experience, employment, and company. We used this data to assess the responses (e.g., little versus much experience) and to put certain responses into the context of specific participant groups. Precisely, we asked participants to select single-choice options regarding their years of experience (Q_{01}), the country in which they are currently employed (Q_{02}, free text), their company's main industry (Q_{03}), whether the company operates internationally (Q_{04}), and how many employees are in their company (Q_{05}). Finally, they should indicate the area in which they are currently working (Q_{06}, multiple answers). For each question, participants had the opportunity to use a *prefer not to say* option to ensure privacy in addition to anonymizing the data.

Section 2 (main causes, prevalence, and severity). In the second section, we asked for the participants' perceptions about vulnerabilities and configurability in software systems to answer **RQ$_1$** and **RQ$_2$**. First, we asked the participants to describe how vulnerabilities and configuring are connected in their daily work (Q_{07}). After that, we used two questions based on five-level Likert scales to assess the relevance of configuration-related vulnerabilities (Q_{08}) and specific triggers of vulnerabilities (Q_{09}), including an option to add other triggers (Q_{10}, optional free text). The next five questions comprised a ranking of development phases in which vulnerabilities are typically detected (Q_{11}) and an assessment of vulnerability risks (Q_{12}–Q_{15}, ten-level scales). Here, we intentionally oriented the assessment towards established security standards (i.e., ISO/IEC 27000 series [36], NIST Guide for Conducting Risk Assessments [69]), relying on severity, likelihood, impact, and exploits.

Section 3 (experiences). The last section mainly referred to experiences with configuration vulnerabilities. In particular, we asked in which domains the participants experienced the vulnerabilities (Q_{16}, multiple answers), how they became aware of them (Q_{17}, multiple answers), how they fixed (Q_{18}, multiple answers) and prevented (Q_{20}, multiple answers) them, and what the greatest impact of an exploit they experienced was (Q_{19}, free text). We again offered the opportunity to use a *prefer not to say* option. Our objective was to use the data we collected in this section to answer **RQ$_3$** and to complement our answers to **RQ$_1$** and **RQ$_2$**.

3.3 Conduct

We distributed our questionnaire through the channels mentioned above: personal network, LinkedIn, and snowballing. In total, we sent 66 invitations via e-mail to security experts in the authors' personal network. Furthermore, the first author shared a post through their LinkedIn network, with the request to contact us if anyone is interested in participating. Four people reached out via the LinkedIn messenger and we shared a link to the questionnaire with them. We encouraged all participants to share the invitation with other potentially interested experts, which led to an unknown number of further people we reached via this distribution channel. After closing the questionnaire, we received a total of 42 responses.

To analyze the data, we downloaded the Excel spreadsheet created by Microsoft forms. In a first review step, the first author screened the responses and excluded one due to insufficient data quality (e.g., using random characters in mandatory questions to skip them). Consequently, we considered 41 responses for our actual data analysis. The first author then applied open-coding and card sorting to gather recurring patterns and to identify relevant categories with their associated data; particularly for the free-text questions. The first two authors discussed all results through two meetings and iteratively refined them until they reached consensus.

4 Results

In the following, we describe the results of our questionnaire, structured according to the three individual sections.

4.1 Demographics

In Table 1, we present the results related to our participants' demographics. Most of our participants (39 %) have between 6 and 10 years of experience in the area of security (Q_{01}), followed by 29 % with 0 to 5 years, 17 % with 11 to 15 years, and 15 % with more than 15 years. All of our participants are from European countries (Q_{02}), mainly Central-Europe including Switzerland (24 %) and Germany (22 %). Other countries involve, for example, Ukraine (15 %), France, Spain, and Austria (10 % each). The companies for which our participants work (Q_{03}) are quite diverse. For instance, they operate in more traditional IT sectors (34 %), healthcare (20 %), finance (17 %), or manufacturing (15 %). Only 10 % are working on research and education, underpinning the practical orientation of our study. The companies (Q_{04}) typically operate internationally (90 %) rather than nationally (10 %), including (Q_{05}) small- (20 %) to medium-sized companies (53 %) and also larger companies or corporate groups (27 %). Not surprisingly, our participants' employment areas (Q_{06}) mainly relate to development (90 %). Other areas overlap with development, such as research, management, or operations (29 % each). Rarely mentioned other areas include, for example, security consulting.

Table 1: Overview of the responses for demographics (n = 41).

question	answers	responses	
Q_{01}: years of experience	0 – 5	12	29 %
	6 – 10	16	39 %
	11 – 15	7	17 %
	16+	6	15 %
Q_{02}: country	Switzerland	10	24 %
	Germany	9	22 %
	Ukraine	6	15 %
	France	4	10 %
	Spain	4	10 %
	Austria	4	10 %
	Netherlands	3	7 %
	Poland	1	2 %
Q_{03}: main industry	IT	14	34 %
	healthcare	8	20 %
	finance	7	17 %
	manufacturing	6	15 %
	research & education	4	10 %
	other	2	4 %
Q_{04}: internat. operation	yes	37	90 %
	no	4	10 %
Q_{05}: employees	0 – 10	8	20 %
	11 – 100	12	29 %
	101 – 500	10	24 %
	501 – 1000	2	5 %
	1001+	9	22 %
Q_{06}: employment area	development	37	90 %
	research	12	29 %
	management	12	29 %
	operations	12	29 %
	other	3	6 %

4.2 Main Causes, Prevalence, and Severity

The connections between configuration vulnerabilities and our participants' daily work (Q_{07}) are quite diverse. However, we identified five recurring patterns in their responses. Not surprisingly, most participants are typically concerned with more general secure application configuring, such as modeling secure configurations or DevSecOps (54 %). Overall, 24 % have experiences in security-related risk assessments of configurations, 17 % in secure versioning, and 10 % each in dependency checking as well as defensive configuring. We found several more experiences, which were mentioned fewer times, involving variant-richness and its reduction (5 %), countermeasure configuring (2 %), and plugin variety (2 %). Most of our participants strongly agreed (59 %) or agreed (39 %) that configuration vulnerabilities are a relevant topic in practice (Q_{08}). Only one person neither agreed nor disagreed with this statement.

Asking for the relevance of different vulnerability triggers (Q_{09}), our participants mentioned dependencies, outdated software, and inconsistent configurations as most relevant (cf. Figure 2). All other triggers were also supported by participants, but we identified

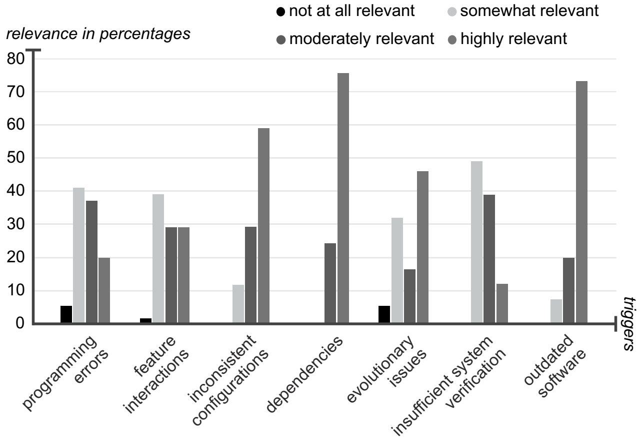


Figure 2: Relevance of vulnerability triggers (Q_{09}).

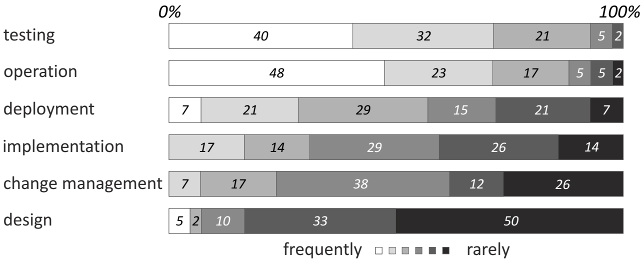


Figure 3: Ranking of the vulnerability occurrence likelihood in different development phases in percentages (Q_{11}).

more disagreements or participants who rated these triggers as only somewhat relevant. This applies in particular to programming errors, feature interactions, evolutionary issues, and insufficient system verification. Interestingly, evolutionary issues highlighted diverging opinions between participants with 46 % considering them highly relevant and 32 % somewhat relevant. Based on the participants' optional notes on triggers (Q_{10}), we found that social engineering is also a relevant trigger (34 %). For example, this relates to compromised development hardware or copying source code with vulnerable configurations from social platforms, such as Stack Overflow. Other triggers that were mentioned include impaired product or process integrity (5 %) and a lack of configuration overview related to configuration complexity (2 %).

According to the participants' ranking, configuration vulnerabilities typically occur or are revealed during testing and operation (Q_{11}, cf. Figure 3). Interestingly, the number of participants who rated operation as most relevant is slightly higher than for testing. In contrast, testing was rated as less relevant by fewer participants than operation. Generally, our participants think that vulnerabilities are less common to occur during the deployment and implementation. Change management and design were usually considered to be the least relevant phases.

As we illustrate in Figure 4, the vulnerability risk ratings (Q_{12}–Q_{15}) with respect to severity (average 7.5), likelihood (average 7.2), impact (average 8), and exploitation (average 7.1) are quite consistent. Regarding severity, most of the practitioners selected 7 (37 %) or 8 (22 %). There is also a great likelihood of configuration vulnerabilities, as implied by 37 % of our participants ranking it as a 7 and 22 % as an 8. Referring to impact, we can see a similar trend (i.e., 34 % refer to 7), with the additional note that 24 % selected a 10. Regarding the likelihood of exploitations, 32 % of our participants referred to an 8 and 29 % to a 7.

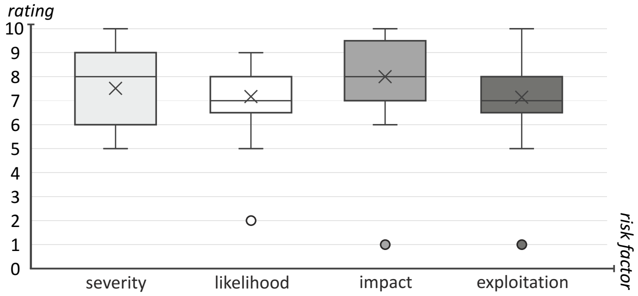


Figure 4: Vulnerability risk rating based on perceived severity, impact, likelihood, and exploitation (Q_{12}–Q_{15}); including interquartile range (boxes), rating range without outliers (outer lines), median (inner lines), average (crosses), and outliers (points) with 0 – lowest rating and 10 – highest rating.

4.3 Experiences

The domains in which our participants experienced configuration vulnerabilities (Q_{16}) are mainly related to server and storage development (93 %), web development (88 %), system and software development (46 %), as well as mobile development (32 %). One participant did not want to answer this question. As we show in Table 2, our participants stated that they usually become aware of configuration vulnerabilities (Q_{17}) through actual security incidents (90 %), word-of-mouth (83 %), regular software testing (76 %), and security scans (73 %). Fewer participants referred to vulnerability disclosures (34 %), new reports (24 %), or penetration testing (15 %).

Typical fixes (Q_{18}) involve configuration changes (85 %), version patches (83 %), risk acceptance (68 %), and implementing countermeasures (49 %). The exploits with the most impact (Q_{19}) vary and heavily depend on whether a participant was involved in the security process of one application or a whole system as well as how many end users were impacted. For example, 29 % of our participants mentioned more than 10,000 impacted users. In contrast, 20 % stated that only more than 100 users were impacted. Interestingly, some participants explicitly mentioned "almost" impacts (5 %) and reputation damage (2 %).

Regarding feasible prevention strategies (Q_{20}), we found (cf. Table 2), not surprisingly, that regularly reviewing and updating configurations as well as conducting regular security scans and/or audits are typical countermeasures (90 % each). Moreover, developing and enforcing security policies and procedures (76 %), integrating security into the development process (51 %), and using configuration management tools (37 %) have been used as prevention strategies by our participants. Interestingly, 12 % of our participants additionally described penetration testing as a prevention strategy.

Table 2: Overview of the awareness triggers (Q_{17}), common fixes (Q_{18}), and prevention strategies (Q_{20}) concerning configuration vulnerabilities.

question	answers	responses	
Q_{17}: awareness	security incident	37	90 %
	word-of-mouth	34	83 %
	software testing	31	76 %
	security scans	30	37 %
	vulnerability disclosure	14	34 %
	news report	10	24 %
	penetration testing	6	15 %
	cyber-threat intelligence	1	2 %
	threat report	1	2 %
Q_{18}: fixes	configuration changes	35	85 %
	patching	34	83 %
	risk acceptance	28	68 %
	countermeasures	20	49 %
	decommission system	1	2 %
	compensating controls	1	2 %
Q_{20}: prevention	regularly reviewing and updating configurations	37	90 %
	regular security scans/audits	37	90 %
	security policies and procedures	31	76 %
	integrating security into the development process	21	51 %
	configuration management tools	15	37 %
	penetration testing	5	12 %
	code reviews	1	2 %

5 Discussion

After describing our results, we now discuss our consequent findings to answer 🔍 RQ$_1$, ⚠ RQ$_2$, and 🛡 RQ$_3$.

5.1 🔍 RQ$_1$: Main Causes

Not surprisingly, our results indicate that configuration vulnerabilities are a relevant topic for practitioners in their daily work, which is further supported by previous research [22, 47, 93]. The responses regarding the main causes for such vulnerabilities point towards dependency issues, outdated software, and inconsistent configurations. We argue that the relevance of dependency issues reflects the challenge of managing complex (cross-)relationships between software systems and their components, with an outdated or insecure dependency easily causing vulnerabilities [55]. Companies may inadvertently introduce configuration vulnerabilities if they fail to regularly update dependencies or thoroughly check the components to integrate for security issues (e.g., when integrating third-party services).

Interestingly, we found that evolutionary issues seem to be a less relevant cause in the perception of practitioners (cf. Figure 2). This is despite evolution and dependency issues being obviously interconnected [3], such as delayed or missing updates [22]. Evolution issues often occur due to configuration drift between development phases, such as implementation, testing, and deployment, or within different branches in the same phase if configurations are not synchronized [25]. Tools that enforce configuration management policies have been suggested to maintain consistency [24], for example, automated dependency management tools to ensure that dependencies are regularly checked for known vulnerabilities [70]. Moreover, the lifecycle phases in which dependencies are integrated seem crucial: typically development and early testing [36, 69]. Ensuring proper dependency management during these phases is key to prevent configuration vulnerabilities in later development phases. Otherwise, the impact of such vulnerabilities can become even worse, as stated by many of our participants who faced such vulnerabilities in the real world (cf. Figure 3).

Other common issues, such as programming errors, social engineering, or feature interactions, were reported fewer times by our participants. In this context, we remark that practitioners may deliberately not report programming errors or social engineering, especially if they were responsible and may feel a certain amount of guilt (i.e., response bias) [22]. So, our findings do not mean that such issues are not relevant. In contrast, human errors are likely to cause misconfigurations [6, 22]. Moreover, the complexity of software systems and their configurable features will likely increase even more in the future. Consequently, we expect even more challenges in handling these complexities, and thus an increased potential for programming errors or unwanted feature interactions causing configuration vulnerabilities [27, 40, 54].

> 🔍 **RQ$_1$: Main Causes:** *Configuration vulnerabilities primarily stem from dependency management issues, outdated software, and inconsistent (cross-)configurations between different software systems and components. Human errors probably have a significant impact in this context.*

5.2 ⚠ RQ$_2$: Prevalence and Severity

Generally, our participants' ratings imply that configuration vulnerabilities are prevalent and severe. Based on our results, we argue that such vulnerabilities are ubiquitous across domains, countries, and company size, further underpinning the need for systematic security engineering and management. More specifically, we can see in Figure 4 that severity, likelihood, impact, and exploitation are all rated in the medium to high ranges (5 to 10). This indicates strong concerns regarding configuration vulnerabilities' potential to cause harm. These insights suggest that current practices, while reasonably effective, may not be sufficiently robust given the constantly evolving threat landscape and refinements of attacks [5].

Our participants' notable agreement on severity implies challenges in maintaining secure and consistent configurations as software complexity increases. Still, there are also a few outliers. These indicate that configuration vulnerabilities do not necessarily have serious consequences in every case. So, it is likely that configuration vulnerabilities are more common than we may anticipate, since they do not have to result in large-scale incidents or exploits [22]. Thus, we argue that configuration vulnerabilities may occur regularly without having critical consequences. Existing research has actually found that such vulnerabilities are often not recognized at all, as it is difficult to detect them; especially if they are silent, and thus do not produce any error messages [94]. If silent vulnerabilities are discovered by malicious actors and actual incidents (i.e., exploits) occur, they usually have a critical impact on the entire software system [47, 93].

Our results further indicate that mistakes in the design, change management, and implementation phases less frequently lead to vulnerabilities compared to the deployment, operation, and test phases. This could suggest that the phases that are primarily related to the design and implementation of the systems are less likely to lead to vulnerable configurations. However, we argue that this is likely a misperception. Most misconfigurations and configuration vulnerabilities originate from these phases [50, 95, 96]. However, they are revealed only during testing in a controlled setting or, in the worst case, during deployment and operation in a less controlled setting with greater impact. In turn, we argue that the awareness for configuration vulnerabilities should be improved already during the design and development of software systems.

> ⚠ **RQ$_2$: Prevalence and Severity:** *Configuration vulnerabilities are acknowledged as prevalent, with high severity and likelihood ratings if detected and exploited. They are typically revealed either in controlled settings (i.e., testing) or less controlled settings with more critical impact (i.e., operation).*

5.3 🛡 RQ$_3$: Treatment and Prevention

Most configuration vulnerabilities our participants experienced were related to server, storage, and web development. This trend is supported by security-related research, which highlights that these domains are particularly prone to vulnerabilities occurring and associated cyber attacks (e.g., cross-site scripting [89]); due to the use of the internet and communication technologies [4, 10, 55]. Surprisingly, more than 83 % of our participants stated that they are aware of vulnerabilities due to security incidents and word-of-mouth between practitioners. This clearly implies more reactive methods being in place regarding security awareness. Consequently, configuration vulnerabilities are apparently often recognized only after they have been exploited or actively discussed within the community (e.g., via Stack Overflow [55]).

Only about half of our participants mentioned that they integrated security into their development process. In contrast, preventive methods through software testing also play an important role (76 %). However, the testing seems less focused on identifying vulnerabilities directly rather than verifying software functionalities. Using security scans or vulnerability disclosures as preventive methods seems underrepresented, or may be not effective enough to identify vulnerabilities at an early stage in the development lifecycle. We argue that the reasons for the limited use of preventive methods may be related to the expensive nature of such methods [9, 22, 91] and the global security workforce gap [17, 41].

Overall, these findings highlight the need to balance reactive and preventive methods, ideally from an early development phase (i.e., design, implementation). For instance, referring to the product-line engineering framework [7], we strongly support the idea to include a security-engineering phase between domain engineering and application engineering [53, 61]. Moreover, as suggested by four participants, defensive configuring may be another feasible strategy at implementation level. For example, this may include defense-in-depth (re)configuration with privilege separation [39].

We can observe a similar trend regarding security fixes. Here, configuration changes and the prompt application of version patches are common reactive methods (each mentioned by more than 83 %). However, these should be supported by preventive methods to anticipate potential vulnerabilities, such as regular threat modeling and automated configuration checks. Accepting risks strategically (68 %) is a short-term, less expensive workaround [31], but clearly highlights the need for long-term strategies. To implement such strategies, enhanced prioritization frameworks and resource allocation can be helpful means.

Surprisingly, there is a quite low percentage of participants that mention the use of configuration management tools in their companies (37 %). We consider such tools as a suitable basis to prevent or identify configuration vulnerabilities. Particularly, we emphasize that there are already proposals in research covering different areas of securely developing highly-configurable software, such as secure product-line engineering [53], vulnerability management [84], and security testing based on feature models [43, 85, 86].

> 🛡 **RQ$_3$: Treatment and Prevention:** *Most practitioners seem to rely on reactive methods like configuration changes to fix configuration vulnerabilities after word-of-mouth or exploitation. Preventive methods like automated (defensive) configuration management are neglected, but should be strengthened in the future to avoid vulnerabilities and exploits from happening.*

6 Threats to Validity

We are aware of potential threats to the internal, external, and construct validity of our work, which we outline next.

Construct Validity. Threats to construct validity refer to the operationalization of variables and measurements (i.e., the questionnaire). The constructs of our questionnaire (e.g., severity of vulnerabilities and their rating) may have lead to misunderstandings among our participants. To mitigate this problem, we based our questions on existing literature and best practices on software security to ensure that the constructs were appropriately operationalized. In particular, the questionnaire and its terms refer to terms, definitions, and scales defined in established security standards, specifically the ISO/IEC 27000 series [36] and the NIST Guide to Conducting Risk Assessments [69]. These are well-established standards security experts are familiar with. Moreover, employing Likert scales (e.g., Q$_{08}$) and closed-ended questions (e.g., Q$_{18}$) may not entirely capture the participants' experiences and perceptions. We mitigated this threat by using open-ended questions (e.g., Q$_{10}$) and free-text options for closed-ended questions (e.g., Q$_{20}$) to allow participants to provide more detailed responses. Nevertheless, we cannot disregard the possibility that participants may have not answered our questions as detailed as possible to save time. We aimed to mitigate this threat by screening all answers at the beginning of the analysis to check whether there were any outliers (i.e., in the responses and the time taken)—leading to one exclusion.

Internal Validity. One threat related to the internal validity is selection bias, as we invited our participants primarily through our personal networks and LinkedIn. As a result, our participants may not represent our target population of software security professionals appropriately, potentially leading to biased results. To address this threat, we carefully identified security experts and sought to broaden our target population by encouraging participants to distribute the questionnaire to additional professionals within their networks. Another threat is response bias: participants may have provided socially desirable answers rather than truthful responses. This issue occurs particularly frequently in the context of sensitive topics [58] including vulnerability management [22]. To address this threat, we included options like *prefer not to say* (e.g., Q_{03}) and ensured that all responses are anonymized (i.e., collecting no personal data) to encourage honest answers. Generally, there may be several threats related to how we interpreted free-text responses (e.g., Q_{10}), which we aimed to mitigate by involving multiple researchers in the analysis process.

External Validity. Again, our participants may not represent the entire population of software security professionals, which limits the generalizability of our results. To address this, we used multiple distribution channels to reach a broader audience and encouraged participants to share the survey. Still, all of our participants are from Europe, and thus their experiences are likely influenced by regional regulations and practices. Moreover, we are aware that a larger number of participants would have strengthened the generalization of our findings. However, we argue that 41 participants and a return rate of 59 % are feasible values to collect and derive reliable results seeing the specific target group of experts we needed to invite [28]. In addition, more than 70 % of our participants have at least six years of experience in the field of security, which increases our confidence in the results.

7 Related Work

Vulnerability Causes and Treatment. There are various works on misconfiguration vulnerability triggers, their exploitation, and how to treat them, in particular related to web and server configuring. For instance, Loureiro [48], Martins et al. [52], and Xu and Zhou [92] surveyed risks and general treatment strategies oriented towards configuration vulnerabilities (e.g., misconfigured web servers). Furthermore, there are several papers presenting tools to detect misconfigurations leading to vulnerabilities. For example, Li et al. [47] proposed their tool ConfVD to identify vulnerabilities caused by SQL injections, while Eshete et al. [26] focused on a tool called Confeagle to detect vulnerabilities in the context of denial-of-service and session-hijacking attacks. Another line of research focuses on exploiting configuration vulnerabilities. For instance, Sulatycki and Fernandez [81] and Haimed et al. [29] present threat patterns to exploit misconfigurations, providing insights in how to build and configure applications more securely. Lastly, researchers focus on evaluations, for instance, Moura et al. [65] reproduced misconfigurations related to DNS services and evaluated their severity. In contrast to such works, we aimed to capture the state-of-practice to identify temporary problems and opportunities for improvements.

Practitioners' Voices. There are only few studies related to practitioners' experiences regarding configuration vulnerabilities. Manfredi et al. [51] studied the usability of security reports related to TLS miconfiguration patching based on a user study with 62 students. As part of their study on misconfigurations in open-source Kubernetes manifests, Rahman et al. [73] conduced interviews with nine developers to validate misconfigurations and obtain insights into the vulnerability detection processes. Bhuiyan et al. [12] presented a survey of 51 developers to identify vulnerability discovery strategies. Interestingly, they found that adapting configurations to trigger misconfigurations is a promising method. Moreover, May et al. [55] analyzed 651 Stack Overflow posts related to configuration vulnerabilities, providing a broader overview of concerns developers face.

The work closest to ours is the one by Dietrich et al. [22], who investigated system operators' perspectives on misconfigurations that impact system security. They combined qualitative interviews with a quantitative survey with more than 200 practitioners. Although they did not involve security practitioners, some of their results are in line with ours (i.e., misconfigurations are a common issue). However, generally Dietrich et al. have another focus, which is much more on business operations, such as setting budget restrictions for incident management. So, although the related work is somewhat similar to ours and may provide partly overlapping findings, we argue that they cover another body of knowledge that is out of our scope. We focus on a different topic based on security experts' opinions, leading to novel, practice-related insights.

8 Conclusion

In this paper, we reported a questionnaire with 41 security experts on configuration vulnerabilities. Our research goal was to understand the main causes, prevalence, severity, and treatments of such vulnerabilities. We found that configuration vulnerabilities are prevalent and severe. They primarily stem from dependency issues, outdated software, and inconsistent (cross-)configurations likely influenced by human errors. Such vulnerabilities are typically identified during testing (i.e., controlled settings) or, in the worst case, during the deployment and operation (i.e., less controlled settings). Overall, we found that there is a clear awareness for configuration vulnerabilities and their impact. However, due to the common use of reactive methods instead of more preventive ones, these vulnerabilities are often discovered too late and lead to a larger negative impact. Note that our results show several threats to validity (cf. Section 6), which might affect the validity of our findings.

Although researchers have proposed methods that partially address preventive methods, the transfer of these methods seems rather limited. Even if the higher costs of using such practices may be difficult to influence, we believe that they are highly valuable and that there are major knowledge gaps regarding their availability as well as use. In our future work, we aim to address these gaps to provide a comprehensive overview of existing methods to mitigate configuration vulnerabilities. For this purpose, we also aim to support the transfer of preventive research methods (e.g., defensive configuring) into practice.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. 2014. 42 variability bugs in the Linux kernel: A qualitative analysis. In *International Conference on Automated Software Engineering (ASE)*. ACM, 421–432.

[2] I. Abal, J. Melo, Ş. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wąsowski. 2018. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Transactions on Software Engineering and Methodology* 26, 3 (2018), 1–34.

[3] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. 2012. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software* 85, 10 (2012), 2228–2240.

[4] M. Abomhara and G. M. Køien. 2015. Cyber security and the internet of things: Vulnerabilities, threats, intruders and attacks. *Journal of Cyber Security and Mobility* (2015), 65–88.

[5] S. AboulEla, N. Ibrahim, S. Shehmir, A. Yadav, and R. Kashef. 2024. Navigating the cyber threat landscape: An in-depth analysis of attack detection within IoT ecosystems. *AI* 5, 2 (2024), 704–732.

[6] M. Alicea and I. Alsmadi. 2021. Misconfiguration in firewalls and network access controls: Literature review. *Future Internet* 13, 11 (2021), 283–298.

[7] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-oriented software product lines*. Springer.

[8] M. Arlitt and C. Williamson. 2004. Understanding web server configuration issues. *Software: Practice and Experience* 34, 2 (2004), 163–186.

[9] A. Asen, W. Bohmayr, S. Deutscher, M. González, and D. Mkrtchian. 2019. Are you spending enough on cybersecurity? *Boston Consulting Group* (2019).

[10] A. Bamrara. 2015. Evaluating database security and cyber attacks: A relational approach. *The Journal of Internet Banking and Commerce* 20, 2 (2015), 1–16.

[11] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. 2017. A general approach to network configuration verification. In *Conference of the ACM Special Interest Group on Data Communication*. ACM, 155–168.

[12] F. A. Bhuiyan, J. Murphy, P. Morrison, and A. Rahman. 2021. Practitioner perception of vulnerability discovery strategies. In *International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 41–44.

[13] M. Bilal, M. Canini, and R. Rodrigues. 2020. Finding the right cloud configuration for analytics clusters. In *ACM Symposium on Cloud Computing*. ACM, 208–222.

[14] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza. 2023. Automation for network security configuration: State of the art and research trends. *Comput. Surveys* 56, 3 (2023), 1–37.

[15] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. 2003. Feature interaction: A critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141.

[16] B. Chung, J. Kim, and Y. Jeon. 2016. On-demand security configuration for IoT devices. In *Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 1082–1084.

[17] W. Crumpler and J. A. Lewis. 2022. *Cybersecurity workforce gap*. JSTOR.

[18] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. 2012. Cool features and tough decisions: A comparison of variability modeling approaches. In *Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[19] D. Dalalana Bertoglio and A. F. Zorzo. 2017. Overview and open issues on penetration test. *Journal of the Brazilian Computer Society* 23 (2017), 1–16.

[20] V. Damasiotis, P. Fitsilis, and J. F. O'Kane. 2018. Modeling software development process complexity. *International Journal of Information Technology Project Management* 9, 4 (2018), 17–40.

[21] S. Dass and A. Siami Namin. 2021. Reinforcement learning for generating secure configurations. *Electronics* 10, 19 (2021), 1–19.

[22] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig. 2018. Investigating system operators' perspective on security misconfigurations. In *Conference on Computer and Communications Security (CCS)*. ACM, 1272–1289.

[23] S. Duan, V. Thummala, and S. Babu. 2009. Tuning database configuration parameters with ituned. *VLDB Endowment* 2, 1 (2009), 1246–1257.

[24] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. 2016. DevOps. *IEEE Software* 33, 3 (2016), 94–100.

[25] M. M. Emmanuel, M. N. Ibrahim, et al. 2015. Automatic synchronization of common parameters in configuration files. *Journal of Software Engineering and Applications* 8, 04 (2015), 192.

[26] B. Eshete, A. V., K. Weldemariam, and M. Zulkernine. 2013. Confeagle: Automated analysis of configuration vulnerabilities in web applications. In *International Conference on Software Security and Reliability (QRS)*. IEEE, 188–197.

[27] A. M. Gamundani and L. M. Nekare. 2018. A review of new trends in cyber attacks: A zoom into distributed database systems. In *IST-Africa*. IEEE, 1–9.

[28] A. N. Ghazi, K. Petersen, S. V. R. Reddy, and H. Nekkanti. 2018. Survey research in software engineering: Problems and mitigation strategies. *IEEE Access* 7 (2018), 24703–24718.

[29] I. B. Haimed, M. Albahar, and A. Alzubaidi. 2023. Exploiting misconfiguration vulnerabilities in Microsoft's Azure Active Directory for privilege escalation attacks. *Future Internet* 15, 7 (2023), 226.

[30] Z. He, K. Li, and K. Li. 2021. Cost-efficient server configuration and placement for mobile edge computing. *Transactions on Parallel and Distributed Systems* 33, 9 (2021), 2198–2212.

[31] A. Heyerdahl. 2022. Risk assessment without the risk? A controversy about security and risk in Norway. *Journal of Risk Research* 25, 2 (2022), 252–267.

[32] Z. B. Houidi and D. Rossi. 2022. Neural language models for network configuration: Opportunities and reality check. *Computer Communications* 193 (2022), 118–125.

[33] M. Humayun, M. Niazi, N. Z. Jhanjhi, M. Alshayeb, and S. Mahmood. 2020. Cyber security threats and vulnerabilities: A systematic mapping study. *Arabian Journal for Science and Engineering* 45, 4 (2020), 3171–3189.

[34] M. S. Iqbal, R. Krishna, M. A. Javidian, B. Ray, and P. Jamshidi. 2022. Unicorn: Reasoning about configurable system performance through the lens of causality. In *European Conference on Computer Systems (EuroSys)*. ACM, 199–217.

[35] ISO/IEC 25010 2011. *Systems and software engineering – SQuaRE - system and software quality*. Standard. ISO.

[36] ISO/IEC 27000 2018. *Information technology – security techniques – information security management systems*. Standard. ISO.

[37] ISO/IEC 27001 2013. *Information security management systems – requirements*. Standard. ISO.

[38] ISO/IEC 27005 2022. *Information security, cybersecurity and privacy protection – Guidance on managing information security risks*. Standard. ISO.

[39] T. Jaeger. 2016. Configuring software and systems for defense-in-depth. In *Workshop on Automated Decision Making for Active Cyber Defense (SafeConfig)*. ACM, 1–1.

[40] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar. 2017. Transfer learning for improving model predictions in highly configurable software. In *International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 31–41.

[41] C. A. Jordan. 2022. *Exploring the cybersecurity skills gap: A qualitative study of recruitment and retention from a human resource management perspective*. Northcentral University.

[42] J. Jürjens, K. Schneider, J. Bürger, F. P. Viertel, D. Strüber, M. Goedicke, R. Reussner, R. Heinrich, E. Taşpolatoğlu, M. Konersmann, et al. 2019. *Maintaining security in software evolution*. Springer.

[43] A. Kenner, S. Dassow, C. Lausberger, J. Krüger, and T. Leich. 2020. Using variability modeling to support security evaluations: Virtualizing the right attack scenarios. In *Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 1–9.

[44] A. Kenner, R. May, J. Krüger, G. Saake, and T. Leich. 2021. Safety, security, and configurable software systems: A systematic mapping study. In *Systems and Software Product Line Conference (SPLC)*. 148–159.

[45] R. A. Khan, S. U. Khan, H. U. Khan, and M. Ilyas. 2021. Systematic mapping study on security approaches in secure software engineering. *IEEE Access* 9 (2021), 19139–19160.

[46] A.-M. Konsta, A. L. Lafuente, B. Spiga, and N. Dragoni. 2024. Survey: Automatic generation of attack trees and attack graphs. *Computers & Security* 137 (2024), 103602.

[47] S. Li, W. Li, X. Liao, S. Peng, S. Zhou, Z. Jia, and T. Wang. 2018. Confvd: System reactions analysis and evaluation through misconfiguration injection. *IEEE Transactions on Reliability* 67, 4 (2018), 1393–1405.

[48] S. Loureiro. 2021. Security misconfigurations and how to prevent them. *Network Security* 2021, 5 (2021), 13–16.

[49] K. Ludwig, J. Krüger, and T. Leich. 2019. Covert and phantom features in annotations: Do they impact variability analysis?. In *Systems and Software Product Line Conference (SPLC)*. ACM, 218–230.

[50] I. Maganha, C. Silva, and L. M. D. F. Ferreira. 2019. The layout design in reconfigurable manufacturing systems: A literature review. *The International Journal of Advanced Manufacturing Technology* 105 (2019), 683–700.

[51] S. Manfredi, M. Ceccato, G. Sciarretta, and S. Ranise. 2021. Do security reports meet usability? Lessons learned from using actionable mitigations for patching tls misconfigurations. In *International Conference on Availability, Reliability and Security (ARES)*. ACM, 1–13.

[52] S. L. Martins, F. M. Cruz, R. P. Araújo, and C. M. R. Silva. 2024. Systematic literature review on security misconfigurations in web applications. *International Journal of Computers and Applications* (2024), 1–13.

[53] R. May, C. Biermann, A. Kenner, J. Krüger, and T. Leich. 2023. A product-line-engineering framework for secure enterprise-resource-planning systems. In *International Conference on ENTERprise Information Systems*. Elsevier, 1–8.

[54] R. May, C. Biermann, J. Krüger, G. Saake, and T. Leich. 2022. A systematic mapping study of security concepts for configurable data storages. In *Systems and Software Product Line Conference (SPLC)*. ACM, 108–119.

[55] R. May, C. Biermann, X. M. Zerweck, K. Ludwig, J. Krüger, and T. Leich. 2024. Vulnerably (mis)configured? Exploring 10 years of developers' Q&As on Stack Overflow. In *Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 112–122.

[56] R. May, J. Gautam, C. Sharma, C. Biermann, and T. Leich. 2023. A systematic mapping study on security in configurable safety-critical systems based on product-line concepts. In *International Conference on Software Technologies (ICSOFT)*. SciTePress, 217–224.

[57] R. May, J. Krüger, and T. Leich. 2024. SoK: How artificial-intelligence incidents can jeopardize safety and security. In *International Conference on Availability, Reliability and Security (ARES)*. ACM, 1–12.

[58] A. McCormac, D. Calic, M. Butavicius, K. Parsons, T. Zwaans, M. Pattinson, et al. 2017. A reliable measure of information security awareness and the identification of bias in responses. *Australasian Journal of Information Systems* 21 (2017), 1–12.

[59] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake. 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *International Conference on Automated Software Engineering (ASE)*. ACM, 483–494.

[60] P. Mell, K. Scarfone, and S. Romanosky. 2006. Common vulnerability scoring system. *IEEE Security & Privacy* 4, 6 (2006), 85–89.

[61] D. Mellado, E. Fernández-Medina, and M. Piattini. 2010. Security requirements engineering framework for software product lines. *Information and Software Technology* 52, 10 (2010), 1094–1117.

[62] D. Mellado, H. Mouratidis, and E. Fernández-Medina. 2014. Secure tropos framework for software product lines requirements engineering. *Computer Standards & Interfaces* 36, 4 (2014), 711–722.

[63] O. Mesa, R. Vieira, M. Viana, V. H. S. Durelli, E. Cirilo, M. Kalinowski, and C. Lucena. 2018. Understanding vulnerabilities in plugin-based web systems: An exploratory study of Wordpress. In *Systems and Software Product Line Conference (SPLC)*. ACM, 149–159.

[64] J. S. Molléri, K. Petersen, and E. Mendes. 2016. Survey guidelines in software engineering: An annotated review. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 1–6.

[65] G. C. M. Moura, S. Castro, J. Heidemann, and W. Hardaker. 2021. TsuNAME: Exploiting misconfiguration and vulnerability to DDoS DNS. In *Internet Measurement Conference (IMC)*. ACM, 398–418.

[66] D. Nešić, J. Krüger, S. Stănciulescu, and T. Berger. 2019. Principles of feature modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 62–73.

[67] A. Nhlabatsi, R. Laney, and B. Nuseibeh. 2008. Feature interaction: The security threat from within software systems. *Progress in Informatics* 5, 75 (2008), 1.

[68] NIST SP 800-154 2016. *Guide to data-centric system threat modeling.* Standard. National Institute of Standards and Technology.

[69] NIST SP 800-30r1 2012. *Guide for conducting risk assessments.* Standard. National Institute of Standards and Technology.

[70] I. Pashchenko, D.-L. Vu, and F. Massacci. 2020. A qualitative study of dependency management and its security implications. In *Conference on Computer and Communications Security (CCS)*. ACM, 1513–1531.

[71] K. Pohl, G. Böckle, and F. Van Der Linden. 2005. *Software product line engineering: Foundations, principles, and techniques.* Springer.

[72] H. Post and C. Sinz. 2008. Configuration lifting: Verification meets software configuration. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 347–350.

[73] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita. 2023. Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–36.

[74] S. Samonas and D. Coss. 2014. The CIA strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security* 10, 3 (2014).

[75] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac. 2017. Synthesizing configuration file specifications with association rule learning. *ACM on Programming Languages* 1 (2017), 1–20.

[76] A. M. Satpute, J. Priya, J. Mishra, and S. Anilkumar. 2022. Software reliability modelling and application in software development life cycle. *International Journal of Advances and Current Practices in Mobility* 5, 123 (2022), 1577–1584.

[77] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo. 2018. Software configuration engineering in practice: Interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering* 46, 6 (2018), 646–673.

[78] K. Scarfone and P. Mell. 2010. The common configuration scoring system (CCSS): Metrics for software security configuration vulnerabilities. *NIST Interagency Report* 7502 (2010).

[79] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. 2012. Software diversity: State of the art and perspectives. *International Journal on Software Tools for Technology Transfer* 14 (2012), 477–495.

[80] M. Seemann and S. van Deursen. 2019. *Dependency injection principles, practices, and patterns.* Simon and Schuster.

[81] R. Sulatycki and E. B. Fernandez. 2015. Two threat patterns that exploit security misconfiguration and sensitive data exposure vulnerabilities. In *European Conference on Pattern Language of Programs*. IEEE, 1–11.

[82] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *European Conference on Computer Systems (EuroSys)*. ACM, 47–60.

[83] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys* 47, 1 (2014), 1–45.

[84] Á. J. Varela-Vaca, D. Borrego, M. T. Gómez-López, R. M. Gasca, and A. G. Márquez. 2023. Feature models to boost the vulnerability management process. *Journal of Systems and Software* 195 (2023), 1–22 pages.

[85] Á. J. Varela-Vaca, R. M. Gasca, J. A. Carmona-Fombella, and M. T. Gómez-López. 2020. AMADEUS: Towards the autoMAteD secUrity teSting. In *Systems and Software Product Line Conference (SPLC)*. ACM, 1–12.

[86] Á. J. Varela-Vaca, D. G. Rosado, L. E. Sánchez, M. T. Gómez-López, R. M. Gasca, and E. Fernandez-Medina. 2021. CARMEN: A framework for the verification and diagnosis of the specification of security requirements in cyber-physical systems. *Computers in Industry* 132 (2021), 1–14.

[87] S. Wang, B. Luo, W. Shi, and D. Tiwari. 2016. Application configuration selection for energy-efficient execution on multicore systems. *J. Parallel and Distrib. Comput.* 87 (2016), 43–54.

[88] W. Wang, S. Jian, Y. Tan, Q. Wu, and C. Huang. 2022. Representation learning-based network intrusion detection system by capturing explicit and implicit feature interactions. *Computers & Security* 112 (2022), 102537.

[89] S. J. Weamie. 2022. Cross-site scripting attacks and defensive techniques: A comprehensive survey. *International Journal of Communications, Network and System Sciences* 15, 8 (2022), 126–148.

[90] Y. Wei, X. Sun, L. Bo, S. Cao, X. Xia, and B. Li. 2021. A comprehensive study on security bug characteristics. *Journal of Software: Evolution and Process* 33, 10 (2021), e2376.

[91] P. Wooderson and D. Ward. 2017. *Cybersecurity testing and validation.* Technical Report. SAE Technical Paper.

[92] T. Xu and Y. Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *Comput. Surveys* 47, 4 (2015), 1–41.

[93] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 159–172.

[94] J. Zhang, R. Piskac, E. Zhai, and T. Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages* 5 (2021), 1–30.

[95] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu. 2021. An evolutionary study of configuration design and implementation in cloud systems. *International Conference on Software Engineering (ICSE)*, 188–200.

[96] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong. 2016. Confmapper: Automated variable finding for configuration items in source code. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 228–235.

Selective Sampling: A Complexity-Aware Sampling Strategy for Combinatorial Scenario-based Testing

Lukas Birkemeyer
Karlsruhe Institute of Technology
Karlsruhe, Germany
lukas.birkemeyer@kit.edu

Tobias Pett
Karlsruhe Institute of Technology
Karlsruhe, Germany
tobias.pett@kit.edu

Tobias Runge
CQSE GmbH
München, Germany
runge@cqse.eu

Ina Schaefer
Karlsruhe Institute of Technology
Karlsruhe, Germany
ina.schaefer@kit.edu

Abstract

The SOTIF-standard (ISO 21448) requires scenario-based testing to verify and validate advanced driver assistance systems and automated driving systems. Feature modeling and sampling have shown promising results for generating scenarios considered test cases for scenario-based testing. Sampling strategies commonly applied for generating scenarios pursue coverage-criteria such as t-wise feature interaction, but ignore the number of selected features in a valid configuration. In the context of scenario generation, the number of features in a configuration correlates to the complexity of a scenario; thus, considering the number of features is relevant for "sufficiently" covering the scenario space and generating SOTIF-compliant scenarios. In this paper, we propose a complexity-aware coverage criterion embedded in selective sampling as a complexity-aware sampling strategy. Selective sampling approximates and replicates the distribution of the number of selected features of valid configurations. We apply selective sampling to generate scenarios for testing two advanced driver assistance systems. Our experiments demonstrate that selective sampling has the potential to improve scenario generation.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Sampling Strategy, Scenario-based Testing, ADAS

ACM Reference Format:
Lukas Birkemeyer, Tobias Pett, Tobias Runge, and Ina Schaefer. 2025. Selective Sampling: A Complexity-Aware Sampling Strategy for Combinatorial Scenario-based Testing. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3715340.3715411

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715411

1 Introduction

The Safety of the Intended Functionality (SOTIF)-standard (ISO 21448) [1] establishes scenario-based testing as state of the art for verifying and validating Advanced Driver Assistance Systems (ADAS) and Automated Driving Systems (ADS). In scenario-based testing, the behavior of a System Under Test (SUT) is evaluated while it interacts with its environment. A scenario serves as a test case for scenario-based testing. If the SUT behaves according to its specification, a scenario is considered passed; otherwise failed. The scenario describes, for example, the road infrastructure and traffic participants [1, 39]. Although SOTIF suggests scenario-based testing, it does not specify the methodology to generate or select scenarios. Testing all scenarios is not feasible due to the extremely large number of possible scenarios. Instead, SOTIF requires that scenarios used for testing *"sufficiently"* [1] cover an overall Operational Design Domain (ODD) but does not define what *sufficient coverage* exactly means which hinders its practical application.

Variability modeling techniques have shown promising results for combinatorially selecting test cases [4, 11, 16]. In the context of scenario generation, Birkemeyer et al. [11] use a feature model to represent the space of all possible scenarios of an ODD. A feature is equivalent to an atomic scenario entity (vehicle, pedestrian, wind, rain, etc.), a configuration is equivalent to a scenario, and a sample is equivalent to a scenario suite. Inspired by combinatorial interaction testing [31], existing approaches for generating scenario suites focus on a feature interaction coverage as a criterion to measure the representativeness of scenario suites wrt. an ODD [11, 19, 27]. Other sampling strategies consider additional feature attributes [4, 16, 19], the similarity of configurations [3], or their distance to a reference point [23]. Concentrating on these additional attributes might be relevant for optimizing sampling strategies for certain application domains. However, the proposed strategies do not cover additional properties of the possible configuration space, i.e., the scenario space. In order to *sufficiently* cover a scenario space as required by the SOTIF standard, properties of the scenario space are relevant. Following this argumentation raises the research question: Are existing feature model sampling strategies capable of generating scenario suites that *sufficiently* cover a scenario space?

In this paper, we propose *selective sampling* as a sampling strategy considering the *complexity-distribution* of a scenario space as a domain-specific attribute for *sufficiently* covering a scenario space.

We formalize a scenario's *complexity* as the number of atomic elements present in a scenario. A scenario containing a larger number of entities, such as traffic participants and traffic signs, is more complex than a scenario with no traffic participants and no traffic signs.[1] The *complexity-distribution* describes the frequencies of scenarios with a specific complexity value in a set of scenarios, i.e., a scenario suite or the overall scenario space. Selective sampling improves existing sampling strategies that already consider the number of features in a configuration [23] (i.e., the complexity of a scenario) by additionally considering the distribution of selected features per configuration (i.e., the complexity-distribution) in the overall possible configuration space. Thus, selective sampling leverages a complexity-aware coverage criterion in the context of sampling-based scenario generation. Additionally, selective sampling serves as a basis to unite complexity-aware coverage criteria with established feature interaction coverage criteria.

We investigate the complexity distribution of an overall scenario space and provide a prototypical implementation of *selective sampling*. We assess generated scenario suites and compare them to two baselines implementing existing complexity-aware sampling [23] and feature interaction coverage sampling [30]. As a metric, we use a mutation score that indicates the scenario suites' ability to detect potential failures [11, 15]. We analyze the impact of concentrating on complexity values that likely or rarely occur in the overall scenario space.

In summary, we make the following contributions:

- We propose *selective sampling* as a sampling strategy that considers the complexity distribution of a scenario space.
- We leverage selective sampling in the context of generating scenario suites for testing ADAS.
- We provide a prototypical implementation.
- We evaluate selective sampling for scenario generation with two automotive case studies.

2 State-of-the-Art

Current research comprises three techniques that contribute to SOTIF-compliant scenario generation [10]: (1) Optimization-based, (2) data-driven, and (3) combinatorial. *Optimization-based* techniques generate scenario suites by optimizing scenario parameters, for example, wrt. a criticality measure such as time to collision (TCC) [9, 42], driveable area [5], or accident velocities [20] potentially triggering hazardous behavior of the SUT. *Data-driven* techniques use (real-world) data when generating scenario suites [14, 17, 28, 32, 33]. This data is statistically analyzed [14, 33], used to train machine learning models [28], or to recreate accidents [17, 32]. *Combinatorial* techniques systematically combine atomic scenario elements [8, 11, 19, 27]. As required by the SOTIF-standard [1], combinatorial scenario generation is promising to generate subsets that cover an overall scenario space according to a coverage criterion [10, 11, 27]. In the following, we will focus on combinatorial scenario generation.

Combinatorial scenario generation commonly consists of two separate steps: (a) modeling a scenario space (i.e., an ODD) and (b) sampling concrete scenario suites from the scenario space model that (systematically) cover the overall scenario space. (a) Existing approaches that formalize scenario spaces use ontologies [8, 26] or feature models [11, 19]. Both models describe atomic scenario entities and their dependencies on each other. While ontologies are more powerful in descriptive power [13], feature models are beneficial for deriving scenario suites according to a coverage criterion by applying established sampling strategies known from variability modeling techniques. In Figure 1, we present an excerpt of a scenario space feature model according to Birkemeyer et al. [11] formalizing a scenario space relevant for testing Automatic Emergency Braking (AEB) functionality. The AEB automatically brakes a vehicle to a standstill to avoid or at least a collision.

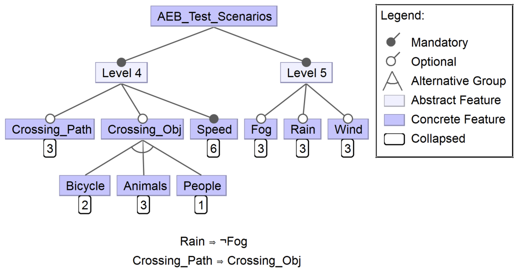
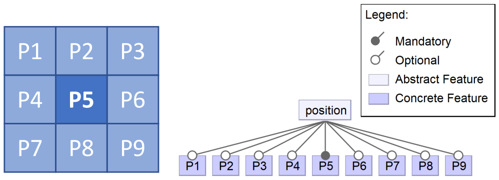


Figure 1: Excerpt of a scenario feature model covering relevant scenarios for testing AEB-functionality.

(b) Testing all possible scenarios covered in a real-world scenario space feature model is practically infeasible due to the combinatorial explosion problem [34, 40]. Thus, Birkemeyer et al. [11] apply sampling strategies that select scenarios either randomly (random sampling) or according to a coverage criterion. Feature interaction coverage sampling, for example, pursues a coverage criterion that focuses on covering all valid t-wise interactions between features [21, 22, 30]. In the context of scenario generation, using an interaction coverage criterion is beneficial for generating scenario suites that are effective in detecting failures of the SUT [11, 24, 27]. This observation is traceable to the interaction of faults discovered by Kuhn et al. [31]. Approaches such as [4, 16, 19] extend the conventional feature interaction coverage criterion by additionally considering feature attributes. They prioritize and optimize configurations wrt. to attributes modeled in the feature model. For example, they aim to generate test cases that maximize a user-defined quality attribute or minimize costs. Al-Hajjaji et al. [3] propose a sampling strategy that has the intention to quickly increase feature interaction coverage within the sampling process. They measure the similarity of two configurations and prioritize configurations that are dissimilar to already selected ones. Kaltenecker et al. [23] introduce distance-based sampling that determines the Manhattan distance [29] of each configuration to a reference point. Configurations are selected based on the distance values so that the resulting sample matches a predefined distance distribution. The predefined distance distribution is defined independently of the characteristics of the overall configuration space using expert knowledge. In diversified distance-based sampling, Kaltenecker et al. [23] prioritize configurations with underrepresented features in the sampling set. Diversified distance-based sampling strives for feature-wise coverage but does not guarantee it.

[1] *Complexity* describes a characteristic of a scenario independently of the SUT. Thus, complexity does not describe how complicated a scenario is wrt. an ADAS/ADS.



(a) position tile **(b) position feature model**

Figure 2: Grid structure to model vehicle positions in a feature model for the running example.

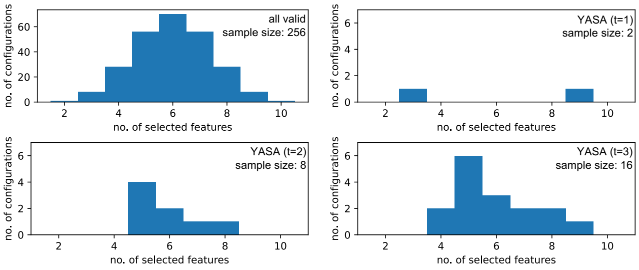


Figure 3: Complexity distribution for scenario suites generated with feature interaction coverage sampling for the running example.

Running Example. In the following, we introduce an (artificial) running example to demonstrate insufficiencies of existing sampling strategies for generating SOTIF-compliant scenario suites: We consider (inspired by [7, 36]) the surroundings of an ego-vehicle in a grid structure (see Figure 2 (a)). We model each cell of the grid as a feature in a feature model (see Figure 2 (b)). Selecting a feature of the feature model indicates traffic in the related cell. We expect the ego-vehicle to be located on position *P5* and model the feature *P5* as mandatory while vehicles in surrounding positions are optional features. In this example, the number of selected features in valid configurations varies between one (no traffic) and nine (traffic on each position). Although the running example is a simplification, it illustrates that the number of selected features in a configuration correlates with the complexity of a scenario.

In Figure 3, we present the complexity distributions for the overall scenario space of our running example (upper left) and the complexity distributions for scenario suites we derive using feature interaction coverage sampling. The scenario suites are generated using the sampling algorithm YASA [30] with the coverage criteria *t=1* (upper right), *t=2* (lower left), and *t=3* (lower right). The complexity distributions of scenario suites generated with YASA differ from those of all valid scenarios. These preliminary results substantiate the claim of Kaltenecker et al. [23], that established feature interaction coverage sampling does not consider the number of selected features in a configuration (i.e., the scenario's complexity or the scenario space's complexity distribution) although it is relevant for *sufficiently* covering a scenario space as required by SOTIF.

3 Selective Sampling for Scenario Generation

In the following, we describe the concept of *selective sampling* for a complexity-aware sampling strategy for generating SOTIF-compliant scenario suites. Selective sampling extends distance-based sampling [23] by considering the complexity distribution of the overall scenario space. First, we approximate the complexity distribution of the overall scenario space using a *preSample*. Second, we define a *selective strategy* implementing the complexity-aware coverage criterion. Third, we introduce the *basic selective sampling* algorithm and describe how we combine it with a feature interaction coverage criterion. To satisfy open science, we share an artifact of selective sampling in the form of a FeatureIDE plugin online.[2]

PreSample. To consider the complexity distribution of the overall scenario space in the sampling process, we need to determine the complexity distribution of the overall scenario space. Since generating and analyzing all valid scenarios is infeasible practically, we approximate the distribution in a *preSample* using uniform random sampling. Uniform random sampling considers each possible configuration equally likely; thus, we expect the preSample to have the same complexity distribution as the overall scenario space. If, for example, a complexity occurs often in the overall configuration space, uniform random sampling will select configurations with the same complexity with a higher probability. Selecting configurations with a complexity that rarely occurs in the overall sample set will rarely occur in the uniform random sample. We expect the larger the preSample, the better the approximation. One might argue that uniform random sampling is sufficient to fulfill a complexity-aware coverage criterion by replicating the complexity distribution of the overall scenario space. We counter that properly replicating the complexity distribution requires large scale scenario suites making scenario-based testing practically infeasible. Moreover, uniform random sampling solely focuses on replicating the complexity distribution although alternative coverage criteria might be relevant as we highlight in the following. Hence, the intention of selective sampling is to generate a large-scale preSample, analyze its characteristics, and derive a small scenario suite to practically perform scenario-based testing.

Selective Strategy. In addition to intuitively replicating the complexity distribution of the overall configuration space, it might be beneficial to focus, for example, on rare complexity values since they might be overlooked by testers. We refer to the strategy that is addressed as the *selective strategy*. In Figure 4, we present the ideal complexity distributions of three selective strategies that might be relevant for testing ADAS. (1) The *replication* strategy (green) has the potential to represent the characteristics of the whole configuration space. (2) The *inverse replication* (yellow) is promising since it focuses on complexity values that are rare in the overall scenario space. Scenarios that are rare in the space of possible scenarios might occur rarely in the real world. Thus, from a testing point of view, they are highly relevant because developers might not have considered them. (3) The *uniform distribution* (black) covers each complexity value equally, aiming to achieve a uniform distribution. In general, the definition of the selective strategy is arbitrary when using selective sampling. The novelty of selective sampling

[2] https://doi.org/10.5281/zenodo.14527453

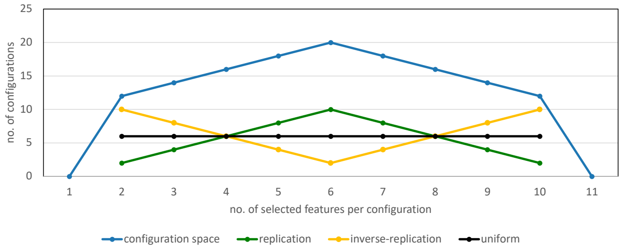


Figure 4: Ideal complexity distributions for the selective strategies replication, inverse, and uniform.

lies in the possibility of applying selective strategies that depend on the complexity distribution of the overall scenario space, for example, by replicating the complexity distribution or its inverse. The uniform distribution is independent of the overall complexity distribution, equivalent to distance-based sampling according to Kaltenecker et al. [23].

Basic Selective Sampling. Algorithm 1 shows pseudo-code for the basic selective sampling algorithm. The selective sampling algorithm is inspired by the distance-based sampling algorithm [23] and differs in terms of considering the preSample. Thus, the following selective sampling algorithm may be used, but is not limited, to generate samples with pre-defined complexity distributions such as uniform distribution. As an input, the basic selective sampling algorithm requires the preSample, the maximum size of the resulting scenario suite (*sampleSize*), and the selective strategy (*selectiveStrategy*). The algorithm generates a sample *sample* (i.e., a scenario suite) as its output. The starting point is an empty sample (line 1). The algorithm selects scenarios from the provided preSample until the maximum sample size (*sampleSize*) is reached or the preSample is exhausted (line 2). To select a scenario from a preSample, we first determine the complexity distribution *dist* of the preSample (see line 3). Based on this distribution and the *selectiveStrategy*, we determine a probability function and select a complexity value *c* (see line 4). Subsequently, we randomly select a scenario *scen* of the preSample that has the complexity value *c* (see line 5). We then add the configuration to the resulting *Sample* (see line 6) and remove it from the preSample (see line 7) to avoid duplicates. Removing already covered scenarios from the preSample modifies the preSample's complexity distribution by lowering the number of scenarios with complexity value *c*, which we consider in the next iteration.

Algorithm 1 Selective Sampling

```
Input: $preSample,\ sampleSize,\ selectiveStrategy$
Output: $Sample$
1: $Sample \leftarrow \emptyset$
2: while $|Sample| < sampleSize$ and $|preSample| > 0$ do
3:     $dist \leftarrow getDistribution(preSample)$
4:     $c \leftarrow selectComplexity(dist, selectiveStrategy)$
5:     $scen \leftarrow selectScenario(preSample,\ Sample,\ c)$
6:     $Sample \leftarrow Sample \cup \{scen\}$
7:     $preSample \leftarrow preSample \setminus \{scen\}$
8: end while
```

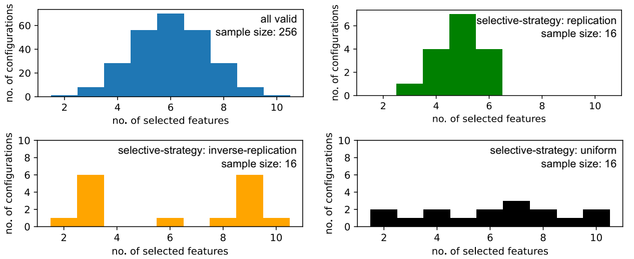


Figure 5: Complexity distributions for scenario suites generated with selective strategies *replication* (green), *inverse-replication* (yellow), and *uniform* (green).

For a proof of concept, we perform a preliminary experiment by applying selective sampling to the running example. We use the three selective strategies that we proposed above. As a preSample, we refer to all valid scenarios. The scenario suites that we generate with selective sampling contain 16 scenarios, the same number as for the sample set of YASA (*t=3*). In general, however, the number of scenarios per scenario suite is arbitrary. In Figure 5, we show the complexity distribution of the running example for samples with the three selective-strategies *replication* (green), *inverse-replication* (yellow), and *uniform* (black). All distributions are similar to the ideal complexity distributions (cf. Figure 4). Thus, the preliminary results underline that selective sampling considers the scenario complexity while selecting scenarios for scenario-based testing.

Combining Selective Sampling and Feature Interaction Coverage Sampling. In addition to the *number* of selected atomic scenario elements, we expect that their *type* is also relevant for testing ADAS/ADS. A stop sign, for example, triggers different behavior of the SUT than a speed-limit sign. Thus, solely focusing on complexity while sampling scenarios might be insufficient. To remedy this, we adapt the diversity concept of Kaltenecker et al. [23]. Instead of randomly selecting a scenario with a specific complexity, we prioritize scenarios that additionally fulfill specific criteria. Aiming for feature-wise coverage, for example, we prioritize a scenario if it contains atomic scenario elements that are rarely or not yet covered by the current sample. Other criteria that might be relevant consider the interaction of *t* features or minimize/maximize additional feature attributes. We implement this concept in the *selectScenario*-method of Algorithm 1 in line 5. Prioritizing configurations serves as a basis to combine selective sampling with coverage-based sampling. Since we do not implement coverage-based sampling strategies such as YASA [30], ICPL [21, 22], or Chvatal [12], this approach does not guarantee *t*-wise feature interaction coverage.

4 Evaluation

4.1 Research Questions

We propose using a preSample to approximate the complexity distribution of the overall configuration space. Therefore, we are interested in **RQ 1: Is a *preSample* a suitable approximation of the overall scenario space?** We subdivide this question by addressing two aspects: *RQ 1.1: How does a scenario-space's complexity distribution look?* The objective is to examine complexity distributions of randomly sampled preSamples with different sizes. If the complexity distributions of the preSamples are similar, we expect them to approximate the distribution of all valid configurations. The second aspect we address is the potential impact of the preSample on the quality of generated scenario suites. We ask *RQ 1.2: To what extent does the preSample impact generated scenario suites?* The objective is to generate scenario suites using selective sampling combined with multiple preSamples and compare them by their ability to detect potential failures in a SUT. We compare the impact of ten versions of preSamples. Inspired by Birkemeyer et al. [11], we use mutation testing to assess scenario suites according to their ability to detect potential failures.

The second research question we address in this evaluation is **RQ 2: Is selective sampling capable of improving combinatorial scenario sampling?** The objective is to generate scenario suites with selective sampling and to compare them to state-of-the-art combinatorial scenario sampling strategies. Similar to RQ 1.2, we use mutation testing to assess the quality of generated scenario suites. We generate scenario suites using basic selective sampling and selective sampling that strives for feature-wise coverage. We use standard feature interaction coverage sampling [30] with *t=1* and *t=2* coverage criterion and standard distance-based sampling [23] that aims for a uniform complexity distribution as baseline for our experiments. We analyze the impact of complexity-aware sampling and the impact of the selective strategy on the mutation score of scenario suites.

4.2 Experiment Design

Regarding RQ 1.1, we visualize complexity distributions of several preSamples and compare them manually with expert knowledge. Regarding RQ 1.2 and RQ 2, we generate scenario suites with different strategies and perform scenario-based testing in a virtual environment using the simulation tool CarMaker by IPG Automotive GmbH.[3] We explicitly vary only one parameter in the testing process at a time to trace possible influences back to specific parameters. In the following paragraphs, we provide details regarding the case studies, the mutation testing setup for assessing scenario suites, and the experimental runs to answer each research question. For the sake of reproducibility, we share the scenario spaces of our case studies online.[4]

Case-Study. We establish two automotive case studies: Adaptive Cruise Control (ACC) and Automatic Emergency Braking (AEB). An Adaptive Cruise Control (ACC) automatically controls the longitudinal velocity of a vehicle considering traffic ahead. An AEB-assistant automatically brakes the car to a standstill, avoiding or at least mitigating a collision. We implement both ADASs as Simulink models[5] that we adapt from [6] and The MathWorks, Inc. For each case study, we define a feature model in FeatureIDE [25, 37] and structure each, inspired by Birkemeyer et al. [11]. The feature model `fm-acc` covers scenarios relevant for testing ACC-functionality, and `fm-aeb` covers scenarios relevant for testing AEB-functionality. `fm-acc` contains 193 features and no cross-tree constraints representing $1.52 \cdot 10^{14}$ valid scenarios; `fm-aeb` contains 70 features and ten cross-tree constraints representing $2.98 \cdot 10^{6}$ valid scenarios.[6]

Assessment of Scenario Suites. To assess generated scenario suites, we refer to the mutation testing setup according to Birkemeyer et al. [11]. In mutation testing, faults are artificially seeded into a SUT resulting in so-called mutants [15]. Subsequently, test suites are assessed according to their ability to detect potential failures. As a metric, we measure the *mutation score*, which is the proportion of killed (i.e., detected) mutants to all generated mutants. The higher the mutation score, the better is the scenario suite. Mutation testing relies on the widely accepted coupling hypothesis [15] assuming that artificially seeded faults are coupled to real-world faults. To practically generate mutants, we seed mutation operators provided by the SIMULTATE mutation framework [35] into the SUT. Those mutation operators imitate typical faults developers implement, for example, mutation operators invert signals, set signals to zero (stuck-at-zero fault), or increment signals with a constant value. For each case study, we generate an arbitrary number of 50 mutants, randomly implementing exactly one mutation operator. The number of mutants is defined by expert knowledge, carefully balancing simulation effort and the meaningfulness of the results. While generating mutants, we avoid syntactic duplicates (i.e., generating the exact same mutant twice) but ignore equivalent mutants [18]. Equivalent mutants potentially systematically lower the mutation score, which, however, is irrelevant for relatively comparing mutation scores. As the test oracle, inspired by back-to-back testing [41], we use the original SUT. We compare the output of the mutant and original SUT after each simulation step. To decide whether a mutant is killed or alive, we refer to the safety-envelope kill criterion proposed by Birkemeyer et al. [11] using a collision detector. We accept changed behavior as long as it is not safety-critical.

RQ 1. Regarding RQ 1.1, we determine the complexity distribution of a real-world scenario space feature model. It is not clear how many scenarios are required to answer RQ 1.1. Thus, we generate three differently-sized scenario suites with uniform random sampling. In particular, we generate preSamples containing 1 000, 10 000, and 100 000 scenarios using the `RandomConfigurationGenerator` of FeatureIDE v3.8.3 [25, 37]; generating significantly greater preSamples is practically infeasible. The 100 000 scenarios cover approx. $0.657 \cdot 10^{-7}$% (`fm-acc`) and approx. 3.36% (`fm-aeb`) of the overall scenario space. We generate ten versions of each preSample. For answering RQ 1.1, we compare complexity distributions using histograms as a visualization.

Regarding RQ 1.2, we analyze the impact of the preSample on the quality of scenario suites generated using selective sampling. We generate ten versions of preSamples with uniform random sampling, including 1 000 scenarios each. We use basic selective sampling replicating the complexity distribution. To mitigate a bias due to selective sampling's nondeterminism, we generate ten versions of scenario suites for each preSample. Each scenario suite contains seven (for AEB) or ten (for ACC) scenarios. The number of scenarios per scenario suite is arbitrary; here, it is inspired by the feature interaction coverage sampling (feature-wise coverage)

[3] https://ipg-automotive.com

[4] https://doi.org/10.5281/zenodo.13958041

[5] https://www.mathworks.com/products/simulink.html

[6] counted with sharpSAT (https://github.com/marcthurley/sharpSAT) [38]

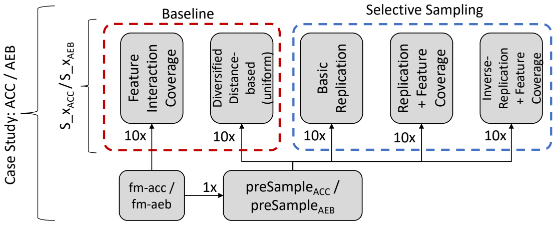

Case Study: ACC / AEB
S_x_ACC / S_x_AEB
Baseline
Selective Sampling
Feature Interaction Coverage
Diversified Distance-based (uniform)
Basic Replication
Replication + Feature Coverage
Inverse-Replication + Feature Coverage
10x
10x
10x
10x
10x
fm-acc / fm-aeb
1x
preSample_ACC / preSample_AEB

Figure 6: Schematic overview of experiments to answer RQ 2.

that we use as a baseline in the following experiments. To answer RQ 1.2, we determine and compare the median mutation scores for each preSample version.

RQ 2. In Figure 6, we present a schematic overview of the scenario suites we generate to answer RQ 2. We generate scenario suites using three selective sampling setups and compare them to two baselines. We perform the same experimental runs for both case studies (ACC and AEB). Regarding selective sampling, we approximate the overall scenario space of each case study with a constant preSample of 1 000 scenarios. To circumvent a bias caused by non-determinism, we generate ten versions of each scenario suite. For each case study, we consider two sizes of scenario suites. Both sizes are inspired by the baselines we describe in the following. To answer RQ 2, we determine and compare the median mutation scores for generated scenario suites.

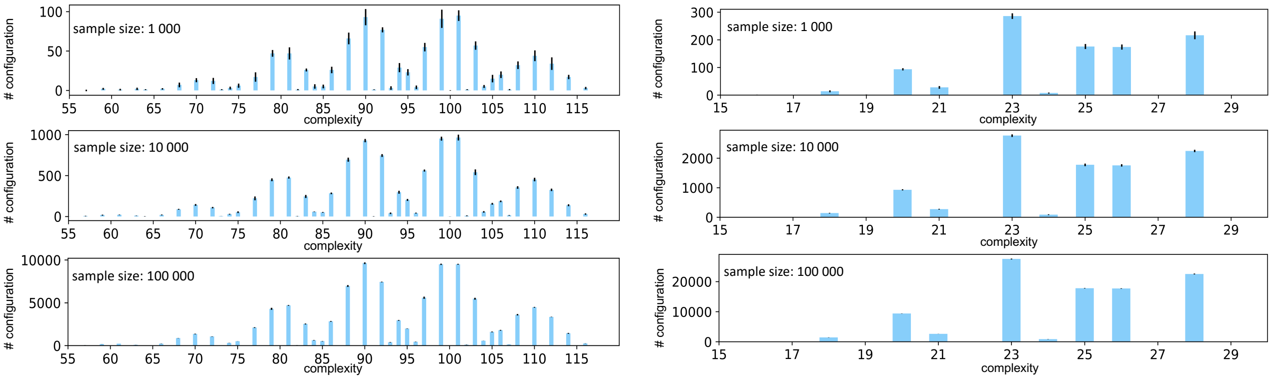
Baseline: We use *feature interaction coverage sampling* and complexity-aware *distance-based sampling* as baselines. To the best of our knowledge, distance-based sampling has never been applied to generate scenarios for scenario-based testing. Thus, we do not solely focus on distance-based sampling as a baseline. We explicitly avoid using baseline scenario suites that leverage state-of-the-art data-driven [14, 17, 28, 32, 33] or search-based optimization techniques [5, 20] because we can not ensure that the underlying scenario spaces (i.e., the scenario space feature model, database, and search space) are similar. Dissimilar scenario spaces might impact the results, leading to unreasonable conclusions. For feature interaction coverage sampling, we refer to feature-wise (*t=1*) and pair-wise (*t=2*) coverage criteria. Practically, we apply the state-of-the-art algorithm YASA [30] with default parameter setting. The `fm-acc`, leads to scenario suites of seven (*t=1*) and 80 (*t=2*) scenarios; the `fm-aeb`, leads to scenario suites of ten (*t=1*) and 86 (*t=2*) scenarios. Aiming for comparable results, we use these numbers to limit the size of the following scenario suites. For the sake of comprehensibility, we cluster all scenario suites of the same number of scenarios and case studies. We refer to them as S_7_{ACC}, S_10_{AEB}, S_80_{ACC}, and S_86_{AEB}. For distance-based sampling as the second baseline, we refer to the state-of-the-art diversified distance-based sampling provided by Kaltenecker et al. [23]. In particular, we focus on striving for a uniform complexity distribution and feature-wise coverage. Instead of applying the distance-based sampling algorithm shared in [23], we opt for the selective sampling algorithm that also strives for feature-wise coverage combined with a uniform selective strategy. As argued above, both implementations pursue similar goals. Using the selective sampling implementation in combination with a uniform selective strategy, however, mitigates a potential bias caused by the preSample while comparing and evaluating the impact of sampling strategies that are independent (distance-based sampling) or that depend on the complexity distribution of the overall scenario space (selective sampling).

Selective Sampling: We generate scenario suites using the replication selective strategy in combination with basic selective sampling and selective sampling that strives for feature-wise coverage. Comparing both scenario suites indicates whether striving for feature-wise coverage has the potential to improve the quality of complexity-aware scenario sampling. Moreover, we generate scenario suites using the inverse-replication selective strategy. We aim to identify whether focusing on rare complexity values in the overall scenario space is relevant for testing ADAS.

4.3 Results RQ 1: Complexity Distribution of a Scenario Space

RQ1.1: In Figure 7, we present the complexity distributions of the preSamples that we generate for the ACC case study (left) and the AEB case study (right). The histograms on top/middle/bottom relate to preSamples of 1 000/10 000/100 000 scenarios. The blue bar indicates the median number of scenarios with a specific complexity value, while the black line indicates the spread between the minimum and maximum number of scenarios. *Observation:* The complexity distributions of all three samples within each case study are similarly shaped. However, the complexity distributions vary across both case studies. The most represented complexity value of the ACC case study contains 90 atomic scenario elements, while most scenarios of the AEB case study contain 23 atomic scenario elements. *Interpretation:* We conclude that the complexity distribution of the overall scenario space is similar to the histograms we present in Figure 7 because all histograms of the same case study lead to similar distributions. Moreover, we conclude that the complexity distributions vary for different case studies.

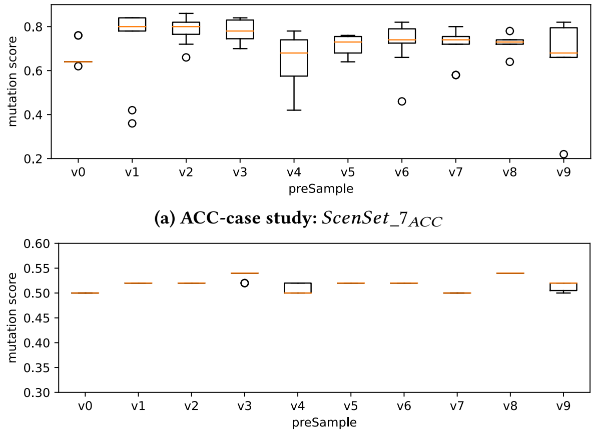
RQ1.2: We determine the impact of the preSample on the quality of generated scenario suites regarding the ability to detect potential failures. In Figure 8, we present mutation scores that we determine for scenario suites based on different preSamples. *Observation:* Regarding the ACC case study (see Figure 8 (a)), the median mutation scores that we determine for different preSamples vary between 0.64 and 0.8. For the AEB case study (see Figure 8 (b)), the median mutation scores vary between 0.5 and 0.54. It is worth mentioning that, for the ACC case study, the mutation scores for the same version of preSample scatter more than for the AEB case study. *Interpretation:* The results reveal two possible aspects that impact the mutation scores: (1) the impact of the preSample and (2) the impact of selective sampling. Both impacts might overlap, and thus, they are not clearly separable. We will address the impact of selective sampling in RQ2. Regarding the impact of the preSample, the median mutation scores scatter in a range of 0.16 (ACC) and 0.04 (AEB), which is greater than the average interquartile range for the same preSample version (0.1369 (ACC) and 0.0035 (AEB)). Thus, the preSample impacts the generated scenario suites, which is sub-optimal since the preSample is originally designed to independently approximate the complexity distribution of the overall scenario space. We conclude that the preSample might be a too



(a) ACC case study: Complexity Distribution

(b) AEB case study: Complexity Distribution

Figure 7: Distributions of median complexity values for differently sized preSamples for the ACC and AEB case study.



(a) ACC-case study: $ScenSet\_7_{ACC}$

(b) AEB-case study: $ScenSet\_10_{AEB}$

Figure 8: Impact of preSamples on generated scenario suites.

coarse granular approximation of the scenario space complexity distribution. A possible cause is uniform random sampling might not uniformly consider each scenario equally likely. Another cause is the size of the preSample, which might be too small since we observe that in the larger ACC-case study, the preSample has a higher impact on the median mutation scores than in the smaller AEB-case study. Acher et al. [2] state that both scalability and uniformity are challenges while realizing uniform random sampling. We consider the impact of the preSample to be a limiting factor of selective sampling. We refer to future work to identify a methodology for generating low-impact preSamples.

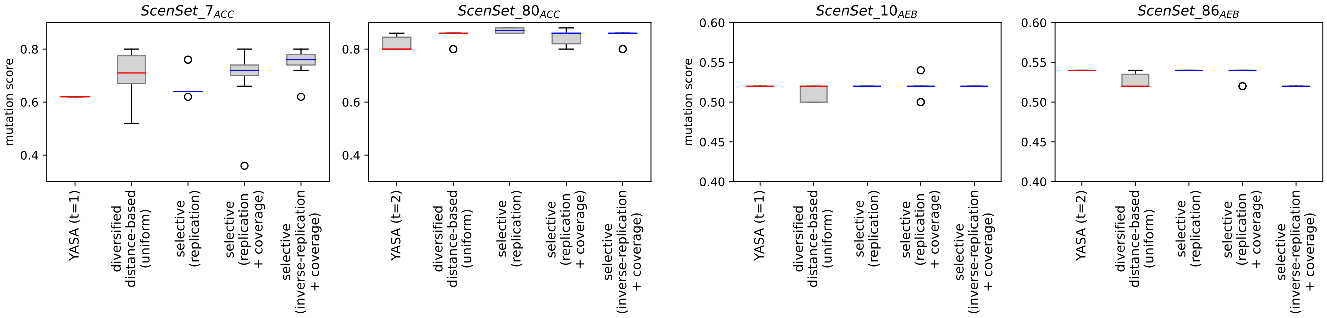
We sum up and answer RQ 1: The preSample approximates the complexity distribution of the overall configuration space (see RQ 1.1). The preSample might be too coarse granular for large-scale scenario spaces, which is a limiting factor for selective sampling. To mitigate a potential bias of the preSample, in the following experiments, we use a random constant preSample (i.e., v0, Figure 8) for each case study.

4.4 Results RQ 2: Assessing Selective Sampling

In Figure 9, we present the mutation scores that we determine for scenario suites generated with selective sampling (blue) and the two baselines (red). On the x-axis, we present the sampling strategies, and on the y-axis, the mutation score. For each case study, we present two diagrams that differ in scenario suite size.

Observation: First, comparing feature-interaction coverage sampling and distance-based sampling, we observe that distance-based sampling leads to higher median mutation scores for the ACC case study, while feature-interaction coverage sampling leads to similar or slightly higher median mutation scores for the AEB case study. The mutation scores for distance-based sampling scatter more than for feature interaction coverage sampling. Second, comparing basic selective sampling to feature interaction coverage sampling, we observe that basic selective sampling leads to higher or similar median mutation scores. Comparing basic selective sampling to distance-based sampling, we observe that basic selective sampling leads to similar or higher median mutation scores for the AEB case study, smaller median mutation scores for S_7_{ACC}, and higher median mutation scores for S_80_{ACC}. Third, comparing basic selective sampling to selective sampling striving for feature coverage, we observe that the median mutation scores are similar or higher for selective sampling striving for feature coverage. We also observe that the mutation scores for scenario sampling that also strive for feature coverage scatter more. Fourth, comparing the selective strategies replication and inverse-replication, we observe that inverse-replication leads to higher median mutation scores for the ACC case study, while inverse-replication leads to equal or slightly lower median mutation scores for the AEB case study.

Interpretation: Our results reveal that considering the scenario's complexity is relevant for generating scenario suites for scenario-based testing. For the ACC case study, complexity-aware sampling strategies lead to higher median mutation scores than solely focusing on feature interaction coverage, indicating higher quality scenario suites. However, the mutation scores scatter more, requiring multiple test runs for a proper safety argumentation. The mutation scores regarding the AEB case study hardly differ for several sampling strategies; thus, we conclude that the case study



(a) ACC case study: mutation scores for several scenario suites

(b) AEB case study: mutation scores for several scenario suites

Figure 9: Mutation scores to assess the failure-detecting ability of scenario suites generated with selective sampling (blue) and two baselines (red) for two case studies: ACC (a) and AEB (b). Mutation scores are not comparable across both case studies.

impacts the results. Combining complexity-aware coverage and feature interaction coverage benefits small scenario suites by leading to higher or similar results as solely focusing on complexity distributions. We assume that large scenario suites implicitly cover all features due to the scenario suites' size; other additional criteria, such as pair-wise feature interaction coverage, might be more relevant. Comparing selective sampling to existing complexity-ware sampling, we see that selective sampling leads to higher or at least similar median mutation scores. The additional benefit of selective sampling is that it applies a domain-specific coverage criterion by considering the complexity distribution of the overall scenario space. However, approximating the complexity distribution (i.e., generating the preSample) requires additional effort by analyzing scenario configurations; additional, cost-intense simulations are not required. Since we do not observe that one specific selective strategy leads to the highest median mutation score for all simulations, we conclude that selecting a useful selective strategy depends on the case study. This leaves the question of which case studies get along with standard distance-based sampling and which case studies benefit more from selective sampling.

We sum up and answer RQ 2: Our results indicate that complexity-aware sampling strategies are relevant for combinatorial scenario generation. Selective sampling can improve existing complexity-aware sampling strategies by realizing a domain-specific coverage criterion. Especially for generating sampling small scenario suites, combining the proposed complexity coverage and state of the art feature-wise coverage is beneficial. However, selective sampling requires additional effort to approximate the complexity distribution of the overall scenario space.

4.5 Threats to Validity

Internal validity: A first threat to internal validity is the implementation of the scenario space feature models, which could impact our results. To mitigate this threat, we implement the feature models according to the state-of-the-art scenario space feature model structure [11]. Another threat to the internal validity is that randomness might impact the results. We use random sampling (a) to populate the preSample, (b) to select a scenario with a concrete complexity value, and (c) to generate mutants. To mitigate a potential bias, we systematically reused random selections to keep them as static as possible and performed each experiment ten times. We reuse the same preSample for all complexity-aware sampling strategies, including distance-based sampling.

External validity: A threat to the external validity is that our case studies do not represent all possible ADASs. Both case studies are established in the automotive domain but impact the mutation scores. Thus, our conclusions might not generalize for arbitrary ADAS/ADS. Another threat to the external validity of our results is the representativeness of the artificially generated mutants we use in our experiments wrt. real-world faults. We refer to the widely accepted coupling effect hypotheses [15] and use mutation operators imitating typical mistakes of developers [35].

5 Conclusion

In this paper, we apply complexity-aware sampling strategies (i.e., sampling strategies that consider the number of selected features in a configuration) in the context of scenario generation for verifying and validating ADAS/ADS. Our results indicate that considering the scenario's complexity is beneficial for generating scenario suites. Moreover, we propose *selective sampling* as a novel complexity-aware sampling strategy. Compared to existing complexity-aware sampling strategies, selective sampling analyzes the overall scenario space. The additional benefit of selective sampling is the additional domain-specific coverage criterion relevant for generating SOTIF-compliant scenario suites for verifying and validating ADAS/ADS.

In future research, we will investigate the impact of the case studies and the preSample on the quality of generated scenario suites to improve the limitations of selective sampling. We are also interested in analyzing whether additional characteristics are required to expand the complexity definition. Using the number of atomic scenario entities to formalize the scenario's complexity might be too straightforward for real-world applications.

Acknowledgments

This paper project is partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263.

References

[1] 2022. ISO 21448:2022, Road vehicles â€" Safety of the intended functionality.

[2] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. 2021. BURST: a benchmarking platform for uniform random sampling techniques. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B*, Mohammad Reza Mousavi and Pierre-Yves Schobbens (Eds.). ACM, 36–40. https://doi.org/10.1145/3461002.3473070

[3] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. 2014. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*. 197–206.

[4] Mauricio Alférez, Mathieu Acher, José Angel Galindo, Benoit Baudry, and David Benavides. 2019. Modeling variability in the video domain: language and experience report. *Softw. Qual. J.* 27, 1 (2019), 307–347. https://doi.org/10.1007/s11219-017-9400-8

[5] Matthias Althoff and Sebastian Lutz. 2018. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1326–1333.

[6] Alberto Arcidiacono. 2018. ADAS virtual validation: ACC and AEB case study with IPG CarMaker.

[7] G Bagschik, T Menzel, C Körner, and M Maurer. 2018. Wissensbasierte szenariengenerierung für betriebsszenarien auf deutschen autobahnen. In *Workshop Fahrerassistenzsysteme und automatisiertes Fahren. Bd*, Vol. 12. 12.

[8] Gerrit Bagschik, Till Menzel, and Markus Maurer. 2018. Ontology based scene creation for the development of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1813–1820.

[9] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 63–74.

[10] Lukas Birkemeyer, Christian King, and Ina Schaefer. 2023. Is Scenario Generation Ready for SOTIF? A Systematic Literature Review. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*.

[11] Lukas Birkemeyer, Tobias Pett, Andreas Vogelsang, Christoph Seidl, and Ina Schaefer. 2022. Feature-Interaction Sampling for Scenario-based Testing of Advanced Driver Assistance Systems. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–10.

[12] Vasek Chvatal. 1979. A greedy heuristic for the set-covering problem. *Mathematics of operations research* 4, 3 (1979), 233–235.

[13] Krzysztof Czarnecki, Chang Hwan, Peter Kim, and KT Kalleberg. 2006. Feature models are views on ontologies. In *10th International Software Product Line Conference (SPLC'06)*. IEEE, 41–51.

[14] Erwin de Gelder and Jan-Pieter Paardekooper. 2017. Assessment of automated driving systems using real-life scenarios. In *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 589–594.

[15] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[16] José Angel Galindo, Mauricio Alférez, Mathieu Acher, Benoit Baudry, and David Benavides. 2014. A variability-based testing approach for synthesizing video sequences. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 293–303. https://doi.org/10.1145/2610384.2610411

[17] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 257–267.

[18] Bernhard JM Grün, David Schuler, and Andreas Zeller. 2009. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 192–199.

[19] Peng Guo and Feng Gao. 2020. Automated scenario generation and evaluation strategy for automatic driving system. In *2020 7th International conference on information science and control engineering (ICISCE)*. IEEE, 1722–1733.

[20] Ian Rhys Jenkins, Ludvig Oliver Gee, Alessia Knauss, Hang Yin, and Jan Schroeder. 2018. Accident scenario generation with recurrent neural networks. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 3340–3345.

[21] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14*. Springer, 638–652.

[22] Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 46–55.

[23] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094.

[24] Ludwig Kampel, Michael Wagner, Dimitris E Simos, Mihai Nica, Dino Dodig, David Kaufmann, and Franz Wotawa. 2023. Applying CT-FLA for AEB Function Testing: A Virtual Driving Case Study. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 237–245.

[25] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 611–614.

[26] Florian Klück, Yihao Li, Mihai Nica, Jianbo Tao, and Franz Wotawa. 2018. Using ontologies for test suites generation for automated and autonomous driving functions. In *2018 IEEE International symposium on software reliability engineering workshops (ISSREW)*. IEEE, 118–123.

[27] Florian Klück, Yihao Li, Jianbo Tao, and Franz Wotawa. 2023. An empirical comparison of combinatorial testing and search-based testing in the context of automated and autonomous driving systems. *Information and Software Technology* (2023), 107225.

[28] Robert Krajewski, Tobias Moers, Adrian Meister, and Lutz Eckstein. 2019. BézierVAE: Improved trajectory modeling using variational autoencoders for the safety validation of highly automated vehicles. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 3788–3795.

[29] Eugene F Krause. 1986. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation.

[30] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: yet another sampling algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–10.

[31] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. 2004. Software fault interactions and implications for software testing. *IEEE transactions on software engineering* 30, 6 (2004), 418–421.

[32] Francesco Montanari, Christoph Stadler, Jörg Sichermann, Reinhard German, and Anatoli Djanatliev. 2021. Maneuver-based resimulation of driving scenarios based on real driving data. In *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1124–1131.

[33] Demin Nalic, Arno Eichberger, Georg Hanzl, Martin Fellendorf, and Branko Rogic. 2019. Development of a co-simulation framework for systematic generation of scenarios for testing and validation of automated driving systems. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 1895–1901.

[34] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product sampling for product lines: the scalability challenge. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 78–83.

[35] Ingo Pill, Ivan Rubil, Franz Wotawa, and Mihai Nica. 2016. Simultate: A toolset for fault injection and mutation testing of simulink models. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 168–173.

[36] Paweł Skruch, Marcin Szelest, and Paweł Kowalczyk. 2021. An approach for evaluating the completeness of the test scenarios for the vehicle environmental perception-based systems. In *2021 25th International Conference on Methods and Models in Automation and Robotics (MMAR)*. IEEE, 133–138.

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

[38] Marc Thurley. 2006. sharpSAT-counting models with advanced component caching and implicit BCP. *SAT* 4121 (2006), 424–429.

[39] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. 2015. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th international conference on intelligent transportation systems*. IEEE, 982–988.

[40] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A classification of product sampling for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 1–13.

[41] Mladen A Vouk. 1988. On back-to-back testing. In *Computer Assurance, 1988. COMPASS'88*. IEEE, 84–91.

[42] Lukas Westhofen, Christian Neurohr, Tjark Koopmann, Martin Butz, Barbara Schütt, Fabian Utesch, Birte Neurohr, Christian Gutenkunst, and Eckard Böde. 2023. Criticality metrics for automated driving: A review and suitability analysis of the state of the art. *Archives of Computational Methods in Engineering* 30, 1 (2023), 1–35.

Cross-Vendor Variability Management for Cloud Systems Using the TOSCA DSL

Tobias Fellner
Johannes Kepler University
Linz, Austria
tobias.fellner@icloud.com

Paul Grünbacher
Johannes Kepler University
Linz, Austria
paul.gruenbacher@jku.at

Abstract

Cloud-based software systems offer numerous configuration options which significantly increase the complexity of variability management. The TOSCA DSL has been defined to reduce the reliance on specific vendors and to enhance the interoperability across different cloud services. This approach, however, does not support systematic variability management. Our paper thus introduces an approach addressing the challenges of supporting different types of configurations across different binding times in cloud-based systems. Specifically, by employing TOSCA's vendor-neutral definition language, our approach standardizes and simplifies the management of cloud configurations. We evaluate the correctness and performance of our approach for different configuration dimensions based on a cloud-based voting application. Our approach can improve the flexibility and efficiency of managing cloud environments, which is essential for robust and effective cloud operations.

CCS Concepts

• **General and reference → Design**; *Performance*; *Evaluation*; **Experimentation**; • **Software and its engineering → Specification languages**; *Design languages*; • **Computer systems organization → Cloud computing**.

Keywords

Cloud-based systems, domain-specific languages, configuration.

ACM Reference Format:
Tobias Fellner and Paul Grünbacher. 2025. Cross-Vendor Variability Management for Cloud Systems Using the TOSCA DSL. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3715340.3715433

1 Introduction

In contrast to monolithic software, cloud-based systems typically distribute business logic across multiple services. As a result, cloud-based systems use a heterogeneous technology stack comprising a variety of languages. To deal with this diversity, services are typically encapsulated in containers, with Docker being the de facto standard for containerization [2, 15, 17, 21]. Such "cloud-native" technologies offer many benefits, such as simplified component scalability, well-defined task management with clear interfaces, as well as higher flexibility regarding programming languages. At the same time, these technologies lead to new engineering challenges. In particular, the setup and maintenance of cloud-based systems require significant and often vendor-specific knowledge to ensure continuous adaptation and long-term success [11, 27]. Vendors typically use different modeling languages and methods for configuring cloud-based systems, often incompatible with each other. This means that changing a vendor is time-consuming, resource-intensive, or often even impossible [6]. To address these challenges, 40 organizations in the OASIS foundation have defined TOSCA (Topology and Orchestration Specification for Cloud Applications) [22], an open-source approach for vendor-independent modeling of an organization's cloud infrastructure, which covers resources, physical devices, and software structures. TOSCA provides a customizable and vendor-independent Domain-Specific Language (DSL) for describing and managing the required structures, processes, and resources, thereby reducing technology lock-in. However, it currently does not adequately address feature-based variability and configuration management needed in cloud-based product lines.

This paper thus introduces an approach that combines vendor-independent TOSCA definitions with variability management. It augments the TOSCA model to support the configuration process in cloud applications via centralized management of multi-dimensional configurations allowing the derivation of variants based on feature models. Specifically, we employ TOSCA to implement a cross-platform software product line (SPL) [1], which clearly distinguishes different levels of configuration and also deals with conflicting configuration settings. Our approach integrates TOSCA with orchestrators such as Docker Swarm[1] for dynamic configuration within a product-line context. We present a case study demonstrating how these challenges can be tackled in practice and discuss the resulting benefits. Furthermore, we examine how the approach impacts scalability and flexibility in cloud-native environments by analyzing the impact of configuration changes.

The paper is organized as follows: Section 2 introduces the background for our research. Section 3 presents an illustrative example showing key challenges. Section 4 presents our approach for systematic variability management in cloud-based environments. Section 5 covers the implementation strategy and the adopted technologies. Section 6 reports the research questions, the research method and the results of our evaluation. Section 7 discusses the results and threats to validity. Section 8 discusses related work. Finally, Section 9 concludes with a discussion and an outlook on future research.

[1] https://docs.docker.com/engine/swarm/

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715433

2 Background

Engineering cloud applications is a challenging and multifaceted task, complicated by the diverse approaches and techniques employed by various vendors. The most popular providers are Amazon Web Services (31%), Microsoft Azure (24%), and Alibaba Cloud (2%), with the remainder shared by Oracle, IBM, Salesforce, and others [26]. These vendors provide specific modeling languages to define individual cloud resources. Examples are Amazon Machine Image (AMI) structure and AWS CloudFormation templates, Microsoft's Azure Resource Manager (ARM), Oracle's VM Templates, or OpenStack's Heat Orchestration Templates (HOT) [4]. These vendor-specific definitions are typically based on JSON or YAML files. Although they allow fine-grained customization of the platforms, this comes at the expense of portability and flexibility.

In addition to this heterogeneity, the engineering of cloud-based systems is challenged by their sheer size and complexity, caused by the many entities and functions like virtual or physical compute instances, storage, databases, containers, serverless functions, or functions for user and policy management [25]. Pierantoni et al. identify application and infrastructure challenges such as deployment, scalability, and security, arising from the heightened complexity. They also point out that ensuring the portability of individual cloud entities can only be achieved with a universal description allowing applications to be deployed, launched, executed, and removed while hiding vendor-specific details [24].

2.1 Containerization

A fundamental concept of cloud computing is containerization [16, 21]. Containers are small operational isolated environments with a bounded context and manageable resource allocation [17]. The host system's kernel facilitates the operation of multiple isolated user-space instances and assigns the required resources to each container individually. Templates are created to bundle all software artifacts required to run a service. Containers can be created and operated simultaneously based on these images, which form their fundamental structure, while each instance's specific setup and configuration may vary, e.g., to accommodate defined resource limits, environmental variables, or network settings. In contrast to other virtualization concepts, containers share the host's OS kernel, which makes them lightweight and manageable, i.e., containers can be created, started, replicated, paused, stopped, and removed individually. While this can be done manually, it is usually done by a central container coordinator called an orchestrator [16]. Service operations are aligned with business processes through orchestrators like Kubernetes, Docker Swarm, Apache Mesos, and OpenShift, which manage scaling, runtime management, and monitoring.

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

Reference models provide foundational frameworks that organize data and its interconnections, supporting a clearer conceptual understanding of a specific domain. Essentially, these models present a universal schema and standardized vocabulary for a distinct field, while maintaining flexibility by not confining to a specific implementation [30]. In this regard, the Topology and Orchestration Specification for Cloud Applications (TOSCA) DSL has become a reference model for describing cloud-based software landscapes. TOSCA allows to define cloud architectures with entities like nodes, relationships, and resources in a vendor-neutral way. It is a universal language for defining the technical contexts of various structures, relationships, and processes within a cloud application landscape. The scope of modeling is vast because such landscapes consist of servers, virtual machines, container platforms, serverless functions, data stores, and many other structural elements needed to run a specific environment. TOSCA aims to deal with many different configuration languages by defining a comprehensive and flexible DSL to define, maintain, and monitor system environments [19]. Brogi et al. emphasize the following three main benefits of TOSCA [5]: *(i)* It automates deployment and management as developers use an abstract and ubiquitous language. *(ii)* It increases portability by defining components, relationships, requirements, and deployment workflows in a vendor-independent manner. *(iii)* It helps to develop reusable and modifiable components with an object-oriented approach, which allows the definition of custom structures and forces the user to use them correctly. It also allows the same elements to be referenced multiple times in multiple places and facilitates the fast replacement of individual components.

Specifically, a TOSCA-compliant system description can be segmented into several files containing definitions in XML or YAML formats. TOSCA provides a metamodel that defines a derivation hierarchy and refinement rules. This derivation concept is based on domain-specific types from which individual instances can be derived. In the context of TOSCA, these templates are called *Node Types*. A particular instance is a *Node Template*, which must adhere to the contract defined by the *Node Type*. Similarly, relationship types can be defined for specific relationships [22].

2.3 Software Configuration

Many authors have pointed out that software configuration is a multi-facetted problem [3, 28]: For instance, Berger et al. present a study on feature usage in industry. They report results based on an in-depth, contextualized analysis of 23 features and present a set of facets showing their diversity, with strong impacts on the configuration process [3]. Similarly, Siegmund et al. discuss the complexity of configuring software systems across multiple dimensions, highlighting the need for a classification based on eight factors: intent, stage, type, binding time, artifact, life cycle, stakeholder, and complexity [28]. The combination of these dimensions complicates tracing decisions and identifying optimal configuration points, particularly in compiled applications where the configuration time greatly affects flexibility across binding stages (build, deployment, loading, runtime). Moreover, the absence of well-defined languages for configuration increases the risk of misconfiguration, a challenge exacerbated as systems become larger and increasingly distributed.

3 Voting Application Example

To further illustrate these configuration challenges we analyzed an open-source voting application (cf. Figure 1) and defined TOSCA specifications for it.

The voting application utilizes the Command-Query-Responsibility-Segregation pattern [21] to separate read and write operations. This significantly increases scalability by allowing for more efficient

Change	Intent	Stage	Binding Time	Artifact	Life Cycle	Stakeholder	Scope
Container Instances	Scalability	Ops	Build	Config Code	Create	DevOps	Local
Port	Creating Multiple Services	Test	Deployment	Config Code	Maintain	DevOps	Distributed
Build Context	Reuse Existing Builds	PreProd	Build Time	Config File	Build	Ops	High Dep.
Logging	Fault Detection	Prod	Run Time	Filesystem	Own	User	Low Dep.
Env Variable	Avoid Side Effects	Dev	Load Time	CLI Parameter	Bind	Developer	Local
Dynamic Config	Increase Platform Flexibility	Test	Deployment	Env Variable	Maintain	DevOps	Distributed

Table 1: Impact of Configuration Changes for the Cloud-Based Voting Application (based on [28]).

resource management, particularly in scenarios where the demands on the system vary significantly between read and write operations. The primary objective of the application is the tracking of votes. It comprises a front-end web application written in Python that enables users to cast their votes between several voting options. The environment consists of a Redis instance that collects new votes, a .NET worker that consumes votes and stores them in a Postgres database backend by a Docker volume, and a Node.js web application that displays the results of the voting in real-time.

The .NET worker of the demonstration application enables the configuration of diagnostic functions by modifying the environment variable DOTNET_EnableDiagnostics. Our feature model defines two features which may be activated simultaneously: the first feature requires diagnostics, whereas the second feature prohibits diagnostics. This creates a clear conflict, as a decision needs to be made regarding the use of the configuration setting. Moreover, the environment variable influences the program's runtime behavior. In case the value is altered, the program must be reloaded or restarted in the .NET context. It is also imperative to ensure that only permitted values are written. Even setting an environment variable with a value range of zero to one can have considerable implications for resource management. When this environment variable is, therefore, combined with other configuration options, the likelihood for conflicts and errors increases significantly.

Table 1 summarizes the characteristics of different examples of configuration changes of the voting application on key configuration dimensions discussed in [28]. Specifically, it shows for six different types of configuration changes – affecting container instances, port, build context, logging, environment variables, and dynamic configuration – how different configuration dimensions are affected at different levels. The following two examples from Table 1 demonstrate how these configuration changes potentially affect various dimensions and levels. The change of a Port has the intent of creating multiple services. It occurs during the testing stage (Stage) for deployment binding (Binding Time) of the configuration code (Artifact) in the maintenance life cycle (Life Cycle). This process is managed by the DevOps team (Stakeholder) within a distributed system (Scope). Another example is the change of "Logging", which aims at improving fault detection (Intent) during the production phase (Stage) for runtime binding (Binding Time) of the file system (Artifact) in the life cycle of ownership (Life Cycle) by the end user (Stakeholder) in the system with low dependence on other instances (Scope).

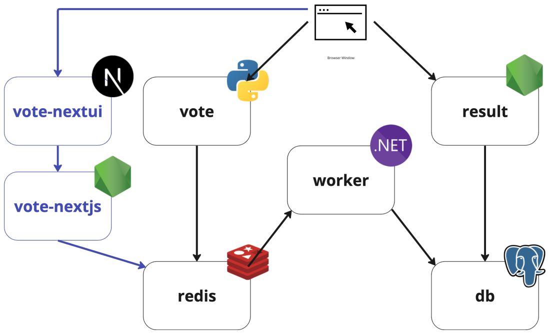


Figure 1: Voting Application. For details see https://github.com/dockersamples/example-voting-app.

4 Approach

Figure 2 provides an overview of our approach, which extends TOSCA's structure definitions: elements of the existing TOSCA 2.0 architecture are shown in gray and white. For creating custom TOSCA implementations, the OASIS TOSCA TC defines the application of a structured pipeline including a *Preprocessor*, *Orchestrator* and a target *Platform* as illustrated in Figure 2. The main task of the preprocessor is to import TOSCA definitions like topologies, policies, and services, and to compose them with external data and specified inputs. These inputs are then typically bundled in a Cloud Service Archive (CSAR), a compression format for TOSCA applications. The resolution process then applies variable inputs, checks compliance, and transforms nodes for further steps. The result is a representation to be processed by an orchestrator, which uses concrete topology representations to configure environments on various platforms. It manages service operations to align with business processes, to coordinate interactions, and to perform tasks like scaling, monitoring, and lifecycle management.

Our approach extends the standard TOSCA architecture by integrating feature-based runtime management and a custom feature resolution mechanism. It relies on additional inputs such as a *Feature Model*, a *Product Configuration*, *Feature-Service Mapping* (FSM), and *Resolution Strategy Definitions* (cf. Figure 2 and Figure 3). A dedicated *Feature Resolver* processes these inputs together with the existing TOSCA structure. It activates or deactivates specific *Features* or resolves conflicts between different configuration settings. The *Feature Resolver* is placed on top of the TOSCA resolver and uses the already parsed and executable TOSCA environment and augments this configuration with the feature configuration according to the Product Configuration and the FSM. The FSM is the central place for connecting features with services and setting configuration variables. Figure 3 illustrates this specific *Manual*

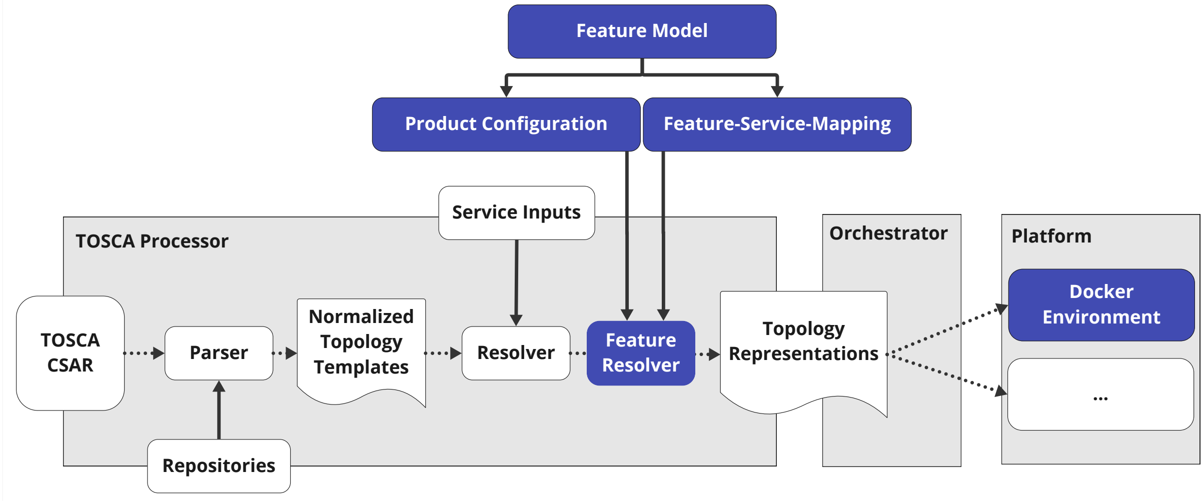


Figure 2: Our approach extends the TOSCA pipeline. Dotted lines represent data flows, solid lines depict references and inputs.

Mapping approach [13], which simplifies and centralizes the management of variability but also improves the traceability of feature configurations, to facilitate the maintenance and evolution of the software product line. Moreover, the FSM defines dependencies and commonalities, as well as conflict resolution strategies.

4.1 Mapping Features to Configuration

Effectively managing the complexity of feature integration and implementation requires a well-structured mapping approach that aligns all components with the overall system architecture, minimizing the risk of inconsistencies and conflicts. Figure 3 illustrates a distinct *Mapping Configuration* that defines the connections between *Features* and their concrete implementations within an additional TOSCA-compliant file. In this configuration, each *Mapping* establishes a $1 : n$ relationship with the associated implementation *Container*, which allows to flexibly associate features with multiple implementation units.

Furthermore, a *Mapping* can reference one or more specific *FeatureConfigs*, or a *FeatureConfigSet*, which aggregates multiple *FeatureConfigs*. This modular design facilitates the reuse of configurations across different features. These *FeatureConfigs* are connected to the corresponding features within the Feature Model, ensuring that the system accurately reflects the intended product capabilities.

Containers and Features may exhibit an $n : m$ relationship, with each connection potentially associated with *FeatureConfigs* that include variables for different binding types. This flexibility allows the customization of feature behavior based on varying deployment contexts, and enables systems to adapt to a wide range of requirements. The structured mapping simplifies the integration process and also supports the efficient resolution of conflicts that may arise due to overlapping configurations.

Constraints between features may limit their simultaneous presence in the final product. However, due to the $n : m$ relationship between features and containers, there is a significant potential for configuration conflicts. For example, if the presence of two features results in changes to the same environment variable, an appropriate conflict resolution strategy must be applied to decide which value to prefer for a special use case and service.

Figure 3 provides an illustrative example of the *Manual Mapping* [13]. In detail, Figure 4 illustrates the YAML-based representation of the entities and their references. For mapping demonstration, a subset of the application presented in Figure 1 is used. Suppose there are three containers, each running an independent service: *result* and *vote* are web-based frontends, while *worker* is a data service. While all services can run independently, there are dependencies to provide a correct product version. Defining the docker containers of the application in TOSCA 2.0 goes hand in hand with defining the feature model comprising container-specific as well as application-specific features. The main problem is aligning the features with the configuration values of the existing TOSCA environment.

This is where the mapping configuration becomes important: relationships are modeled in a graph and enriched with additional values. While an adapted database connection string for the feature Dev may be essential for the *worker*, the frontend may only have visual effects, like showing a version number. Therefore, different targeted *FeatureConfigs* can be defined using many-to-many relationships to handle conflict resolutions individually. To improve reusability, certain *FeatureConfigs* can be grouped into *FeatureConfigSets*, a bundling mechanism not affecting functionality. The FSM is built on top of the TOSCA environment as a central configuration and mapping location, as shown in Figure 3. Therefore, each mapped service has an associated mapping (M1, M2, M3). Independently of this mapping, a Feature Model (cf. Figure 3) defines seven features (Java, C#, Go, Light, Dark, Dev, Prod), which can be linked to services in form of *FeatureConfigs*. Depending on the configuration, particular values need to be set. While M3 depends on the two features Dev and Prod, M1 refers to a set of features. As already mentioned, this improves the reusability of context-related features and does not offer additional functionality. Thus, M2 can refer to this feature set without redefining any configuration variable. Even though M1 and M2 may have the same features, they

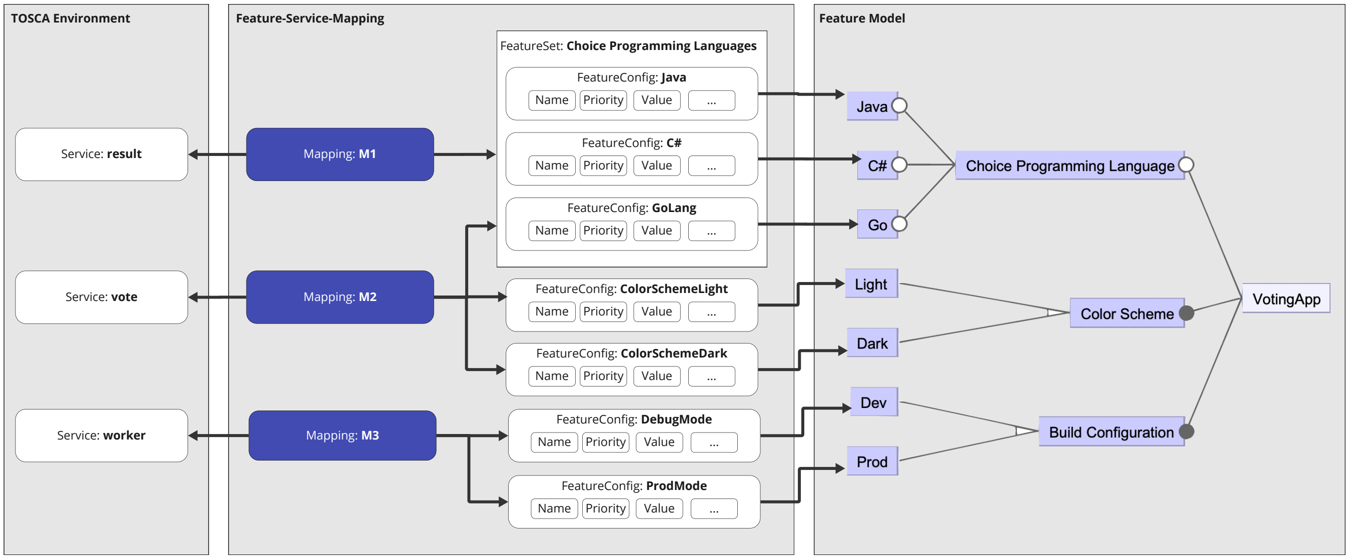


Figure 3: Mapping Features to Containers.

can individually define a conflict resolution strategy. While M1 can decide the priority of the values, M2 can use some custom logic defined in a conflict resolution script. Additionally, it may be crucial to assign these services to the same virtual network for communication purposes, which can also be grouped in the FSM.

4.2 Resolving Variability Conflicts

Conflict resolution is managed through the *Mapping Configuration*. As previously mentioned, each feature can have multiple associated feature configurations, and each feature configuration can define multiple variables. Figure 4 illustrates an example of setting an environment variable when the feature is enabled. This configuration is part of the FSM (cf. Figure 3). The script referenced here can be tailored to suit specific business needs. This method is invoked from the main process in a JavaScript runtime to make quick and individual adjustments, and the resulting outcome is used as a resolved conflict. In Figure 4, the resolveHighestPriority method from configresolving.js is employed for the service worker. It's worth noting that each service can utilize its own resolution strategy if necessary, and different sub-systems can respond to conflicts in their own unique ways.

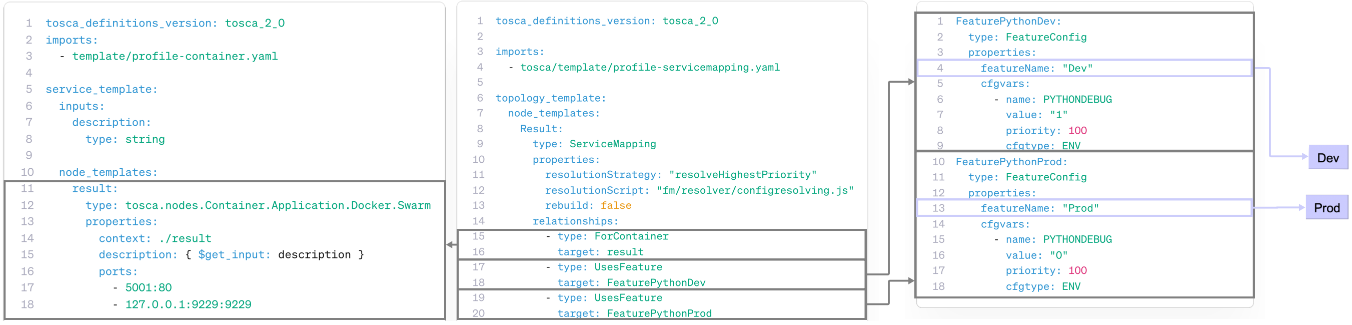
In the example in Figure 4, the variable is defined with the type ENV, indicating that a restart is required if the variable changes. Alternatively, DYNAMIC or DYNAMIC_LIST could be executed at runtime, while TOSCA_INPUT or a rebuild property, for example, could trigger a rebuild of the service. Additionally, a priority of 100 is assigned to the individual FeatureConfigs FeaturePythonDev and FeaturePythonProd. Since the resolution strategy is set to resolveHighestPriority, this function is invoked by the associated script whenever conflict resolution is necessary.

5 Implementation

Our implementation is publicly available[2] and uses the Dependency Inversion Principle [23] to allow working with different orchestrators. A pipeline oversees each process phase, from importing all essential files and resolving features to translating them into the specific orchestrator. After deploying the environment to the specified target, configuration and product changes are managed using an event-driven approach over a common *Consul*[3] configuration management platform. We selected Docker Swarm as the primary orchestrator for the initial demonstration. Furthermore, we developed a minimalist Kubernetes endpoint demonstrating the extensibility of our approach.

The criteria for selecting programming languages and frameworks included their open-source availability and prevalence of adoption. The *Feature IDE* [18] framework, widely used in research and practice, was selected for variability modeling and management. The model and product configuration are stored in an XML format, ensuring subsequent processing and interoperability with other tools and systems. The open-source programming language Go[4], a leading choice in cloud computing, was chosen for implementation. An extensive review of Cloud Native Computing Foundation projects[5] reveals that Go is the dominant language in the cloud-native ecosystem and one of the most popular and efficient languages for cloud-native services. The resolution of dynamic features was implemented in via JavaScript, while we used TypeScript[6], a strongly typed, object-oriented language, to define the resolution strategies. This TypeScript code can be modified at runtime. Before the resolution process the TypeScript code is compiled into JavaScript, which is subsequently interpreted. Using compiled

[2] https://gitlab.com/tobias.fellner/tosca_featuremodeling
[3] https://www.consul.io/
[4] https://go.dev/
[5] https://www.cncf.io/projects/
[6] https://www.typescriptlang.org/



```
tosca_definitions_version: tosca_2_0
imports:
  - template/profile-container.yaml

service_template:
  inputs:
    description:
      type: string

  node_templates:
    result:
      type: tosca.nodes.Container.Application.Docker.Swarm
      properties:
        context: ./result
        description: { $get_input: description }
        ports:
          - 5001:80
          - 127.0.0.1:9229:9229
```

```
tosca_definitions_version: tosca_2_0

imports:
  - tosca/template/profile-servicemapping.yaml

topology_template:
  node_templates:
    Result:
      type: ServiceMapping
      properties:
        resolutionStrategy: "resolveHighestPriority"
        resolutionScript: "fm/resolver/configresolving.js"
        rebuild: false
      relationships:
        - type: ForContainer
          target: result
        - type: UsesFeature
          target: FeaturePythonDev
        - type: UsesFeature
          target: FeaturePythonProd
```

```
FeaturePythonDev:
  type: FeatureConfig
  properties:
    featureName: "Dev"
    cfgvars:
      - name: PYTHONDEBUG
        value: "1"
        priority: 100
        cfgtype: ENV
FeaturePythonProd:
  type: FeatureConfig
  properties:
    featureName: "Prod"
    cfgvars:
      - name: PYTHONDEBUG
        value: "0"
        priority: 100
        cfgtype: ENV
```

Figure 4: Connecting TOSCA components and features with references.

TypeScript code and interpreted JavaScript code facilitates dynamic reconfiguration at runtime.

The necessity for compatibility with TOSCA 2.0 considerably reduced the scope of suitable TOSCA-compliant tools. An open-source implementations fully supporting TOSCA 2.0 is Puccini[7], which follows the structural design proposed by the standard (cf. Figure 2) and divides the implementation process into sequential steps. The library allows dynamic loading of TOSCA files, including all associated imports and structures. In addition, the data structures are converted into an object-oriented format and validated before further processing.

6 Evaluation

Our preliminary evaluation assesses the correctness and performance of our approach. Specifically, we investigated two research questions:

RQ1–Correctness. To what extent is the approach capable to generate correct variants of cloud-based software, i.e., performs its intended functions? We explored different configuration scenarios to check if the results are correct.

RQ2–Performance. To what extent do configuration changes at different binding times affect the performance and scalability?

6.1 Research Method

The voting application introduced in Section 3 comprises heterogeneous cloud services and is well suited to evaluate the different aspects of our approach. As already shown, even though it is a small example, it effectively highlights various configuration dimensions, showcasing the versatility and adaptability of our method. We first defined the application defined in the TOSCA DSL before extending it with additional nodes, such as a *NextJS* Frontend and a *NodeJS* backend. In addition, we defined a feature model also including important constraints. Subsequently, the FSM could be carried out as depicted in Figure 3. We then defined various types of configurations and assigned them to services. These include different database configurations, environment variables that affect container-internal Python and .NET compilation processes (e.g., debug mode), global settings for color schemes, and feature-based configuration of the content displayed in web applications. In addition, conflict resolution strategies were implemented in TypeScript files referenced by the FSMs, thus enabling each service to resolve configuration conflicts. This approach ensured that each configuration type was tailored to the specific needs of the service, thus improving the overall system flexibility and robustness.

Regarding RQ1, we generated different variants with diverse configurations requiring various binding times. Our findings are based on visual inspection, systematic performance measurements, and review of artifacts. Moreover, we applied different resolution strategies to demonstrate the correctness of the dynamic configuration resolution. Therefore, custom resolving functions were extended by using the highest priority, the value of the variable, and a custom static value.

We used prototyping to determine the resulting artifacts' functionality, flexibility, and correctness. This allowed the identification of problems and potential optimization opportunities. In particular, the whole prototyping was carried out in iterative cycles, where the software was regularly tested and improved to meet the system's requirements. A CI/CD pipeline was continuously run [17] to guarantee the development quality of the tool prototype across various tests. Functional tests were also conducted on this basis.

Regarding RQ2, we performed quantitative measurements to demonstrate the configuration changes, conducting 30 tests each using detailed logging and systematic, automated evaluation, as illustrated in the Figure 5. Depending on the presence of features, different containers were started, stopped, and reconfigured. For this, the evaluation involved monitoring the system's dynamic behavior, specifically tracking the lifecycle of containers as they were started, stopped, or reconfigured based on the activated features. Dynamic configuration changes were only made in the FSM and Feature Model entities. Serious structural changes to the application or the interrelationships must be handled in the current implementation by running through the entire pipeline again. All operations and measurements were conducted on an Apple MacBook Pro 2018, equipped with a Quad-Core Intel Core i5-8259U processor running at 2.3 GHz, 16 GB of RAM, and a 500 GB APFS-formatted SSD. The operating system used was Sonoma 14.5. For development and execution, the following integrated development environments (IDEs) were utilized for coding and execution: JetBrains GoLand 2024.1[8], PyCharm Professional 2023.2.1[9], and Visual Studio Code 1.93.0[10].

[7] https://github.com/tliron/puccini

[8] https://www.jetbrains.com/de-de/go/

[9] https://www.jetbrains.com/de-de/pycharm/

[10] https://code.visualstudio.com/

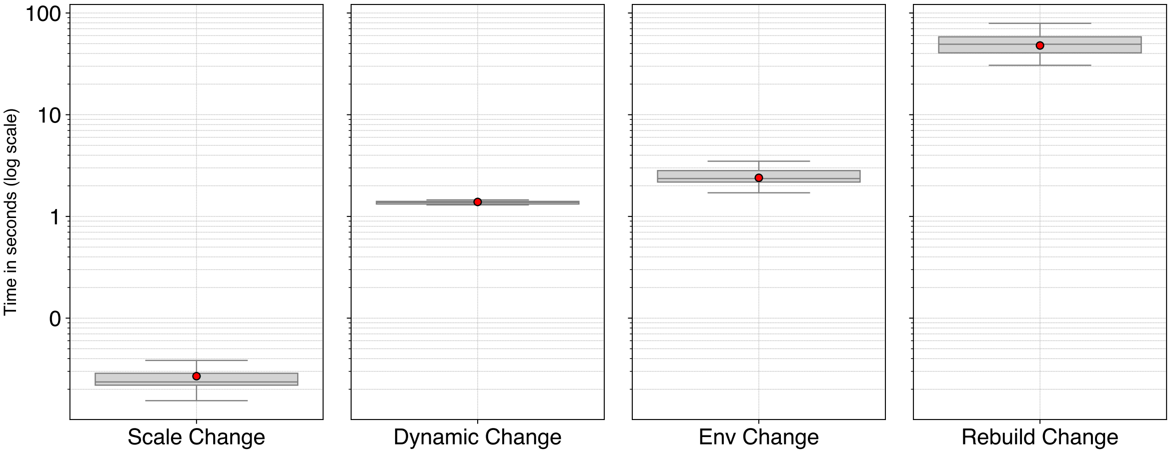


Figure 5: Runtime performance of different change types (in seconds, 30 test runs).

Additionally, Docker version 27.1.1, was used for containerization. Docker was also used to run the onboard single-node Kubernetes cluster with client version 1.30.2 and Kustomize version 5.0.4.

6.2 Results

We discuss the results for RQ1 (correctness) and RQ2 (performance):

RQ1–Correctness. A representative subset of the features and configurations was selected with different variants of features and configurations and checked for functional correctness. This subset contains both the different binding times and the conflicting existence of configuration values. Using the three different conflict resolution strategies described in Section 6.1, we checked the variants for correctness by manual visual inspection of the user interfaces. Additionally, functional tests were performed by applying various product configurations and analyzing system logs, service logs, container settings, and manually inspecting the resulting web applications. The feature-based creation (or non-creation) of containers was performed in five different configurations: (a) enable AngularJS UI, (b) enable AngularJS and NextUI, (c) no UI, (d) enable Postgres database and (e) no database. The containers were created with the correct configurations. However, since the database is a central component of the application and due to dependencies to other containers (e.g., vote requires a database), some containers were started correctly, but without the required functionality. The successful operation could be guaranteed again after adapting the feature model by requiring a database. In total 20 different variants were tested for correctly applied settings in different dimensions and all found to be correct. The choice of these variants ensured that all relevant change types were covered (cf. Table 1). These potential changes will help confirm the accuracy of the settings across various scenarios.

RQ2–Performance. As expected, the performance results of changing different configurations depend on the degree of depth. All test categories illustrated in Figure 5 were performed with the Python service (“vote”) to maintain comparability. Rebuilding the application and its container demands significant resources and, as a result, takes the longest time (mean of 49.6 seconds) to complete. On the other hand, making dynamic changes during runtime, such as altering the color scheme, can be accomplished swiftly (mean of 1.4 seconds). For a change that requires a restart (e.g., setting an environment variable), the mean time was 2.5 seconds, considering that the container has to be stopped and restarted. Changes that are fully orchestrated and do not affect the container’s functionality (e.g., changing scaling) were significantly slower.

7 Discussion and Threats to Validity

The evaluation results regarding the correctness and performance of the voting application are promising. However, several threats to validity must be acknowledged.

Construct Validity. The use of FSMs to define configurations and enable reuse across services is a strength of the approach. However, the evaluation relied on a limited set of configurations, which might not fully represent the complexity of real-world scenarios. The potential for misconfigurations, such as incorrect links between features and services or illegal container settings, also introduces a risk of inaccuracies in assessing the true effectiveness of our approach.

Internal Validity. Performance measurements were conducted under controlled conditions, but factors such as variations in build times, network conditions, and resource availability may have influenced the results. Additionally, while the adaptive architecture supports flexibility, the scope of feature-based settings was limited to Docker Swarm, leaving the influence of other technologies on the results unexplored.

External Validity. The voting application serves as a small but representative example, containing typical elements required for configuring cloud-based systems. However, further validation across diverse applications and configurations is necessary to confirm the applicability to other domains. Still, the demonstration of technological diversity using Grafana, Loki, and Promtail on Kubernetes highlights the potential for broader applicability.

Conclusion Validity. While the results suggest that the approach is effective, any extrapolation to other configurations or environments should account for potential distortions. Furthermore, the accuracy of conclusions depends on the validity of performance measurements, which may have been impacted by inconsistencies in test conditions.

Threats Related to TOSCA Implementation. The TOSCA documentation was instrumental in establishing the foundation of the proposed method. However, understanding specific implementations posed challenges due to missing or outdated information and compatibility issues. Additionally, TOSCA's flexibility is not uniformly supported across platforms and vendors without custom adaptations. As TOSCA 2.0 remains under development, future updates may necessitate further adjustments to implementation strategies. Despite these limitations, TOSCA remains a versatile and promising DSL for defining entities and connections in cloud environments.

8 Related Work

Cloud-based systems require fundamental conceptual and systematic approaches to effectively manage their complexity and heterogeneity [5, 24]. Therefore, Bergmayr et al. compare various cloud modeling languages, confirming TOSCA's capabilities [4]. Pierantoni et al. highlights the necessity of an Application Description Language (ADL) that can be interpreted independently of specific vendors, further justifying the need for a TOSCA-based ADL. In their study on research contributions related to microservices, Di Francesco et al. identified OASIS TOSCA as the most promising framework for defining and managing microservice landscapes, emphasizing its potential as a critical enabler in this domain [7]. Furthermore, to guarantee the required quality of service (QoS), existing reference architectures provide standardized methods for designing, measuring, and monitoring cloud services [12]. Similarly, Brogi et al. describe TOSCA as an emerging standard for modeling, packaging, and managing complex multi-service applications, underlining its relevance for the evolving cloud ecosystem [5, 22].

Feature interactions between software features and feature-to-code mappings have been investigated in different communities. For instance, Zave [32] reported on the problem of feature interactions in continuously evolving systems. Ferber et al. have shown that such dependencies are often difficult to represent in feature models [9]. This issue was investigated by Feichtinger et al. who presented an approach supporting engineers in identifying and resolving inconsistencies between features and the code implementing them [8]. The technique combines feature-to-code mappings, static code analysis, and a variation control system to lift complex code-level dependencies to feature models. When engineers manually maintain feature-to-code mappings, e.g., in annotation-based product lines, it is very challenging to carry out such changes to features while at the same time keeping the mappings consistent [14, 20, 31]. For instance, merging features at a certain point is difficult when done manually since features are mapped to diverse and complex artifacts.

Our approach is related to earlier work by Galindo et al. which integrates different variability models to support a distributed configuration process [10]. However, we go beyond this work by covering configuration changes happening at different binding times, which is essential for cloud-based system, where configuration decisions are often deferred. We also presented mechanisms to update and resolve configuration choices after changes.

Widespread Open Source implementations considering variability in TOSCA applications are *Variability4TOSCA* and *OpenTOSCAVintner*[11]. *OpenTOSCA Vinter* enables comprehensive variability management to create different variants of cloud services. However, in contrast to the implementation described in this paper, *OpenTOSCA Vintner* requires explicitly defined inputs in the TOSCA definition. This concept is more about deciding which value to use based on a single condition. An example, presented by Stötzner et al. [29], distinguishes between Java Community and Java Enterprise Platform when creating a service. As a result, the environment has to be completely recreated at the start of the program for each configuration change, increasing effort and potential delays, especially in dynamic environments [29]. Unlike OpenTOSCA Vintner, which depends on static configurations and necessitates complete system redeployment for any changes, our approach supports dynamic variability management. It allows for the individual resolution of configuration conflicts and enables runtime updates across binding times, offering enhanced flexibility and efficiency in managing dynamic cloud environments.

9 Conclusion and Future Work

Our research explored the integration of the SPL approach within cloud environments, focusing on leveraging OASIS TOSCA alongside SPL. Our preliminary evaluation shows that this combination is useful, with TOSCA's vendor-independent language allowing to model a wide range of cloud entities. Despite some challenges we implemented a flexible approach demonstrating how feature models can be integrated with the TOSCA DSL. Our approach enables the standardized description and management of variability across multiple dimensions.

Despite the importance of variability management in cloud computing, a noticeable gap exists in the literature and practice concerning approaches that effectively combine feature modeling with independent DSLs. This gap highlights the need for innovative solutions to address the complexities of managing feature-based and vendor-neutral configuration variability in cloud environments. Our work represents a step in this direction. By implementing comprehensible configurations through an independent language, our tool-supported approach facilitates the management of cloud features and underscores the potential benefits of integrating feature modeling with DSLs to enhance the adaptability and efficiency of cloud services. Despite progress, establishing direct connections with key vendors remains a hurdle, requiring a consensus on interpretations and an iterative approach. We believe that our method for managing cloud features using an independent configuration language and an adapting resolution method opens up opportunities for additional research on vendor applications, elevating online reconfiguration and enhancing traceability visualization.

[11] https://vintner.opentosca.org/

Acknowledgments

This research was supported in part by the JKU Linz Institute of Technology and the state of Upper Austria [LIT-2023-12-SEE-128].

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation* (1 ed.). Springer Berlin, Heidelberg, Berlin, Heidelberg. XVI, 315 pages. doi:10.1007/978-3-642-37521-7

[2] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. 2022. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering* 27, 2 (2022), 21–23. doi:10.1007/s10664-021-10081-7

[3] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a feature?: a qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 16–25. doi:10.1145/2791060.2791108

[4] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* 51, 1, Article 22 (Feb. 2018), 38 pages. doi:10.1145/3150227

[5] Antonio Brogi, Jacopo Soldani, and PengWei Wang. 2014. TOSCA in a Nutshell: Promises and Perspectives. In *Service-Oriented and Cloud Computing*, Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–186.

[6] Giuseppina Cretella and Beniamino Di Martino. 2015. A semantic engine for porting applications to the cloud and among clouds. *Softw. Pract. Exper.* 45, 12 (Dec. 2015), 1619–1637. doi:10.1002/spe.2304

[7] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2019. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 150 (2019), 77–97. doi:10.1016/j.jss.2019.01.001

[8] Kevin Feichtinger, Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. 2021. Guiding feature model evolution by lifting code-level dependencies. *Journal of Computer Languages* (2021), 101034. doi:10.1016/j.cola.2021.101034

[9] Stefan Ferber, Jürgen Haag, and Juha Savolainen. 2002. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Software Product Lines (Lecture Notes in Computer Science)*, Gary J. Chastek (Ed.). Springer Berlin Heidelberg, 235–256.

[10] José A. Galindo, Deepak Dhungana, Rick Rabiser, David Benavides, Goetz Botterweck, and Paul Grünbacher. 2015. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Information and Software Technology* 62, 1 (11 Feb. 2015), 78–100. doi:10.1016/j.infsof.2015.02.002

[11] Amit Gera and Cathy H Xia. 2011. Learning curves and stochastic models for pricing and provisioning cloud computing services. *Service Science* 3, 1 (2011), 99–109.

[12] Ximena Guerron, Silvia Abrahão, Emilio Insfran, Marta Fernández-Diego, and Fernando González-Ladrón-De-Guevara. 2020. A Taxonomy of Quality Metrics for Cloud Services. *IEEE Access* 8 (2020), 131461–131498. doi:10.1109/ACCESS.2020.3009079

[13] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: mapping features to models. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 943–944. doi:10.1145/1370175.1370199

[14] Daniel Hinterreiter, Lukas Linsbauer, Kevin Feichtinger, Herbert Prähofer, and Paul Grünbacher. 2020. Supporting feature-oriented evolution in industrial automation product lines. *Concurrent Engineering* 28, 4 (2020), 265–279. doi:10.1177/1063293X20958930

[15] Michał Jagiełło, Marian Rusek, and Waldemar Karwowski. 2019. Performance and Resilience to Failures of an Cloud-Based Application: Monolithic and Microservices-Based Architectures Compared. In *Computer Information Systems and Industrial Management*, Kashif Saeed, Rituparna Chaki, and Vladimir Janev (Eds.). Lecture Notes in Computer Science, Vol. 11703. Springer International Publishing, Cham, 445–456. doi:10.1007/978-3-030-28957-7_37

[16] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. 2019. Container Orchestration Engines: A Thorough Functional and Performance Comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE, Shanghai, China, 1–6. doi:10.1109/ICC.2019.8762053

[17] Stefan Kehrer, Florian Riebandt, and Wolfgang Blochinger. 2019. Container-Based Module Isolation for Cloud Services. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, San Francisco East Bay, California, USA, 177–17709. doi:10.1109/SOSE.2019.00032

[18] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. 2005. Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange* (San Diego, California). Association for Computing Machinery, New York, NY, USA, 55–59. doi:10.1145/1117696.1117708

[19] Paul Lipton, Derek Palma, Matt Rutkowski, and Damian A. Tamburri. 2018. TOSCA Solves Big Problems in the Cloud and Beyond! . *IEEE Cloud Computing* 5, 02 (March 2018), 37–47. doi:10.1109/MCC.2018.022171666

[20] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time. In *Proc. 20th ACM SIGPLAN Int'l Conference on Generative Programming: Concepts and Experiences* (Chicago, IL, USA). ACM, New York, NY, USA, 2–15. doi:10.1145/3486609.3487195

[21] Sam Newman. 2021. *Building Microservices* (2nd ed.). O'Reilly Media, Inc., Sebastopol, CA.

[22] OASIS. 2024. TOSCA - Topology and Orchestration Specification for Cloud Applications Version 2.0.

[23] Matthew D O'Connell, Cameron T Druyor, Kyle B Thompson, Kevin E Jacobson, William K Anderson, Eric J Nielsen, Jan-Reneé Carlson, Michael A Park, William T Jones, Robert T Biedron, et al. 2018. T-infinity: The Dependency Inversion Principle for Rapid and Sustainable Multidisciplinary Software Development. In *AIAA 2018 Aviation Meeting*. American Institute of Aeronautics and Astronautics (AIAA), Atlanta, GA, United States, 5–10. doi:10.2514/6.2018-3856

[24] Gabriele Pierantoni, Tamas Kiss, Gabor Terstyanszky, James DesLauriers, Gregoire Gesmier, and Hai-Van Dang. 2020. Describing and processing topology and quality of service parameters of applications in the cloud. *Journal of Grid Computing* 18 (2020), 761–778.

[25] Rajiv Ranjan, Boualem Benatallah, Schahram Dustdar, and Michael P. Papazoglou. 2015. Cloud Resource Orchestration Programming: Overview, Issues, and Directions. doi:10.1109/MIC.2015.20

[26] Felix Richter. 2024. Cloud Infrastructure Market–Amazon Maintains Cloud Lead as Microsoft Edges Closer. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

[27] Ayob Sether. 2016. Cloud Computing Benefits. *SSRN Electronic Journal* (2016), 5–17. doi:10.2139/ssrn.2781593

[28] Norbert Siegmund, Nicolai Ruckel, and Janet Siegmund. 2020. Dimensions of software configuration: on the configuration context in modern software development. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 338–349. doi:10.1145/3368089.3409675

[29] Miles Stötzner, Uwe Breitenbücher, Robin D. Pesl, and Steffen Becker. 2024. Managing the Variability of Component Implementations and Their Deployment Configurations Across Heterogeneous Deployment Technologies. In *Cooperative Information Systems*, Mohamed Sellami, Maria-Esther Vidal, Boudewijn van Dongen, Walid Gaaloul, and Hervé Panetto (Eds.). Springer Nature Switzerland, Cham, 61–78.

[30] Oliver Thomas. 2006. Understanding the Term Reference Model in Information Systems Research: History, Literature Analysis and Explanation. In *Business Process Management Workshops*, Christoph J Bussler and Armin Haller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 484–496.

[31] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. 2010. Flexible and scalable consistency checking on product line variability models. In *Proc. IEEE/ACM Int'l Conference on Automated Software Engineering*. ACM, New York, NY, USA, 63–72.

[32] Pamela Zave. 1993. Feature interactions and formal specifications in telecommunications. *Computer* 26, 8 (Aug 1993), 20–28. doi:10.1109/2.223539

The Kubernetes variability model

Synthesizing variability from the K8s API Documentation: A case study

José Miguel Horcas
ITIS Software, Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain
horcas@uma.es

Mercedes Amor Pinilla
ITIS Software, Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain
map@uma.es

Lidia Fuentes
ITIS Software, Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain
lfuentes@uma.es

Abstract

Kubernetes is widely adopted across the Cloud, Edge and IoT Continuum to orchestrate containerized applications efficiently, from centralized clouds to edge devices and IoT endpoints. Managing configuration variability in platforms like Kubernetes remains challenging due to a wide range of options that, while documented, are often implicit in configuration files. This complexity limits the use of many features, with most configurations employing only a subset of available options, forcing developers to study extensive manuals to understand configurable parameters and their variants. This paper presents a synthesized variability model for Kubernetes, derived directly from its official documentation rather than from existing configurations. The resulting feature model captures the configuration space and its constraints, enabling the automatic generation of tailored, consistent configurations that reduce manual effort and ensure compatibility across platform versions. We validate the model against over 250,000 real-world configurations to confirm its effectiveness.

CCS Concepts

• **Software and its engineering** → **Software product lines**; *Software configuration management and version control systems*; Documentation; Software as a service orchestration system.

Keywords

Configuration, documentation, Kubernetes, feature model, variability synthesis, manifest files, YAML

ACM Reference Format:
José Miguel Horcas, Mercedes Amor Pinilla, and Lidia Fuentes. 2025. The Kubernetes variability model: Synthesizing variability from the K8s API Documentation: A case study. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3715340.3715440

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715440

1 Introduction

Software containerization and orchestration platforms such as *Docker* and *Kubernetes* have revolutionized how applications are deployed and managed across diverse environments from centralized clouds to edge devices and Internet-of-Thing (IoT) endpoints [6]. Deploying an application in the Cloud, Edge and IoT (CEI) Continuum [15, 23] requires configuring multiple services with varying setups based on application needs [34, 43].

Kubernetes (often abbreviated as K8s) [5, 37] has become widely adopted across the CEI Continuum to orchestrate containerized applications. This platform offers a high degree of configurability, enabling developers to fine-tune deployment parameters to suit various application needs. However, this flexibility comes with the challenge of managing the vast variability inherent in configuration options [8, 38]. Configuration files (aka *manifest files* in K8s), typically written in YAML [26], provide the means to specify deployment parameters, but the options and constraints are often implicit, requiring manual exploration of the official documentation to fully understand the available valid configurations [5, 37].

Despite reverse engineering techniques have been widely applied to extract variability from existing configurations [2, 4, 29, 31, 42], this approach is not well-suited to K8s for several reasons. First, K8s' configuration principles advocate for simple, minimal configurations that avoid errors by omitting default parameters in manifest files [8, 38]. As a result, these files lack many of K8s' configurable parameters, complicating the reverse engineering process. Second, K8s includes complex configuration options beyond Boolean flags, requiring advanced variability modeling concepts — such as typed features (e.g., numerical features [35]), feature cardinalities [7], attributes [28], and complex constraints [24] (e.g., arithmetical) — that current reverse engineering techniques of feature modes do not take into account [22, 31]. Third, while reverse-engineered feature models can capture the variability in a specific set of configurations, they often lack generalizability. Furthermore, these models tend to be intricate and challenging for domain engineers to interpret, in terms of their hierarchical structure and constraint definitions, presenting issues similar to the *black box problem* in explainable Artificial Intelligence (AI) techniques [47].

In this paper, we propose synthesizing a variability model of K8s directly from its official API Reference documentation [46]. By systematically analyzing the documentation, we aim to construct an explainable feature model that represents the full configuration space, providing a structured representation of the K8s variability, which can be used to automate the generation of valid configuration files tailored to specific deployment scenarios [27]. Furthermore, our approach yields an explainable feature model specified in the Universal Variability Language (UVL) [44], incorporating the latest UVL extensions [45] to support advanced variability modeling concepts of K8s. The contributions of this paper are:

- A manual process for synthesizing a variability model from the official API documentation of K8s (Section 3).
- A comprehensive and explainable feature model of K8s in UVL supporting advanced variability modeling concepts (Section 4).
- A process to extract over 250,000 real K8s configurations from existing repositories to validate the feature model (Section 5.2).
- A set of variable templates for the automatic generation of K8s manifest files from the feature model, reducing the likelihood of errors caused by misconfiguration (Section 5.3).

Although variability models of other large intensive-variability systems such as the Linux Kernel [9, 11, 33, 41] have been extracted, to the best of our knowledge, no variability model of K8s has been developed to date [36, 43] (Section 6). With this work, we aim to provide the software product line community with a new case study of an intensive-variability system.

2 Upbringing Kubernetes

Kubernetes (K8s) [5, 37] is an open-source platform for automating the deployment, scaling, and management of containerized applications. K8s achieves this by grouping containers into logical units called *Pods* that are scheduled and managed by the K8s cluster. The platform provides a robust system for self-healing, where it automatically replaces failed containers, balances workloads across nodes, and scales applications based on demand.

A fundamental concept in K8s is the *Kubernetes Resource Object* (KRO), which refers to the basic unit of configuration in the system. KROs can represent various components such as *Pods*, *Containers*, *Services*, *Deployments*, and *ConfigMaps*, among others, each of which defining different aspects of application deployment and management. The configurations of these components are specified in files known as *manifests*, which declare the desired state of each resource along with its associated parameters.

2.1 The K8s manifest: Configuring Kubernetes

The *Kubernetes manifest* is a configuration file that defines the resources to be created, managed, or updated within a K8s cluster. This file is typically written in YAML [26] and specifies how K8s components (e.g., Pods, Deployments, Services) are configured.

Listing 1 shows a basic YAML manifest example that instructs K8s to create a Pod named `my-pod` with a container running the *NGINX* web server, exposing port 80. The manifest outlines key details about the resources, including the `apiVersion` (i.e., the version of the Kubernetes API used to define the object), the `kind` (i.e., the type of resource, such as Pod, Service, or Deployment), the `metadata` (i.e., information about the resource such as its name, labels, and annotations), and the `spec` (i.e., the resource specification, containing the parameters and configurations for its operation). For instance, in a Pod, the `spec` component would include details about the containers it runs, their images, exposed ports, and volumes, among other configuration options. K8s then ensures that the cluster matches that declared state by managing and adjusting resources as needed.

Listing 1: YAML manifest for configuring a Pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      ports:
        - containerPort: 80
```

2.2 The K8s API Reference documentation

The official *Kubernetes API Reference documentation* (K8s API) [46] is a comprehensive guide that details the Kubernetes API. It is systematically organized to provide detailed and structured information about the concepts, resources, and operations, available within the K8s ecosystem. A snippet of the API documentation is shown in Figure 1. The K8s API is divided into key sections, including an overview of the API structure, descriptions of resource types and their configuration options, as well as guidelines for versioning and deprecation. Each resource is presented with its specifications, fields, and operational methods, allowing users to configure or interact with the K8s cluster efficiently.

The relationship between the K8s manifest and the official K8s API is that the manifest serves as the practical, declarative implementation of the API specifications outlined in the documentation. For instance, the fields `apiVersion`, `kind`, `metadata`, and `spec` in the manifest (Listing 1) directly correlate with the API documentation, ensuring the resources are correctly configured and managed.

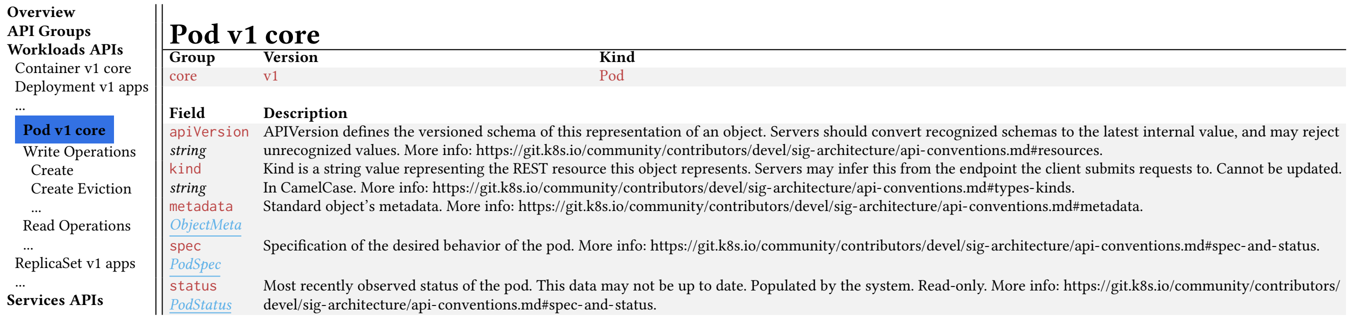
3 Methodology: Variability synthesis process

This section describes the process of extracting and synthesizing a variability model from the K8s API documentation. This approach aims at formalizing the configuration options and their variability into a feature model, providing a structured representation of the deployment parameters and constraints inherent to K8s.

The process of extracting variability from the K8s API involves three main steps following an incremental methodology, which is illustrated in Figure 2. The initial step is the **Documentation exploration**, followed by the **Synthesis of the feature model** and concluding with the **Model validation**. The **Synthesis of the feature model** step is refined into three sub-steps: **Classification of features**, **Variability analysis**, and **Rule definition**. Each of these steps contributes to the systematic construction of a variability model that captures the essence of the configuration space described in the documentation. This is an iterative, non-sequential process, that allows transitioning between any steps and sub-steps at any moment. The process is incremental because, in each iteration, the feature model is refined by adding new features, relations and constraints from the documentation.

Documentation exploration: Systematically navigating the K8s API to identify configuration options, default values, and possible variations for each parameter.

Synthesis of the feature model: Structuring the identified configuration options into a feature model, including mandatory and optional features, as well as grouped features (or, xor, cardinality



Overview
API Groups
Workloads APIs
Container v1 core
Deployment v1 apps
...
Pod v1 core
Write Operations
Create
Create Eviction
...
Read Operations
...
ReplicaSet v1 apps
...
Services APIs
...

Pod v1 core

Group	Version	Kind
core	v1	Pod

Field	Description
apiVersion *string*	APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources.
kind *string*	Kind is a string value representing the REST resource this object represents. Servers may infer this from the endpoint the client submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds.
metadata *ObjectMeta*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata.
spec *PodSpec*	Specification of the desired behavior of the pod. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status.
status *PodStatus*	Most recently observed status of the pod. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status.

Figure 1: Snippet of the main components of a Pod resource in Kubernetes.

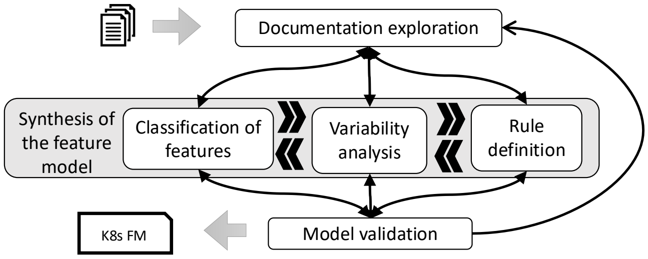

Documentation exploration
Synthesis of the feature model
Classification of features
Variability analysis
Rule definition
K8s FM
Model validation

Figure 2: Iterative process for extracting variability from K8s.

groups), features cardinalities [7] (aka multi-features or clonable features), and constraints between the features.

Classification of features: Categorizing the configuration options into relevant groups based on their function and dependencies within the Kubernetes ecosystem.

Variability analysis: Identifying variation points and variants (i.e., where multiple options exist for a given configuration parameter) and understanding the relations between options (e.g., typed features, default values, mutual exclusivity or conditional dependencies).

Rule definition: Formulating formal rules based on the variability model to ensure the correctness and validity of the generated configurations. This includes the definition of the cross-tree constraints associated with the identified dependencies between features.

Model validation: Testing the synthesized feature model against real K8s configurations to validate its completeness and accuracy.

The process begins by conducting a thorough exploration of the K8s API documentation [46], which provides detailed descriptions of every resource type, its configurable fields, and associated operations (see Figure 1). A KRO has several primary components (e.g., `apiVersion`, `kind`, `metadata`, `spec`) that describe the resource and define its behaviour. The hierarchical structure of the K8s API reflects how elements such as `metadata` and `spec` further link to subobjects like `ObjectMeta` and `PodSpec` (in case of a Pod resource), revealing nested fields and dependencies. Each kind of KRO (e.g., Pod, Container, Deployment) has its own specification with specific sub-fields and set of possible values and constraints. For instance, the specification of the Pod resource includes configuration options such as `containers`, `restartPolicy`, and `volumes`, among others, each of which has its own set of possible values and constraints that may be different from the same specification options of other KRO. In the exploration process, each resource's sub-fields are expanded systematically in a depth-first exploration, ensuring that every child field is fully processed before progressing to the next.

Note that not all fields correspond to configuration options. There are non-configurable fields of a KRO that appear in the K8s API as "`Read-only`" fields. For instance, the `status` field (shown below the `spec` in Figure 1) specifies the most recently observed status of the KRO, and their values are populated by the system. Our feature model does not include these kinds of characteristics because they cannot be specified in a configuration manifest.

In the following, we provide a detailed description of the synthesis performed within the K8s API to obtain the feature model.

4 Synthesis of the feature model

This is the main step of our approach where we detail the design decisions considered to build the feature model. We use the *Universal Variability Language* (UVL) [44] to specify the feature model and adhere to its official grammar [45] to support the required extensions beyond boolean features. We follow a process of featurization in which each configurable field of the K8s API is represented in the model with a feature. Listings 2 and 3 show two excerpts of the model, while the complete one is available online (Section 5.1).

4.0.1 ***Classification of features.*** The feature model was constructed by classifying the features identified during documentation exploration. Features were grouped according to their functionality and dependencies within K8s. The feature model hierarchy mimics the K8s API structure, adapting it to the UVL grammar (e.g., features cannot be duplicated). Assuming that the feature model encodes any configuration of any KRO in Kubernetes, we start defining a root feature "`Kubernetes Resource Object`" (line 2 in Listing 2).

There are a number of primary fields that are shared across many KRO, being only the `apiVersion` and `kind` common to all types of KROs. For instance, while most of the KROs such as pods, jobs, services, and deployments, have the `apiVersion`, `kind`, `metadata`, and `spec` fields, other types of KROs such as secrets or events have additional fields (e.g., `data`). These primary fields form the basis of the feature hierarchy in the variability model. Thus, as top features of the model, we define the primary fields `apiVersion` (line 4) and `kind` (line 18) as mandatory features, while the other primary fields (e.g., `metadata`, `spec`, `data`) as defined as optional children of the root feature (lines 34–42).

Then, some primary fields have their own specification based on the KRO type. Thus, we create a specific branch for each KRO below each feature representing a primary field. For instance, `PodSpec`

Listing 2: UVL excerpt of the top features.

```
1  features:
2    "Kubernetes Resource Object" {abstract}
3      mandatory
4        apiVersion {doc 'Versioned schema of this representation of an object.'}
5          mandatory
6            Group {abstract, doc 'Refers to the API group that organizes related resources.'}
7              alternative
8                GROUP_authentication
9                GROUP_core
10               GROUP_events
11               ...
12           Version {abstract, doc 'Version of the API group that the resource should use.'}
13             alternative
14               VERSION_v1
15               VERSION_v1beta1
16               VERSION_v1alpha1
17               ...
18       kind {doc 'The REST resource this object represents.'}
19         alternative
20           Workloads_APIs {abstract, doc 'Set of API resources that manage and run workloads.'}
21             alternative
22               KIND_Container
23               KIND_Deployment
24               KIND_Pod
25               ...
26           Service_APIs {abstract, doc 'Manage access to applications running in a cluster.'}
27           Config_storage_APIs {abstract, doc 'Manage configurations, secrets, and storage.'}
28           Metadata_APIs {abstract, doc 'Manage the metadata associated with resources.'}
29           Cluster_APIs {abstract, doc 'Manage the overall configuration, state, and resources.'}
30             alternative
31               KIND_APIService
32               KIND_TokenRequest
33               ...
34     optional
35       metadata {doc 'Standard metadata of the object.'}
36       spec {doc 'Specification of the desired behavior of the object.'}
37         optional
38           DeploymentSpec {abstract, doc 'Specification of the behavior of the Deployment.'}
39           PodSpec {abstract, doc 'Specification of the behavior of the Pod.'}
40           ...
41       data
42       ...
43 constraints:
44   // Compatibility dependencies
45   GROUP_core => VERSION_v1
46   GROUP_authentication => VERSION_v1 | VERSION_v1beta1 | VERSION_v1alpha1
47   (KIND_Container | KIND_Pod | ...) => GROUP_core
48   (KIND_TokenRequest | ...) => GROUP_authentication
49   ...
50   // Artificial organizational constraints
51   KIND_Pod => PodSpec
52   KIND_Deployment => DeploymentSpec & PodSpec
53   PodSpec => KIND_Pod | KIND_Deployment
54   DeploymentSpec => KIND_Deployment
55   DeploymentSpec => PodSpec
56   ...
```

(line 39) represents an entire branch defining all the configuration options for a *Pod*, while `DeploymentSpec` (line 38) does so for a *Deployment*. Listing 3 shows an excerpt of the `PodSpec` branch.

We follow this rationale to build the structure and branches of the feature model because some fields are common and others are specific to a KRO. Moreover, some fields (shared or specific) may have dependencies with other fields, even with fields of other KRO (see Section 4.0.3). For instance, *containers* are only ever created within the context of a *Pod*. Thus, we define a sub-branch `PODSPEC_containers` for the specification of the containers (line 4 in Listing 3). This is shown in the K8s API as a `Warning!` in the specification of a container, which is difficult to be aware of and synthesize without iteratively exploring the whole documentation.

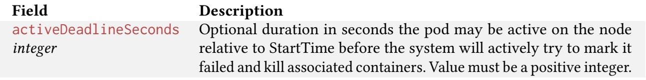
Listing 3: UVL excerpt of the PodSpec feature.

```
1  ...
2  PodSpec {abstract, doc 'Specification of the behavior of the Pod.'}
3    mandatory
4      PODSPEC_containers cardinality [1..*] {doc 'List of containers belonging to the pod.'}
5        mandatory
6          String CONTAINERS_name {doc 'Name of the container specified as a DNS_LABEL.'}
7          ...
8        optional
9          String CONTAINERS_image {doc 'Container image name.'}
10         CONTAINERS_ports cardinality [1..*] {doc 'Ports to expose from the container.'}
11           mandatory
12              Integer CONTAINERS_PORTS_containerPort {doc 'Port number for the pod IP.'}
13           optional
14             String CONTAINERS_PORTS_hostIP {doc 'Host IP to bind the external port.'}
15             Integer CONTAINERS_PORTS_hostPort {doc 'Port number to expose on the host.'}
16             ...
17   optional
18     Integer activeDeadlineSeconds {doc 'Duration the pod is active before killing containers.'}
19     PODSPEC_hostNetwork {doc 'Host networking requested for this pod.'}
20     enableServiceLinks {default true, doc 'Inject services information into pods env variables.'}
21     restartPolicy {doc 'Restart policy for all containers within the pod.'}
22       alternative
23         Always {default, doc 'Automatically restarts the container after any termination.'}
24         OnFailure {doc 'Only restarts the container if it exits with error (non-zero exit status).'}
25         Never {doc 'Does not automatically restart the terminated container.'}
26     ...
27 constraints
28   // Domain constraints
29   activeDeadlineSeconds > 0
30   CONTAINERS_PORTS_hostPort > 0 & CONTAINERS_PORTS_hostPort < 65535
31   PODSPEC_hostNetwork => CONTAINERS_PORTS_hostPort ==
         CONTAINERS_PORTS_containerPort
32   ...
```

Naming convention for features. When naming the features in the model, an effort was made to follow a consistent nomenclature to avoid duplicate names. If a feature appeared more than once in the model, it was assigned an uppercase prefix referencing one of its parent features (typically the most representative one). In cases where this was not sufficient to prevent duplicates, additional prefixes were added until the name was unique. For instance, the `name` field appears in several elements (e.g., to identify containers). Thus we designated that field under the container feature as `CONTAINERS_name` (see line 6 in Listing 3).

4.0.2 ***Variability analysis.*** During variability analysis, the features are further refined into types, including Boolean, Integer, and String. Where possible, the variability model includes selection groups (i.e., or-groups) when more than one option can be selected in a field, and alternative groups (i.e., xor-groups) for mutually exclusive options, reflecting the constraints specified in the documentation. Also, feature cardinalities are considered when more than one instance of an object can be configured. Finally, features are enriched with feature attributes to include further information.

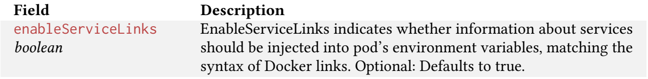
Parent-child relationships. Regarding the feature relationships (i.e., *mandatory*, *optional*, *or*, or *alternative*), the descriptions of each field in the K8s API are carefully considered. Typically, if a configurable option is optional or required (i.e., mandatory) to be defined, this is explicitly stated in the field's description. For instance, the description of the `activeDeadlineSeconds` field (see Figure 3) indicates that this is an optional configuration option. Similarly, if it is mandatory to choose only one possible value from a set (i.e., mutually exclusive attributes), the *alternative* relation is used. The *or-group* relation is applied when one or more child features can be selected. If no clear indication is provided, or if there is uncertainty, the feature is defined as optional.



Field	Description
activeDeadlineSeconds *integer*	Optional duration in seconds the pod may be active on the node relative to StartTime before the system will actively try to mark it failed and kill associated containers. Value must be a positive integer.

Figure 3: An integer field specification.

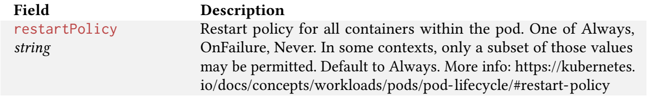
Typed features. In configuration options, Boolean features typically represented flags or switches, while Integer and String features captured more complex configurations such as timeout durations or image names. In the K8s API, some fields require specifying a specific value for its configuration. Data types (Boolean, Integer, String) are derived directly from the documentation (i.e., the data type is shown below each field). For instance, activeDeadlineSeconds is an Integer field (see Figure 3) which we synthesize in the model as an Integer feature (line 18 in Listing 3). We use Boolean features as default when types are not specified. Note also that Boolean fields can specify a default value. For example, the enableServiceLinks is a Boolean optional field whose default value is true. When the default value of a Boolean field is true, it is not necessary to explicitly include that option in the K8s manifest following the K8s principles of simplifying configuration files. To account for this in the feature model, we define the corresponding feature with a {default true} attribute (line 20), which will be considered later when generating configurations.



Field	Description
enableServiceLinks *boolean*	EnableServiceLinks indicates whether information about services should be injected into pod's environment variables, matching the syntax of Docker links. Optional: Defaults to true.

Figure 4: A Boolean field with default value.

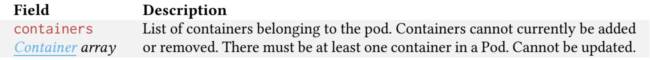
Discretization of feature's values. In addition, when a field has a String or Integer type, and its possible values are limited to a few variants, we featurize the corresponding values: the type of the affected feature is changed to Boolean, and as many children feature as possible variants are added with an alternative relationship. For instance, the restartPolicy String field (see Figure 5) can only take as values "Always", "OnFailure", or "Never". Thus, we synthesize such information in the model with the four features depicted in Listing 3 (lines 21–25). We use a default attribute to indicate that "Always" is the default option in this group (line 23).



Field	Description
restartPolicy *string*	Restart policy for all containers within the pod. One of Always, OnFailure, Never. In some contexts, only a subset of those values may be permitted. Default to Always. More info: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy

Figure 5: A string field with a limited set of values.

Feature cardinalities. Cardinality is assigned to a feature when the documentation specifies that the attribute is an array of elements under the field's name. This implies that the affected object can be instantiated multiple times and its sub-fields need to be configured for each instance. In such a case, we define that field as a feature cardinality (aka multi-feature, clonable feature) [7] with a multiplicity of [1..*]. For example, the containers array field (Figure 6) allows configuring the list of containers belonging to a pod. We synthesize this in the variability model with a feature cardinality PODSPEC_constainers depicted in Listing 3 (line 4).



Field	Description
containers *Container array*	List of containers belonging to the pod. Containers cannot currently be added or removed. There must be at least one container in a Pod. Cannot be updated.

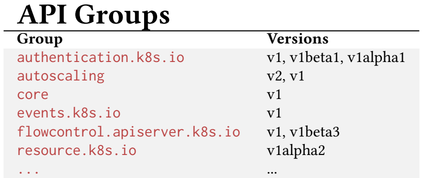
Figure 6: An array field specification.

Feature attributes. The feature model is enriched with the following feature attributes:

- *Abstract features:* We use the {abstract} attribute to differentiate abstract and concrete features. Abstract features do not appear explicitly in the K8s API but are necessary for organizing the hierarchy of the feature model more clearly.
- *Default values:* We use the {default «value»} and the {default} attributes to include a default value within a typed feature or within an alternative group, respectively.
- *Documentation:* We use {doc «string»} to include a brief description of the feature. This information is extracted from the K8s API and helps users understand the purpose of the feature.
- *Versioning:* We use {added_version «version»} to indicate that the feature has been added in a specific version of K8s.
- *Unavailable features:* We mark a feature with a {unavailable} attribute to indicate that such a feature has a missing configuration specification. This occurs when K8s releases new features and its documentation is currently under development and is not completed (e.g., the description of the field is empty).
- *Incomplete features:* We mark a feature as {incomplete} to indicate that such a feature has not yet been fully synthesized due to the complexity of modelling it in a feature model.

4.0.3 ***Rule definition.*** Once the features are classified and analyzed, the next step is to define the formal rules that govern the restrictions between them, that is, the cross-tree constraints. We distinguish three types of rules in the K8s feature model: the compatibility rules between KROs, the rules imposed by the domain, and the rules defined for organizational purposes. The first two types of rules are derived from the K8s API documentation, where constraints and dependencies between fields are explicitly described. The rules will be translated to cross-tree constraints in our feature model ensuring that the configurations generated from the feature model are valid and adhere to the documented rules.

Compatibility constraints. In K8s, the apiVersion and kind core fields define the compatibility across KROs. Each kind of KRO is classified in an API Group that can be compatible with specific versions of the K8s. This information is available in a compatibility table available in the K8s API overview (see K8s API 7). For example, those KROs belonging to the core group are compatible with version v1, while the KROs of the authentication group are compatible with versions v1, v1beta1, and v1alpha1. In our feature model, this information is expressed by defining first the valid groups and versions (lines 4–17 in Listing 2), and then the associated cross-tree constraints such as GROUP_core => VERSION_1 and GROUP_authentication => VERSION_v1 | VERSION_v1beta1 | VERSION_v1alpha1 (lines 45–46).



API Groups

Group	Versions
authentication.k8s.io	v1, v1beta1, v1alpha1
autoscaling	v2, v1
core	v1
events.k8s.io	v1
flowcontrol.apiserver.k8s.io	v1, v1beta3
resource.k8s.io	v1alpha2
...	...

Figure 7: Excerpt of the API Groups and their versions.

Furthermore, information on the group to which each KRO belongs (i.e., the relation between kind and group) is available within each KRO specification (see Pod v1 core in Figure 1). For example, *Containers* and *Pods* belong to the core group, while *TokenRequest* belongs to the authentication group. This leads to the constraints (KIND_Container | KIND_Pod) => GROUP_core (line 47) and KIND_TokenRequest => GROUP_authentication (line 48), respectively. Note that, wherever possible, we group together those constraints that share a common right-hand implication to simplify the model and reduce the number of constraints.

Domain constraints. In the description of the fields, it is often mentioned whether selecting a specific value affects the options of another field, or whether a number must fall within a certain range, among other conditions. These are constraints imposed by the K8s domain. For example, the field `hostNetwork` of a *Pod* describes that if it is set, the ports that will be used must be specified in the associated container (see Figure 8 and Figure 9). Moreover, we can see from the description of the field `hostPort` (Figure 9) that if the field `hostNetwork` is specified in the Pod, its value must match the field `containerPost`; and this value must be a valid port number. This leads to the following constraints in Listing 3: `PODSPEC_hostNetwork => CONTAINERS_PORTS_hostPort == CONTAINERS_PORTS_containerPort` (line 31), and `CONTAINERS_PORTS_hostPort > 0 & CONTAINERS_PORTS_hostPort < 65535` (line 30).

Field	Description
hostNetwork *boolean*	Host networking requested for this pod. Use the host's network namespace. If this option is set, the ports that will be used must be specified. Default to false.

Figure 8: A field describing a dependency.

Field	Description
containerPort *integer*	Number of port to expose on the pod's IP address. This must be a valid port number, 0 < x < 65536.
hostIP *string*	What host IP to bind the external port to.
hostPort *integer*	Number of port to expose on the host. If specified, this must be a valid port number, 0 < x < 65536. If HostNetwork is specified, this must match ContainerPort. Most containers do not need this.

Figure 9: Dependencies between Pods and ContainerPort.

Hierarchical constraints. There are some constraints that, while not explicitly stated in the K8s API, are logically inferred from the definition of the feature model due to its hierarchical organization. For example, the constraint `KIND_Pod => PodSpec` (line 51 in Listing 2) indicates that the kind of KRO *Pod* needs to define a *Pod* specification. The constraint `DeploymentSpec => PodSpec` (line 55) arises from the fact that when defining a *Deployment*, it is mandatory to also define a *Pod*. Another example is the constraint `KIND_Deployment => DeploymentSpec & PodSpec` (line 52), meaning that if the object type is a *Deployment*, we must define both the `DeploymentSpec` (to define the *Deployment*'s behaviour) and the `PodSpec` (to define the behaviour of the *Pod* that will be created alongside the deployment).

5 Model validation

To ensure the accuracy and completeness of the synthesized feature model, we performed a validation process that involves (1) testing the model against real K8s configuration, verifying that those configurations are valid according to the feature model's relations and constraints, and (2) generating K8s manifest files with valid configurations from the feature model. We first present the feature model and then answer the following research question (RQ):

RQ1 *To what extent does the synthesized variability model cover real-world Kubernetes configurations and detect configuration errors?*

Manual effort. The total manual effort involved in the main tasks is calculated at 2.75 person-months, reflecting the complexity of the variability synthesis and its validation. Concretely, 216 hours (1.35 person-months) spent on the synthesis of the K8s feature model, 116 hours (0.73 person-months) in the development of scripts to extract configurations from real K8s manifests, and 108 hours (0.68 person-months) in the creation of K8s configuration templates. Existing similar works of synthesizing variability do not provide a detailed breakdown of the manual effort [1, 12, 14, 18, 40], making direct comparisons challenging.

5.1 Open science artifact

Following open science best practices, our K8s feature model and all associated resources are publicly available online:

- Zenodo: **https://doi.org/10.5281/zenodo.14036131**
- GitHub: **https://github.com/CAOSD-group/kubernetes_fm**

The repository contains the K8s feature model, its full characterization with all structural and analytical metrics, as well as all the resources and scripts developed in Python to collect real K8s configurations (Figure 10), and the templates (Listing 4) to generate new configurations from the feature model.

5.2 Extracting real K8s configurations

Figure 10 illustrates the process for extracting and validating real K8s configurations. This process involves filtering over 225,000 YAML files from 17,483 GitHub repositories to identify valid configurations according to the feature model's relations and constraints. We conducted three distinct searches through the official *GitHub REST API*, using the keywords "`kubernetes`", "`kubernetes manifest`", and "`kubernetes validators`", to obtain a diverse set of YAML files from January 2024. Of the 225,072 YAML files retrieved, 87% were valid K8s manifest files, while 13% were either non-K8s manifest or contained syntax errors, preventing configuration extraction. On average, each GitHub repository contained 23 YAML files, with the largest repository containing 11,676 files. Each manifest file averaged 1.2 configurations, though some contained up to 615. A configuration here refers to a KRO configuration, such as for *Pods*, *Deployments*, or *Services*. Ultimately, we extracted 282,401 configurations, of which 95% (267,561 configurations) were valid, meaning they satisfied the feature model constraints, while 5% (14,840 configurations) were non-satisfiable due to misconfiguration.

5.3 Generating valid K8s configurations

Listing 4 presents an excerpt from a K8s manifest template for generating configurations using a code generation approach with the *Jinja2* template engine. Variable elements are defined within double curly braces and are highlighted in bold, representing variation points in the feature model that will be substituted with the selected variants in the final configurations. Jinja2 provides a robust set of directives for code generation, including variable substitution, assignment, and control structures such as *if/elif/else* statements, *for-loops*, *macros*, and *blocks*. For example, the `for-loop` in Listing 4 (lines 12–23) iterates over the containers within a *Pod*, corresponding to the multi-feature `PODSPEC_containers[1..*]` in the variability model. Additionally, typed features, such as the Integer feature

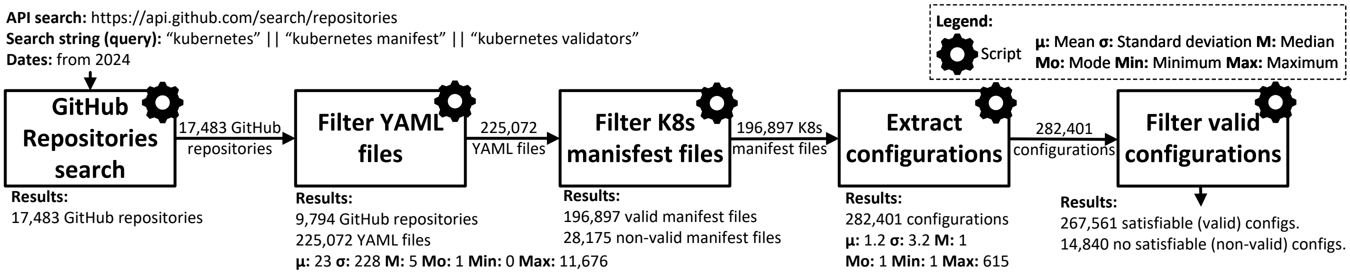

API search: https://api.github.com/search/repositories
Search string (query): "kubernetes" || "kubernetes manifest" || "kubernetes validators"
Dates: from 2024
Legend:
Script
μ: Mean σ: Standard deviation M: Median
Mo: Mode Min: Minimum Max: Maximum
GitHub Repositories search
17,483 GitHub repositories
Filter YAML files
225,072 YAML files
Filter K8s manisfest files
196,897 K8s manifest files
Extract configurations
282,401 configurations
Filter valid configurations
Results:
17,483 GitHub repositories
Results:
9,794 GitHub repositories
225,072 YAML files
μ: 23 σ: 228 M: 5 Mo: 1 Min: 0 Max: 11,676
Results:
196,897 valid manifest files
28,175 non-valid manifest files
Results:
282,401 configurations
μ: 1.2 σ: 3.2 M: 1
Mo: 1 Min: 1 Max: 615
Results:
267,561 satisfiable (valid) configs.
14,840 no satisfiable (non-valid) configs.

Figure 10: Process extraction of real K8s configurations from YAML files in GitHub repositories.

CONTAINERS_PORTS_containerPort are directly assigned in the template (line 18). Furthermore, we define an internal mapping between those String features that were discretized in the variability model and the variation points in the template to assign the correct (e.g., string) value. For example, the GROUP_authentication feature has mapped the "authentication.k8s.io" value to be used in the template within the {{GROUP_authentication}} variation point (line 1). This approach ensures flexibility and consistency in generating tailored configurations that align with the feature model's constraints.

Listing 4: Jinja template to generate K8s configurations.

```
1 apiVersion: {{GROUP_authentication}}{{GROUP_events}}...{% if
       GROUP_core %}{% else %}/{% endif %}{{VERSION_v1}}{{
       VERSION_v1beta1}}{{VERSION_v1alpha1}}...
2 kind: {{KIND_Container}}{{KIND_Deployment}}{{KIND_Pod}}...
3 {% if metadata %}
4 metadata:
5   name: {{METADATA_name}}
6   ...
7 {% endif %}
8 {% if spec %}
9 spec:
10   {% if PodSpec %}
11   containers:
12   {% for c in PODSPEC_containers %}
13     - name: {{c.CONTAINERS_name}}
14       image: {{c.CONTAINERS_image}}
15       {% if c.CONTAINERS_ports %}
16       ports:
17         {% for p in c.CONTAINERS_ports %}
18         - containerPort: {{p.CONTAINERS_PORTS_containerPort}}
19         ...
20         {% endfor %}{% endif %}
21       ...
22   {% endfor %}{% endif %}
23   {% endif %}
24 ...
```

5.4 Results and Discussion

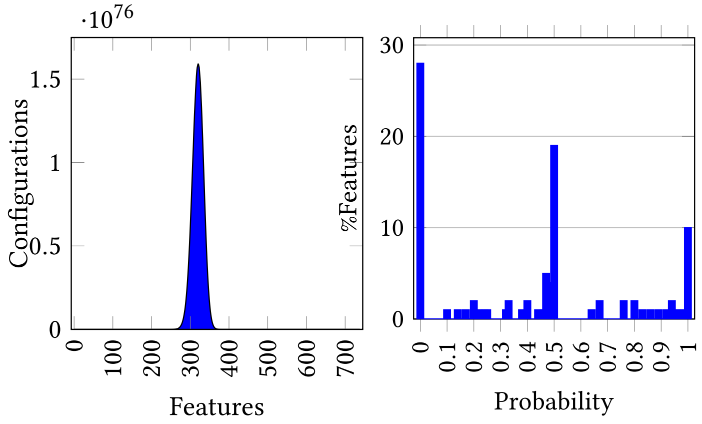
Feature model characterization. The synthesized feature model is characterized in terms of its structural and analytical metrics. Figure 13 in Appendix A shows the characterization of the K8s feature model (generated with the *FM Fact Label* tool [19, 21]). This characterization illustrates the model's complexity providing insights into its size and the variety of captured features and constraints. The model contains 738 features and 93 constraints, allowing for 5.73e77 configurations. It incorporates advanced variability modeling concepts: 319 typed (non-boolean) features, with 57 numerical and 262 string features. Additionally, 61 multi-features allow multiple instances of certain features. Among the constraints, 37% (34 constraints) are arithmetic, involving numerical features, while 63% (59 constraints) are logical. In particular, 25 of these logical constraints are complex, potentially representing multiple constraints, and 172 features participate in constraint definitions. Regarding rule definitions, 6 constraints are defined for compatibility of K8s resources, 65 are domain constraints, and 21 are artificially created for organizing the feature model (this information is not shown in Figure 13). Furthermore, 73% of the features (538) include at least one attribute, with documentation for 517, 86 with default values, 4 unavailable, and 19 incomplete.

RQ1. To what extent does the synthesized variability model cover real-world Kubernetes configurations and detect configuration errors? This research question aims to assess the comprehensiveness of the variability model by comparing it with a large set of real K8s configuration files. We will measure the coverage of the model by identifying how many features present in real configurations are accurately captured by the model, and whether the model includes all the variation points used in practice.

The process for extracting and validating real K8s configurations (Figure 10) results in 282,401 configurations, of which 95% (267,561 configurations) were deemed valid based on satisfiability [10] checks against the synthesized feature model, while 5% (14,840 configurations) were non-satisfiable. Such non-satisfiable configurations can occur for several reasons, from misconfigurations within the K8s API (e.g., invalid values or constraint violations) to errors in the synthesized feature model (e.g., incomplete and unavailable features, or human errors). Regardless of the reason, our feature model is able to cover 95% of the K8s configurations extracted from real projects available in GitHub repositories.

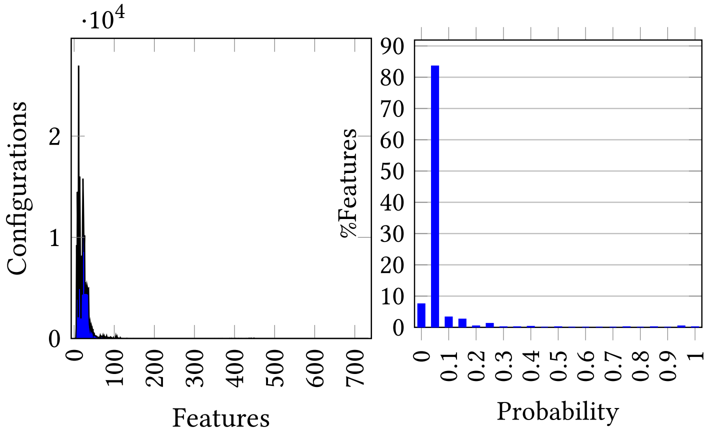
We complement our analysis of the extracted configurations with the *product distribution* [17] (Figure 11a for the feature model and Figure 12a for real configurations) that determines the number of valid configurations having a given number of features (omitting abstract features). We can observe that most of the extracted configurations contain a similar number of features, and this number (< 100 features) is low in comparison to the total number possible features in the model (738). This would suggest that configurations are homogeneous (i.e., very similar). However, the *feature inclusion probabilities* [17] (Figure 11b for the feature model and Figure 12a for real configurations) that determines the probability for a feature of being included in a valid configuration (i.e., for each feature, the proportion of valid configurations that include it), reveals that the extracted configurations are very heterogeneous (i.e., very different) and have low variance. We can observe that most features (74%) are nearly never included in a configuration (0.05% probability of being included in a valid configuration).

Table 1 reveals the differences in the product distribution for the feature model and the K8s configurations. This way, the most frequently occurring number of features for a configuration is 320 in the feature model vs 11 in the real configurations. This difference,



(a) Product distribution. **(b) Feature inclusion probabilities.**

Figure 11: K8s feature model analysis.



(a) Product distribution. **(b) Feature inclusion probabilities.**

Figure 12: K8s real configurations analysis.

Table 1: Difference between the product distribution of the feature model and the real K8s configurations.

	Mean	Std. dev.	Median	MAD	Mode	Min	Max	Range
Feature model	319.50	14.36	320	11.44	320	9	484	475
K8s configurations	22.00	18.99	21	9.82	11	5	449	444

observed also in Figures 12 and 11, aligns with the *K8s configuration tips* that establish to not specify default values unnecessarily so that simple, minimal configuration will make errors less likely. Moreover, this would indicate that most of the K8s configuration options are never considered, and users tend to leave default values in configurations. However, in the configurations generated with our feature model, we make explicit those features with their default values so that the users are aware of the full configuration of the K8s system. We also observe that the smallest and largest products in the feature model contain 9 and 484 features, respectively, compared to 5 and 449 features in the extracted configurations. These differences likely reflect the presence of abstract features in the model that do not appear in real configurations. We conclude that our synthesized feature model effectively captures the variability present in real K8s configurations, achieving a 95% coverage rate and identifying misconfigurations that would otherwise remain hidden.

5.5 Threats to Validity

Our study faces several threats to validity. First, the synthesized feature model may quickly become outdated due to ongoing updates to the K8s API documentation. In future work, we will consider using the *Kubernetes JSON Schemas*, which provide a more structured complement to the K8s API and can be processed by automated tools. Another threat to external validity is the generalization of our synthesis process, as it has only been tested with Kubernetes. To address this, we plan to apply and evaluate the synthesis approach with other systems, such as *Docker* and *Docker Compose* [39].

The ecological validity focuses on possible errors in the synthesis process and analysis, such as (1) the limited support of existing analysis tools for multi-features and arithmetical constraints that may impact the accuracy of configuration counts; and (2) human errors in synthesizing the model, as evidenced by the presence of 207 dead features in the model, for which we currently lack an explanation. To mitigate this threat we have relied on well-known tools for automated analysis of feature models such as *Flamapy* [13], *FM Fact Label* [19, 21], *BDDSampler* [16], and *UVLS* [32].

6 Related work

The synthesis of feature models, or reverse engineering of feature models, has gained attention for systematically managing variability in complex systems [31]. Techniques such as *feature graphs* [42], *evolutionary algorithms* [31], and *Monte-Carlo methods* [20] have been developed to extract feature models from various sources, including product variants [4, 29], propositional constraints [42], UML class diagrams [3], and source code artifacts [2]. However, models synthesized with these techniques often present interpretability challenges for domain engineers, resembling the *black box problem* in explainable AI [47]. Manual synthesis from documentation is less common but has been applied in domains such as Android systems [14], video sequences [1, 12], and data visualization [18, 40].

Reverse-engineering techniques have successfully extracted feature models from large, variability-intensive systems like the Linux Kernel [9, 11, 33, 41]. However, to date, no variability model for K8s has been developed [36, 43]. Additionally, related work in variability modeling for software deployment platforms is limited [25, 43]. Stötzner et al. [43] propose an integrated variability modeling approach for component implementations and deployment configurations across diverse technologies to manage variant complexity. Kumara et al. [25] introduce *Feature-Oriented Cloud* (FOCloud), which employs feature modeling to structure and constrain the deployment space of cloud services. Our K8s feature model presents a new problem for software product line evaluation [30], as it has become the Linux Kernel [41], but incorporates advanced extensions of variability modeling [45].

7 Conclusions and future work

This work introduces a real large intensive-variability case study and contributes a comprehensive feature model for Kubernetes, synthesizing its configuration space from its official API documentation. With 95% coverage of real configurations, the model effectively represents practical deployments. The explicit inclusion of default features provides users with a complete view of potential configuration options, enhancing comprehensiveness and error prevention. With this contribution, we aim to provide the software product line community with a new case study of an intensive-variability system, offering an alternative to the extensively studied Linux Kernel. This addition should broaden the scope of research and applications within the field.

Future directions include refining the model by automatically incorporating more comprehensive K8s JSON Schema data and extending the approach to other containerized platforms, such as Docker and Docker Compose.

Acknowledgments

We thank Enrique López Encinas who has also contributed to this work. Work supported by the projects *IRIS* PID2021-122812OB-I00 (co-financed by FEDER funds), and *TASOVA PLUS* research network (RED2022-134337-T); and by Universidad de Málaga.

References

[1] Mauricio Alférez, Mathieu Acher, José Angel Galindo, Benoit Baudry, and David Benavides. 2019. Modeling variability in the video domain: language and experience report. *Softw. Qual. J.* 27, 1 (2019), 307–347. https://doi.org/10.1007/S11219-017-9400-8

[2] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants. *Empir. Softw. Eng.* 22, 4 (2017), 1763–1794. https://doi.org/10.1007/S10664-016-9462-4

[3] Wesley K. G. Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon. 2020. Automatic extraction of product line architecture and feature models from UML class diagram variants. *Inf. Softw. Technol.* 117 (2020). https://doi.org/10.1016/J.INFSOF.2019.106198

[4] Wesley K. G. Assunção, Silvia R. Vergilio, Roberto E. Lopez-Herrejon, and Lukas Linsbauer. 2023. Search-Based Variability Model Synthesis from Variant Configurations. In *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer International Publishing, 115–141. https://doi.org/10.1007/978-3-031-11686-5_5

[5] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running.* " O'Reilly Media, Inc.".

[6] Emiliano Casalicchio. 2019. *Container Orchestration: A Survey.* Springer International Publishing, Cham, 221–235. https://doi.org/10.1007/978-3-319-92378-9_14

[7] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Softw. Process. Improv. Pract.* 10, 1 (2005), 7–29. https://doi.org/10.1002/spip.213

[8] Boubacar Diarra, Karine Guillouard, Meryem Ouzzif, Philippe Merle, and Jean-Bernard Stefani. 2024. In-Depth Analysis of Kubernetes Manifest Verification Tools for Robust CNF Deployment. In *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*. 17–24. https://doi.org/10.1109/ICIN60470.2024.10494445

[9] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. Analysing the Linux kernel feature model changes using FMDiff. *Softw. Syst. Model.* 16, 1 (2017), 55–76. https://doi.org/10.1007/S10270-015-0472-2

[10] Alexander Felfernig, Andreas A. Falkner, and David Benavides. 2024. *Feature Models - AI-Driven Design, Analysis and Applications.* Springer. https://doi.org/10.1007/978-3-031-61874-1

[11] David Fernández-Amorós, Ruben Heradio, José Miguel Horcas Aguilera, José A. Galindo, David Benavides, and Lidia Fuentes. 2024. Pragmatic Random Sampling of the Linux Kernel: Enhancing the Randomness and Correctness of the conf Tool. In *28th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. A. ACM, Dommeldange, Luxembourg, 24–35. https://doi.org/10.1145/3646548.3672586

[12] José Angel Galindo, Mauricio Alférez, Mathieu Acher, Benoit Baudry, and David Benavides. 2014. A variability-based testing approach for synthesizing video sequences. In *International Symposium on Software Testing and Analysis (ISSTA)*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, San Jose, CA, USA, 293–303. https://doi.org/10.1145/2610384.2610411

[13] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. 2023. 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 (SPLC)*, Vol. B. ACM, Tokyo, Japan, 16–19. https://doi.org/10.1145/3579028.3609008

[14] José Angel Galindo, Hamilton A. Turner, David Benavides, and Jules White. 2016. Testing variability-intensive systems using automated analysis: an application to Android. *Softw. Qual. J.* 24, 2 (2016), 365–405. https://doi.org/10.1007/S11219-014-9258-Y

[15] Panagiotis Gkonis, Anastasios Giannopoulos, Panagiotis Trakadas, Xavi Masip-Bruin, and Francesco D'Andria. 2023. A Survey on IoT-Edge-Cloud Continuum Systems: Status, Challenges, Use Cases, and Open Issues. *Future Internet* 15, 12 (2023). https://doi.org/10.3390/fi15120383

[16] Ruben Heradio, David Fernández-Amorós, José A. Galindo, David Benavides, and Don S. Batory. 2022. Uniform and scalable sampling of highly configurable systems. *Empir. Softw. Eng.* 27, 2 (2022), 44. https://doi.org/10.1007/S10664-021-10102-5

[17] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the statistical analysis of variability models. In *41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada, 843–853. https://doi.org/10.1109/ICSE.2019.00091

[18] José Miguel Horcas, José A. Galindo, and David Benavides. 2022. Variability in data visualization: a software product line approach. In *26th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. A. ACM, Graz, Austria, 55–66. https://doi.org/10.1145/3546932.3546993

[19] José Miguel Horcas, José A. Galindo, Lidia Fuentes, and David Benavides. 2025. FM fact label. *Sci. Comput. Program.* 240 (2025), 103214. https://doi.org/10.1016/J.SCICO.2024.103214

[20] José Miguel Horcas, José A. Galindo, Ruben Heradio, David Fernández-Amorós, and David Benavides. 2023. A Monte Carlo tree search conceptual framework for feature model analyses. *J. Syst. Softw.* 195 (2023), 111551. https://doi.org/10.1016/J.JSS.2022.111551

[21] José Miguel Horcas, José Angel Galindo, Mónica Pinto, Lidia Fuentes, and David Benavides. 2022. *FM fact label*: a configurable and interactive visualization of feature model characterizations. In *26th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. B. ACM, Graz, Austria, 42–45. https://doi.org/10.1145/3503229.3547025

[22] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023. Empirical analysis of the tool support for software product lines. *Softw. Syst. Model.* 22, 1 (2023), 377–414. https://doi.org/10.1007/S10270-022-01011-2

[23] Danylo Khalyeyev, Tomás Bures, and Petr Hnetynka. 2022. Towards Characterization of Edge-Cloud Continuum. In *Software Architecture. ECSA 2022 Tracks and Workshops (Lecture Notes in Computer Science, Vol. 13928)*. Springer, Prague, Czech Republic, 215–230. https://doi.org/10.1007/978-3-031-36889-9_16

[24] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, Paderborn, Germany, 291–302. https://doi.org/10.1145/3106237.3106252

[25] Indika Kumara, Mohamed Hameez Ariz, Mohan Baruwal Chhetri, Majid Mohammadi, Willem-Jan Van Den Heuvel, and Damian A. Tamburri. 2023. FOCloud: Feature Model Guided Performance Prediction and Explanation for Deployment Configurable Cloud Applications. *IEEE Transactions on Services Computing* 16, 1 (2023), 302–314. https://doi.org/10.1109/TSC.2022.3142853

[26] Van-Cuong Le and Myungsik Yoo. 2021. Application for Managing YAML Template for Kubernetes. *The Journal of Korean Institute of Communications and Information Sciences* 46, 11 (2021), 1950–1957. https://doi.org/10.7840/kics.2021.46.11.1950

[27] Ka Chun Anthony Lee, Maria-Teresa Segarra, and Stephane Guelec. 2014. A Deployment-oriented Development Process based on Context Variability Modeling. In *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, Lisbon, Portugal, 454–459. https://doi.org/10.5220/0004806304540459

[28] Michael Lettner, Jorge Rodas-Silva, José Angel Galindo, and David Benavides. 2019. Automated analysis of two-layered feature models with feature attributes. *J. Comput. Lang.* 51 (2019), 154–172. https://doi.org/10.1016/J.COLA.2019.01.005

[29] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Softw. Syst. Model.* 16, 4 (2017), 1179–1199. https://doi.org/10.1007/S10270-015-0512-Y

[30] Roberto E. Lopez-Herrejon and Don S. Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *3rd International Conference on Generative and Component-Based Software Engineering (GCSE) (Lecture Notes in Computer Science, Vol. 2186)*, Jan Bosch (Ed.). Springer, Erfurt, Germany, 10–24. https://doi.org/10.1007/3-540-44800-4_2

[31] Roberto Erick Lopez-Herrejon, Lukas Linsbauer, José Angel Galindo, José Antonio Parejo, David Benavides, Sergio Segura, and Alexander Egyed. 2015. An assessment of search-based techniques for reverse engineering feature models. *J. Syst. Softw.* 103 (2015), 353–369. https://doi.org/10.1016/J.JSS.2014.10.037

[32] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. 2023. UVLS: A Language Server Protocol For UVL. In *27th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. B. ACM, Tokyo, Japan, 43–46. https://doi.org/10.1145/3579028.3609014

[33] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In *14th International Conference on Software Product Lines: Going Beyond (SPLC) (Lecture Notes in Computer Science, Vol. 6287)*. Springer, Jeju Island, South Korea, 136–150. https://doi.org/10.1007/978-3-642-15579-6_10

[34] Daniel-Jesus Munoz, Mónica Pinto, and Lidia Fuentes. 2022. Quality-aware analysis and optimisation of virtual network functions. In *26th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. A. ACM, Graz, Austria, 210–221. https://doi.org/10.1145/3546932.3547007

[35] Daniel-Jesus Munoz, Mónica Pinto, Lidia Fuentes, and Don S. Batory. 2023. Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language. *J. Syst. Softw.* 204 (2023), 111770. https://doi.org/10.1016/J.JSS.2023.111770

[36] Giuseppe Muntoni, Jacopo Soldani, and Antonio Brogi. 2021. Mining the Architecture of Microservice-Based Applications from their Kubernetes Deployment. In *Advances in Service-Oriented and Cloud Computing*. Springer International Publishing, Cham, 103–115.

[37] Nigel Poulton. 2023. *The kubernetes book*. NIGEL POULTON LTD.

[38] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 99 (May 2023), 36 pages. https://doi.org/10.1145/3579639

[39] David Reis, Bruno Piedade, Filipe F. Correia, João Pedro Dias, and Ademar Aguiar. 2022. Developing Docker and Docker-Compose Specifications: A Developers' Survey. *IEEE Access* 10 (2022), 2318–2329. https://doi.org/10.1109/ACCESS.2021.3137671

[40] David Romero-Organvidez, José Miguel Horcas, José A. Galindo, and David Benavides. 2024. Data visualization guidance using a software product line approach. *J. Syst. Softw.* 213 (2024), 112029. https://doi.org/10.1016/J.JSS.2024.112029

[41] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. In *4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS) (ICB-Research Report, Vol. 37)*. Universität Duisburg-Essen, Linz, Austria, 45–51. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[42] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wasowski, and Krzysztof Czarnecki. 2014. Efficient synthesis of feature models. *Inf. Softw. Technol.* 56, 9 (2014), 1122–1143. https://doi.org/10.1016/J.INFSOF.2014.01.012

[43] Miles Stötzner, Uwe Breitenbücher, Robin D. Pesl, and Steffen Becker. 2023. Managing the Variability of Component Implementations and Their Deployment Configurations Across Heterogeneous Deployment Technologies. In *29th International Conference on Cooperative Information Systems (CoopIS) (Lecture Notes in Computer Science, Vol. 14353)*. Springer, Groningen, The Netherlands, 61–78. https://doi.org/10.1007/978-3-031-46846-9_4

[44] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet another textual variability language?: a community effort towards a unified language. In *25th ACM International Systems and Software Product Line Conference (SPLC)*, Mohammad Reza Mousavi and Pierre-Yves Schobbens (Eds.), Vol. A. ACM, Leicester, United Kingdom, 136–147. https://doi.org/10.1145/3461001.3471145

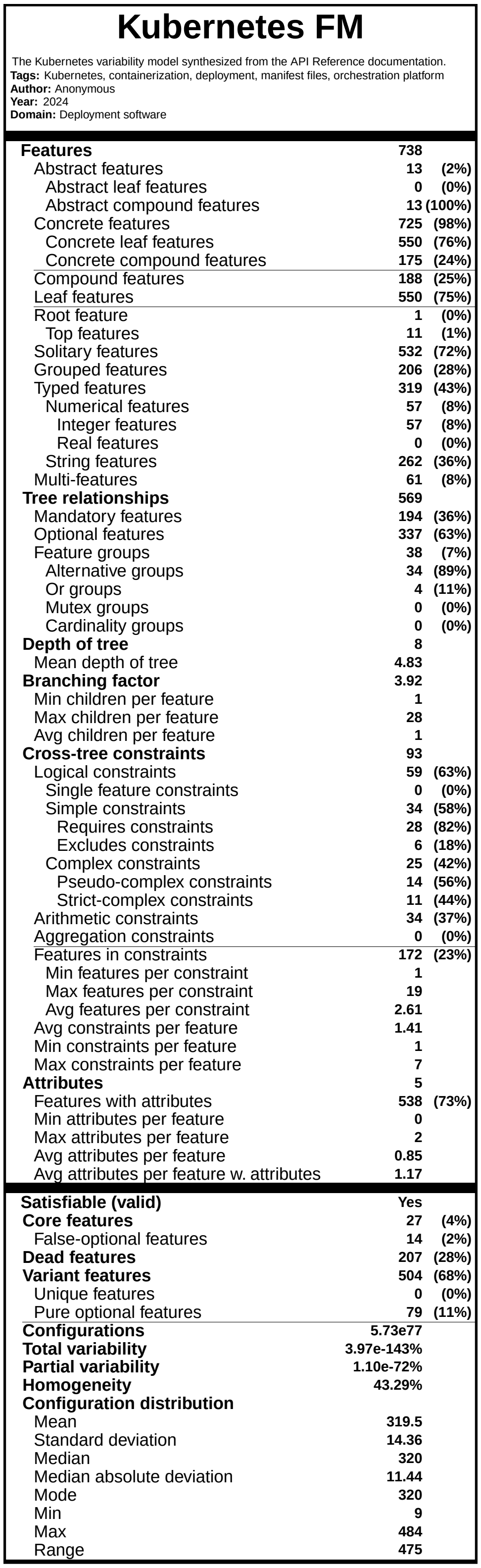
[45] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A. Galindo, and David Benavides. 2023. UVLParser: Extending UVL with Language Levels and Conversion Strategies. In *27th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. B. ACM, 39–42. https://doi.org/10.1145/3579028.3609013

[46] The Kubernetes Authors. 2024. Kubernetes API documentation. https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.30/#api-overview v1.30.0.

[47] Carlos Zednik and Hannes Boelsen. 2022. Scientific Exploration and Explainable Artificial Intelligence. *Minds Mach.* 32, 1 (2022), 219–239. https://doi.org/10.1007/S11023-021-09583-6

A K8s feature model characterization

Figure 13 shows the characterization of the K8s feature model generated with the FM Fact Label tool [19, 21].



Kubernetes FM

The Kubernetes variability model synthesized from the API Reference documentation.
Tags: Kubernetes, containerization, deployment, manifest files, orchestration platform
Author: Anonymous
Year: 2024
Domain: Deployment software

Features	738	
Abstract features	13	(2%)
Abstract leaf features	0	(0%)
Abstract compound features	13	(100%)
Concrete features	725	(98%)
Concrete leaf features	550	(76%)
Concrete compound features	175	(24%)
Compound features	188	(25%)
Leaf features	550	(75%)
Root feature	1	(0%)
Top features	11	(1%)
Solitary features	532	(72%)
Grouped features	206	(28%)
Typed features	319	(43%)
Numerical features	57	(8%)
Integer features	57	(8%)
Real features	0	(0%)
String features	262	(36%)
Multi-features	61	(8%)
Tree relationships	569	
Mandatory features	194	(36%)
Optional features	337	(63%)
Feature groups	38	(7%)
Alternative groups	34	(89%)
Or groups	4	(11%)
Mutex groups	0	(0%)
Cardinality groups	0	(0%)
Depth of tree	8	
Mean depth of tree	4.83	
Branching factor	3.92	
Min children per feature	1	
Max children per feature	28	
Avg children per feature	1	
Cross-tree constraints	93	
Logical constraints	59	(63%)
Single feature constraints	0	(0%)
Simple constraints	34	(58%)
Requires constraints	28	(82%)
Excludes constraints	6	(18%)
Complex constraints	25	(42%)
Pseudo-complex constraints	14	(56%)
Strict-complex constraints	11	(44%)
Arithmetic constraints	34	(37%)
Aggregation constraints	0	(0%)
Features in constraints	172	(23%)
Min features per constraint	1	
Max features per constraint	19	
Avg features per constraint	2.61	
Avg constraints per feature	1.41	
Min constraints per feature	1	
Max constraints per feature	7	
Attributes	5	
Features with attributes	538	(73%)
Min attributes per feature	0	
Max attributes per feature	2	
Avg attributes per feature	0.85	
Avg attributes per feature w. attributes	1.17	
Satisfiable (valid)	Yes	
Core features	27	(4%)
False-optional features	14	(2%)
Dead features	207	(28%)
Variant features	504	(68%)
Unique features	0	(0%)
Pure optional features	79	(11%)
Configurations	5.73e77	
Total variability	3.97e-143%	
Partial variability	1.10e-72%	
Homogeneity	43.29%	
Configuration distribution		
Mean	319.5	
Standard deviation	14.36	
Median	320	
Median absolute deviation	11.44	
Mode	320	
Min	9	
Max	484	
Range	475	

Figure 13: Characterization of the K8s feature model.

Applying Graph Neural Networks to Learn Graph Configuration Spaces

Michael Mittermaier
School of Computer Science and Statistics
Trinity College Dublin
Dublin, Ireland
mittermm@tcd.ie

Takfarinas Saber
School of Computer Science
University of Galway
Galway, Ireland
takfarinas.saber@universityofgalway.ie

Goetz Botterweck
School of Computer Science and Statistics
Trinity College Dublin, Lero
Dublin, Ireland
goetz.botterweck@tcd.ie

Abstract

The configuration process for feature-oriented product lines is well-researched for Boolean and numerical feature configurations. However, in several engineering fields, we encounter the challenge of finding the optimal *graph* structure to describe a product configuration. Optimising complex graph structures towards multiple objectives within numerous constraints requires a deep understanding of the graph configuration space and the product properties it represents. This study aims to leverage graph neural networks (GNNs) to predict product properties, thereby supporting the configuration process in product lines. In a controlled experiment, we compare a GNN-based approach to a recent state-of-the-art approach utilising graph embeddings. We evaluate these methods on both accuracy and learning efficiency. Our findings indicate that the GNN-based approach outperforms the embedding-based method in terms of accuracy. However, it requires a substantially larger volume of training data to achieve these results. Overall, this research demonstrates the applicability of an ML-supported framework for engineering product lines using graph configurations.

Keywords

Graph Neural Network, Graph Configurations, Learning Configuration Spaces.

ACM Reference Format:
Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck. 2025. Applying Graph Neural Networks to Learn Graph Configuration Spaces. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3715340.3715444

1 Introduction

Feature models are a common approach used in Software Product Line Engineering (SPLE) to represent configuration choices and variability among products within a family of software-intensive systems. During the configuration process, we make, for instance, *Boolean or numerical* configuration decisions within the options and constraints defined in the feature model to obtain a product configuration describing a particular product.

In several engineering fields, however, it is more amenable to the particular domain to use configurations (in the wider sense) in *structural or topological* form. Examples for such problems are arranging the components of a software architecture to improve non-functional properties of the software system, sequencing the processing steps in a manufacturing plant such that costs and production time are minimised, or arranging rooms in a building to minimise energy consumption.

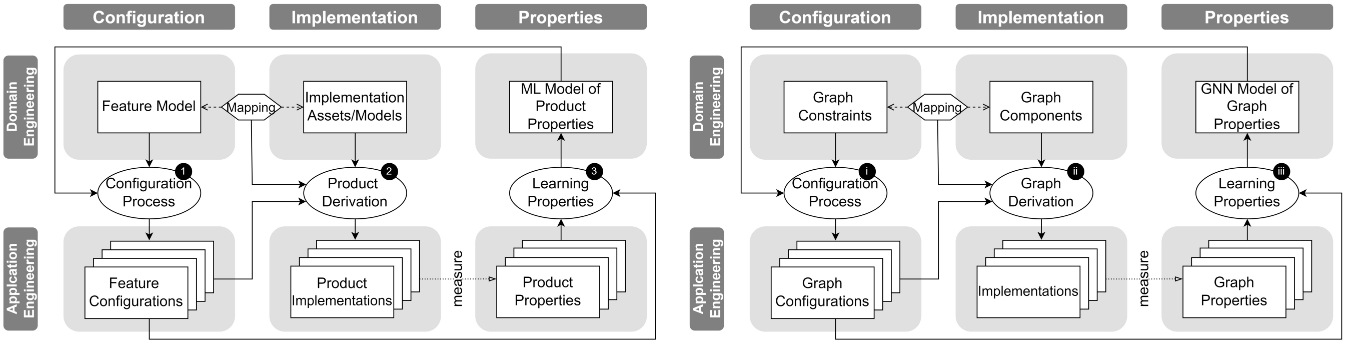
In these cases, we have to find a graph representing the system with the best resulting properties. In this paper, we argue that this problem of finding the best graph (based on given constraints) can be seen as a configuration and optimisation process with similarities to finding an optimal feature configuration (based on a feature model). This process requires the selection of a configuration out of all candidate graphs while simultaneously considering various constraints. Since this potentially requires evaluating 10,000s of graphs (including, e.g., computationally expensive simulation of the represented systems), one alternative is to rely on regression analysis to predict measurements to narrow down the configuration space. We refer to this class of problems as learning graph configuration spaces (LEGCS), inspired by Pereira et al. who define a framework for corresponding non-graph case of learning a configuration space [29]. The central challenge of LEGCS is applying regression techniques on graph structures, while most techniques operate on Euclidean structures such as vectors or matrices.

Mittermaier et al. [27] have shown that regression analysis based on graph embeddings into a vector space can support learning graph configuration spaces. The initial benefit here is that mapping the graph space into an Euclidean vector space facilitates just applying a large number of well-known machine learning (ML) techniques operating on fixed-sized vectors. However, these ML algorithms can only operate on the information that are not “lost in translation”, when the graph embedding translates information contained in the graphs into vectors, potentially leading to a loss of accuracy.

We hypothesise that it would be beneficial to apply ML approaches that work on the graph-oriented structures directly – in particular, in terms of the accuracy of the obtained results. Graph neural networks (GNN), a family of neural network algorithms, interpret the graphs themselves as a neural network on which they operate. In contrast to ML approaches that work on Euclidean data, GNNs do not require the extra step of graph embedding. Hence, this *might* avoid the “lost in translation” problem. This leads us to our research question:

VaMoS 2025, Rennes, France

ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715444



(a) Learning-Supported Product Configuration in Software Product Line Engineering.

(b) Adaptation of Software Product Line Configuration to Learning-Supported Graph Configuration.

Figure 1: Product Line Engineering in Euclidean and Graph Spaces.

> *When learning graph-based configuration spaces, how does learning based on GNNs compare to regression techniques based on graph embeddings – with respect to the accuracy of the predictions and the size of the required training set?*

Contributions. Our paper makes the following contributions:

(1) We describe a conceptual framework that resembles well-known feature-oriented frameworks for Software Product Line Engineering and translates them to graph-oriented configurations.
(2) We propose a new regression approach in context of learning graph configuration spaces based on GNNs that uses graph representations throughout process, thus retaining the graph structural information at all stages, to potentially achieve more accurate predictions of product properties.
(3) We report on a controlled experiment that explores the efficiency of regression analysis with (a) the embedding-based approach and (b) three different GNN techniques (GCN, GAT, GraphSAGE) with respect to accuracy and training data size.
(4) We use the same dataset[1] as the state-of-the-art [27] to encourage follow-up work and ensure replicability.

The remainder of the paper is structured as follows: We discuss conceptual frameworks of feature-oriented and graph-oriented product line engineering (Section 2). Then, we go into more detail about the existing learning approach using graph embeddings (Section 3), and the alternative approach using GNNs (Section 4). Afterwards, we report on an experiment applying these two learning strategies (Section 5) and its results (Section 6), and conclude the paper (Section 7).

2 Conceptual Framework

In this section, to relate our work to the existing body of knowledge in Software Product Line Engineering (SPLE) we first outline a conceptual framework for SPLE using common terminology. We then describe the challenge of learning a model representing the whole configuration space (of configurations and resulting product properties).

Corresponding to this, we then introduce our graph-oriented framework for product line engineering, which has a similar logical structure but uses graphs to represent configurations. Then, analogously, we describe the challenge of learning a model of the graph-based configuration space.

We close this section with a look into related work on search-based graph optimisation.

2.1 Feature-oriented Product Line Engineering

We are mainly interested in the general principle of representing configuration choices in SPLE and then configuring and deriving products [6, 10]. For the sake of the discussion here, we focus on feature-oriented approaches, but we acknowledge that there are other related approaches, e.g., decision modelling [12] or other forms of variability modelling.

For general orientation, we start from a generic feature-oriented SPLE framework (see Figure 1a), which covers two levels (domain engineering and application engineering) and three aspects (configuration, implementation, and properties).

Configuration. We assume that the available configuration options and constraints among them is captured in a feature model. We use this model in the ❶ configuration process to derive a feature configuration (or similar artifact) representing one product [20].

Implementation. If the feature model is mapped to implementation assets, we can use this mapping to ❷ derive the implementation of a particular product from a feature configuration. There are many different ways on how we can represent the effect of configuration decisions in the implementation, e.g., in annotative or compositional approaches [5, 15]. Ideally, this product derivation is automated or at least tool-supported.

Product Properties. Every implementation of a product has measurable properties. Once we have a meaningful number of implemented products and associated properties, we can aim to ❸ learn a model that captures knowledge about product properties for all products. This model can support the configuration process by estimating product properties prior to often costly implementations and measurements [32, 33].

[1] https://github.com/mittermm/LEGCS

In the following Section 2.2, we discuss the adaptations necessary to translate such an SPLE framework to use graph configurations instead of Euclidean feature configurations (see Figure 1b). Afterwards, we go into depth on the ❸ learning process of product properties in feature-oriented configuration spaces and graph configuration spaces.

2.2 Graph-oriented Product Line Engineering

In our adaptation to graph-oriented product lines, vertically we keep the well-known two levels of domain engineering focusing on the whole product line, and application engineering on individual products derived from that product line. Horizontally, i.e. distinguishing different aspects to focus on, we keep the three aspects of configuration, implementation, and resulting product properties.

Configuration. Instead of a feature model representing the configuration choices, we now have a set of graph constraints directly mapped to the domain-specific graph components limiting the number of valid graph configurations in the graph space. We use these constraints to (i) obtain a graph configuration describing one product. Note that in many application cases that we hope to address, unlike a feature configuration the graph configuration is not *constructed* out of a sequence of discrete user decisions. Often, the process will be rather *search-oriented* – more like optimising a feature configuration that fulfils a number of given constraints.

Implementation. Similarly to a feature-oriented approach, we can (ii) derive a product implementation from the graph configuration, potentially using pre-existing graph components or translating graph nodes and edges into implementation components, and measure the product properties.

Product Properties. Corresponding to similar learning approaches for traditional feature-oriented (i.e., non-graph-oriented) configurations, we can use these product properties to (iii) train a graph-based learning model to predict product properties for all valid graph configurations in order to support the configuration process.

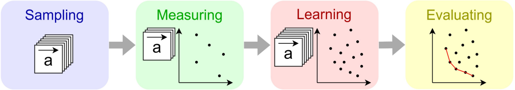


Figure 2: Four phases of learning configuration spaces according to Pereira et al. [29].

2.3 Learning (Feature-oriented) Configuration Spaces

Configuring large systems requires understanding the space of possible configurations, such that we can optimise the system towards several goals while simultaneously fulfilling multiple constraints. For instance, configuring the Linux kernel with more than 15,000 configuration options can lead to 2^{15000} possible variants of the kernel [3], more candidates than there are atoms in the known universe. Although this example is extreme, it shows how the configuration space can become incomprehensibly large. Hence, for systems of realistic size and complexity, there is a need to systematically *learn* the configuration space, e.g., to make predictions on properties of a previously-unseen configuration. Below, we give an introduction to common practices for ❸ learning product properties from product configurations to aid the configuration process.

Pereira et al. provide a survey of current practices for learning software configuration spaces [29]. These practices follow a four-phase model consisting of sampling, measuring, learning, and evaluating (see the illustration in Figure 2).

In the first phase of such an approach, we gather a sample of valid software configurations, each represented using a vector holding at least the value of each parameter in the software they depict. Then, we measure the performance of the configurations within this sample using its vector. Subsequently, we also train an ML model using the same vector of sampled configurations and their measurements, and finally we predict system properties (a.k.a. performances) for more unknown software configurations (expressed as vectors of the same dimension) and evaluate the quality of the ML model (in particular, with respect to low prediction errors, a small model size, and a reasonable measurement effort). There are various strategies for each of the phases in this model.

Application domains for learning configuration spaces include, e.g., pure predictions of the configuration properties, interpretability, optimisation, mining constraints, and evolution [29].

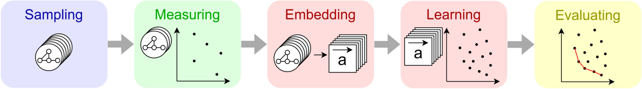
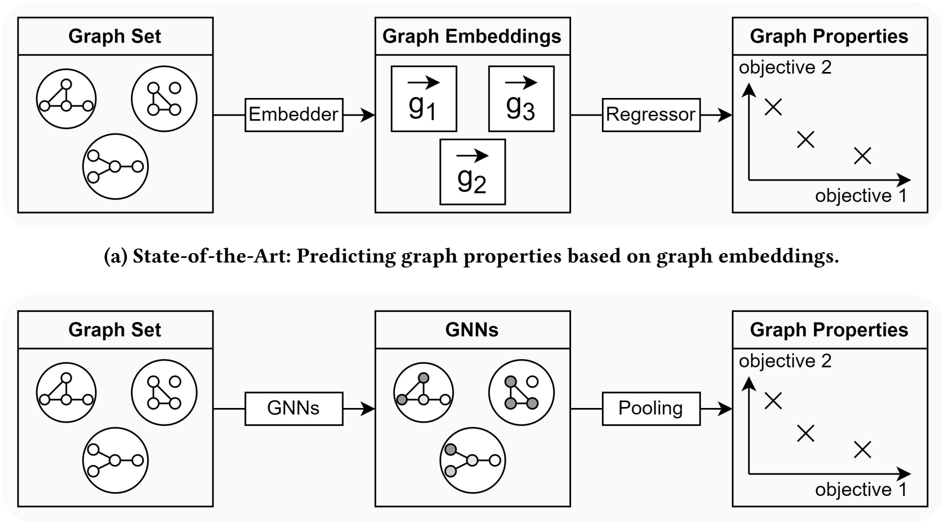


Figure 3: Adaptation to learn graph configuration spaces (LEGCS) by Mittermaier et al. [27].

2.4 Learning Graph Configuration Spaces

Similarly to Euclidean configuration spaces, we also need to (iii) train a learning model for graph configuration spaces that captures product properties of the whole configuration space. While the described four-phase framework above provides a conceptual model to structure the problem, the considered techniques are focused on (and limited to) Euclidean spaces. There are, however, applications where it would be beneficial to consider and configure a system in terms of its *structure* or topology. For such application domains, previous work by Mittermaier et al. [27] has adapted techniques and concrete strategies of learning software configuration spaces to learning graph configuration spaces (LEGCS). This requires the addition of embedding to the same four phases of sampling, measuring, learning, and evaluating (see Figure 3).

The authors proposed a pipeline that starts with generating a large number of valid graph configurations. Within this larger set, one can randomly select a sample of graphs, and measure their performances using a simulator. The learning phase requires a model capable of performing regression analysis on a whole-graph level. In regression analysis (a type of function approximation), the overall goal is to accurately predict a continuous numeric output for several independent input variables [11]. In our particular case, we aim to predict properties such as the performance of the graph configuration towards a certain goal. In Section 3, we explain a first strategy using a two-step model of embedding the graph set and applying ML on the embeddings, and in Section 4, we propose an alternative learning model based on graph neural networks.



(a) State-of-the-Art: Predicting graph properties based on graph embeddings.

(b) Proposed Approach: Predicting graph properties based on GNNs. Trained feature vectors of the nodes represented in greyscale.

Figure 4: Learning strategies to predict graph properties.

2.5 Search-Based Graph Optimisation

In this work, we focus on learning methods in context of LEGCS for pure prediction within an optimisation problem by finding the k best solutions. This problem has also been addressed in many applications in engineering outside of LEGCS.

Search heuristics, such as A*, Best-First, and Depth First to identify the k best solutions have already been adapted to graphs in Bayesian networks [16]. However, the search space here is limited to finite domains of discrete variables. For NP-hard optimisation problems, we require metaheuristics to solve large-scale instances in reasonable computing times. Metaheuristics that require simulations of candidates to assess their fitness are called simheuristics [9], they are particularly used in application domains dealing with uncertainties, such as logistics, transportation, and other supply chain areas [23]. Simheuristics find the best candidate within a configuration space, the challenge here, however, lies in understanding the graph configuration space, and finding the k best candidates.

3 State-of-the-Art for Learning Phase: Emb+RF

The learning phase of LEGCS has been addressed before by two separate learning steps that are visualised in Figure 4a:

In the first step, whole-graph embedding is used to map every graph into a fixed-dimensional vector space. There are various methods to achieve an expressive (i.e., graph-information preserving) and efficient (low embedding time and dimensionality) embedding [8], in previous work on LEGCS [27], an embedding based on the degrees of neighbouring nodes [7] was used to capture the topology.

In the second step, an ML model such as random forest [25] is trained to produce predictions based on the vectors representations of those graphs. Early results showed that this process sped up the search for the k best graph representations by suggesting well-performing graph configurations to simulate.

In this paper, we refer to LEGCS systems that rely in their learning phase on this combination of embedding and a random forest regressor as $LEGCS_{Emb+RF}$. The first step allows us to utilise a large set of ML models that only operate on Euclidean structures (vectors and matrices) in the second step. However, while regression on whole graphs based on embeddings has been shown to be feasible in various applications, it encounters a fundamental limitation: the embedding cannot fully represent the graph set's inherent complexity within a reasonable embedding size. This means, that any ML system operating on these embeddings cannot exploit all graph information, only data stored in the embedding. In the following section, we propose an alternative learning strategy to improve the learning model's understanding of the graph space, and thus, increase the prediction accuracy.

4 Proposed Approach for Learning Phase: GNN

In this section, we show an alternative way of learning the configuration space with embeddings, i.e., regression with graph neural networks, and describe its application for LEGCS.

4.1 Graph Neural Networks

GNNs are used for various tasks in several fields of science and engineering. Those tasks include classifying nodes [35], subgraphs [4], or graphs [17], link predictions [39], or in case of this work predicting non-functional properties of graphs. Zhou et al. [41] present an overview over current practices and applications of GNNs. While we focus on predicting properties of graphs in an engineering context, other applications for predicting non-functional properties include: predicting chemical reaction products (with molecules as graphs) in organic chemistry [13], predict protein interfaces (with proteins as graphs) in biology [19], or predicting traffic states (with street networks as graphs) in applied mathematics [40]. Even predicting connections, e.g., in social networks [26] can also be interpreted as GNNs predicting non-functional properties.

The fundamental concept of GNNs is treating a graph itself as a neural network which allows direct processing and manipulation of graph-structured data. This way, GNNs can capture intricate relationships and dependencies among graph elements, and enable expressive and context-aware learning models.

Depending on the graphs we are operating on, and the task our GNN should perform, we need to make various decisions during the design process. You et al. [38] describe important design decisions for building GNNs in:

1. *Message passing*, where we determine what features of each node we pass on to its neighbours.
2. *Aggregation*, where we calculate the feature vector of a node in the next GNN layer out of its current feature vector and the neighbour's messages.
3. *Layer connectivity*, where we define how GNN layers are interconnected. Usually, we sequentially calculate the new feature vector of each node in every GNN layer. That could lead to over-smoothing where the feature vectors of all nodes become too similar. Solutions here include (1) a low number of layers and expressive aggregation, for instance, by aggregating with a deep neural network, or (2) add layers without message passing (pre- and post-processing layers).
4. *Graph manipulation*, where we improve learning results by modifying the input graph to a computational graph. Reasons for modifications include: lack of features in the input graph, sparse input graph (inefficient message passing), dense input graph (costly message passing), large input graph (high computational complexity). Solutions include feature augmentation, adding virtual nodes and edges, sample neighbours while passing messages, and sampling subgraphs to compute embeddings.
5. *Learning objective*, where we decide whether the objective is on the node, edge, or whole-graph level.

In terms of regression on a whole-graph level, the standard method to go from here, especially for small graphs, is global pooling [37]. Here, we pool the feature vectors of all nodes that were calculated during the learning process of the GNN together, and calculate the regression result by, for instance, picking the maximum/minimum/mean vector or summing these vectors together [36].

4.2 GNN for LEGCS

In Section 3, we analysed the current approach on LEGCS using a graph embedding. By mapping graphs into a fixed-dimensional vector space, we lose structural and relational nuances, which the ML system cannot exploit when predicting the graph properties. Consequently, learning models operating solely on these such embeddings might be limited in their predictive power.

Hence, we explore a variation of the first model (Emb+RF) that replaces the two learning steps of graph embedding and regression with a new kind of ML approach based on GNNs that we call $LEGCS_{GNN}$. This approach can use graph structures in form of a neural network to learn feature vectors for each node that keep information of the node itself and its neighbourhood. Subsequently, we can use this learned GNN to predict graph properties (by pooling properties from the individual nodes). Figure 4b presents a visual explanation of this variation.

We hypothesise that such a learning approach that preserves the graph's inherent information might have benefits in terms of predicting properties of the graph in terms of accuracy. On the other hand, training a GNN until it reaches the required accuracy might take more training samples (and hence more time) compared to the embedding-based approach.

To investigate whether this new variant is promising, we conduct an experiment where we evaluate $LEGCS_{Emb+RF}$ and three variants of $LEGCS_{GNN}$ based on three kinds of GNN. We report on the experiment in the following section.

5 Experiment

After describing the existing learning approach Emb+RF (Section 3), and our proposed alternative approach using GNNs (Section 4), we report in this section on our experiments to compare them in context of the learning phase in LEGCS.

5.1 Methodology

The research question we raised in the introduction asks for a "design, evaluation, or analysis of a particular instance" [31]. We conduct a controlled experiment [14] and compare $LEGCS_{Emb+RF}$ basing its learning phase on a combination of graph embedding and ML (in our case, random forest) against $LEGCS_{GNN}$ with respect to the accuracy of the predictions and the size of the training set. For $LEGCS_{GNN}$, we will evaluate different popular GNN algorithms: GCN, GAT, and GraphSAGE. We follow the reporting guidelines by Jedlitschka et al. [22] that provide structure and questions to cover, e.g., in terms of planning, analysing, and discussing the experiment.

5.2 LEGCS Problem: HVAC

Our example of a LEGCS problem is from civil engineering dealing with HVAC (Heating, Ventilation, and Air-Conditioning) systems [1]. Our goal is to accurately predict the AC running time per month (ideally with low sampling costs to train the ML model) for all of the candidates in the graph configuration space. This space of candidates consists of floor plans and their air ventilation systems. We interpret and represent the topological configurations of these floor plans as graphs configuring a building, making this an exploration problem on graph configurations. These predictions can later be used to explore and optimise the graph configuration space.

5.3 Dataset

In our paper, we use the same dataset that was designed and used by the state-of-the-art [27].

5.3.1 Graph Space: In the dataset, we explore a graph space consisting of 40,000 floor plans that we interpret as graphs; each of them is a different configuration of rooms within the same building. We interpret rooms as nodes in our graph model, and the airflow between neighbouring rooms as edges. One room A can be connected to another room B by (1) air exchange, (2) sucking hot air out of room B into room A's AC, or (3) both. These graphs contain between three and ten nodes. Simulink's building ventilation problem [1] sets the constraints for the remaining graph properties. These constraints are:

- Each node (room) has a room size and Boolean flags indicating whether this room is connected to AC or an outside door.
- Every connected subgraph has at least one room with AC.
- Each graph has one room with an outside door.
- There are two edge types. One edge type connects two rooms for regular airflow; each node can have up to three of these edges. The other edge type connects rooms that have neither AC nor an outside door with AC. This edge pulls hot air into the AC to cool down; every AC needs one edge of this type.

5.3.2 *Training Data Collection:* This optimisation problem is based on a MATLAB Simulink example project [2]. Simulink is a simulation tool that is commonly used in various engineering disciplines. In order to collect training data, we built a tool transforming our graph representations of floor plans into MATLAB models that the MATLAB engine can execute as simulations to determine the AC costs (the number of hours the AC runs within a month).

5.4 Algorithms

The goal is to accurately predict the outcome of the simulation for each graphs within seconds (while the simulation of one configuration takes about a minute). This way, we can explore the graph configuration space more efficiently by predicting the simulation results of most configurations and only simulating promising configurations. Below, we first describe the implementation of the embedding-based and the GNN based approach.

5.4.1 *Embedding-based Learning:* The State-of-the-Art strategy embeds the graph set with the LDP algorithm [7] implemented by the Karateclub framework [30] into a *160*-dimensional vector space, and uses the embeddings as input for a random forest regressor [25] corresponding to the default implementation by Scikit-learn [28], i.e., 100 trees in the forest, no limit on the depth of the random forest, using the mean absolute error as learning criterion.

5.4.2 *GNN-based Learning:* In Section 4.1, we identified a regression technique based on GNNs that we use to propose an alternative learning model in context of LEGCS. In this variation, we replace the two steps of embedding and regression with GNNs and pooling. By this variation, we investigate whether such an approach indeed provides the expected increase in prediction accuracy since the machine learning model is operating directly on the graph structures, instead of using a potentially limiting vector representation.

Following the guidelines in Section 4.1 about designing a GNN architecture and applying the GNN on a graph model described in Section 5.3, we have made the following decisions for the learning system:

In terms of *message passing*, we have chosen a five-dimensional feature vector for each node. The first dimension represents the room's size, the remaining ones are flags if the room is connected to an AC (and on which end of an AC edge the room is), an outside door, or neither. In terms of *aggregation*, *layer connectivity*, and *graph manipulation*, we have chosen to test three alternative architectures: GCN, GAT, and GraphSAGE.

- *GCN:* For our first GNN approach, we use a graph convolutional neural network (GCN) [24] implemented by the Pytorch Geometric framework [18]. It is a common GNN architecture [41] where we first assign a feature vector to each node. In every iteration step, we modify the feature vector of each node by aggregating it with the feature vector of the node's neighbours. Finally, we have a new set of vectors for the nodes that are learned from the neighbourhood of the nodes.
- *GAT:* Graph Attention Networks (GAT) [34] add an attention mechanism to the GNN. By assigning coefficients to neighbourhoods of each node, the network allows the capturing of complex relationships and dependencies within the graph.
- *GraphSAGE:* GraphSAGE [21] is another GNN architecture samples and aggregates features from a node's local neighbourhood instead of training individual embeddings for each node to scale GNNs for large graphs.

Particularly for our experiment, we run the GNNs with two hidden layers between input and output layers. The number of channels in those layers are determined according to a parameter grid with up to 128 channels. We picked the batch size dynamically based on the amount of training data, for more than 500 graphs the batch size was 128, for less training data smaller. We run the GNNs for 1,000 epochs. Since our *learning objective* is a prediction on whole-graph level (i.e., the AC consumption of the house configuration per month), we acquire this prediction by mean pooling the feature vectors of all nodes in the GNNs.

5.5 Learning Procedure and Performance Metrics

Eventually, we will compare the prediction quality of the two regression models by considering multiple accuracy metrics over multiple iterations with different numbers of training data to assess how efficiently each model learns.

The mean absolute error (MAE) is the average prediction error, and the mean squared error (MSE) penalises larger prediction errors more heavily than smaller ones as shown in Eq. (1).

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \quad \text{and} \quad \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

where n is the number of predictions, y represents true values, and $\hat{y}$ the predictions.

The coefficient of determination (r^2 score) compares the quality of the predictions against a simple baseline by taking the loss of the predicted values in relation to the loss of the average true value. If the coefficient is 1, the predicted values are a perfect fit on the true values, 0 means the predictions are as good as always predicting the average true value, and negative values mean that the predictions perform worse than always predicting the average true value. The coefficient of determination is described in Eq. (2):

$$r^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - avg)^2}, \qquad \text{where } avg = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{2}$$

Assessing both, mean errors and the coefficient of determination supports us in quantifying the model performance. For instance, if our model performs well for the MAE, but close to zero for the coefficient of determination, we know that our model does not perform well for the variance in the dataset, even if the MAE seems satisfactory. We then can investigate further whether our model is underfitting or if the data range in the target value is too narrow.

In the next section, we first compare the accuracy of the learning models described in the previous section. Additionally, we compare the MAE of the techniques for various sizes of training data (between 25 and 5,000 graphs). Finally, we evaluate the implications of these results for using LEGCS to explore graph configuration spaces.

6 Results and Discussion

In this section, we evaluate the performance of different variations of the proposed GNN approach (i.e., GCN, GAT, and GraphSAGE) against the state-of-the-art embedding-based approach (i.e., Emb+RF) in terms of prediction accuracy (answering the research question raised in the introduction). Afterwards we discuss what these results mean for exploring graph configuration spaces.

6.1 Accuracy of Learning

First, we assess the capability of the algorithms to learn the HVAC performance based on a sample of graph configurations and their ability to predict the HVAC performance of unknown graph configurations.

Table 1: Learning accuracy of the various learning models (in terms of MAE, MSE, and r^2 accuracy metrics).

Learning Model	MAE	MSE	r^2
Emb+RF	0.12	0.051	0.61
GCN	**0.09**	**0.023**	**0.82**
GAT	0.12	0.046	0.64
GraphSAGE	0.12	0.044	0.65

Table 1 shows the accuracy of the embedding-based learning (Emb+RF) and the three variants of GNN-based learning (i.e., GCN, GAT, and GraphSAGE) in terms of MAE, MSE, RMSE, and r^2. Here, we compare the accuracy of models we trained with 4000 graphs and validated with 1000 unseen graphs (i.e., 80% training and 20% testing sample configurations). We can see how (from training sets with about 1600 graphs) the model based on GCN outperforms the other learning models in terms of accuracy.

Figure 5 shows the prediction accuracy (assessed by the mean absolute error) for the learning strategies over various sizes of training sets. All models were trained with training sets of varying sizes (between 25 and 5000), and tested on 35,000 unseen graphs. We can see that for all approaches, the prediction error shrinks when we provide more training data to the learning models. In practice, it is unpractical (time-wise) to source large training graph samples with their respective HVAC performance given the time it takes to simulate each graph (one minute per graph). Therefore, a tradeoff must be made between more training samples and the accuracy of the training.

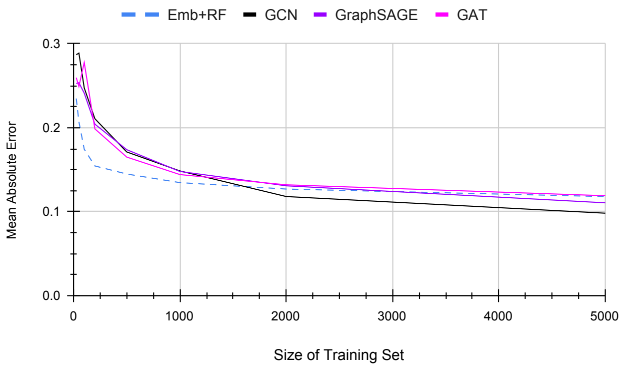


Figure 5: Accuracy of the GNN algorithm and the Emb+RF approach at predicting the HVAC performance of a testing set composed of 35,000 unseen samples while varying the size of their training set.

The embedding-based approach of using a random forest regressor operating on LDP graph embeddings (Emb+RF) delivers accurate results for low amounts of training data. We did find however that adding more and more training data did not significantly close the gap between simulation and prediction results. For our GNN approaches, we can see that we need more training data to get equally accurate predictions, but eventually GCN outperforms the first approach (Emb+RF) in terms of accuracy.

In terms of training time, on a Windows 10 machine with 16GB RAM and Intel Core i7-8665U CPU, the embedding-based approach takes less time for both steps (about 2 minutes for embedding and seconds for the regression) than the GNN-based approaches (about 15 minutes). Lowering the number of epochs or training data can reduce the calculation time as well as the use of efficient hardware (e.g., GPUs or TPUs). In the following section, we will discuss how that compares to simulating the HVAC performance (about 1 minute simulation time per graph).

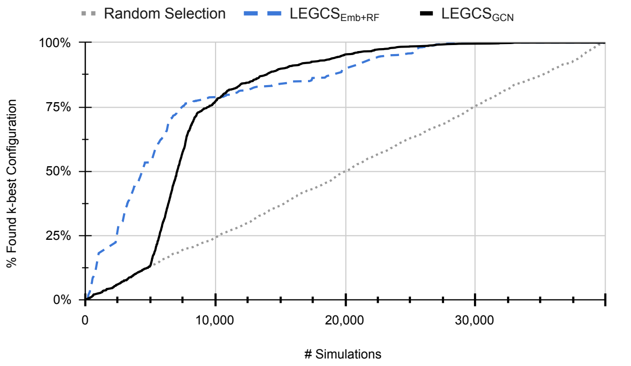


Figure 6: Identification of k best graph configurations in the set after simulating according to predictions of LEGCS with embedding and LEGCS with GCN, and a random selection.

6.2 Improvement of Graph Configuration Space Exploration

After confirming the capability of our proposed GNN (particularly for GCN) to predict the HVAC performance of unknown graph configuration, we will now assess the implications for graph space exploration.

Figure 6 shows the number of simulations we need for LEGCS with an embedding-based learning model (i.e., $LEGCS_{Emb+RF}$), a GCN-based model (i.e., $LEGCS_{GCN}$), and a random selection to find the k best graph configurations in the generated graph space (in this case, the top 3%). We can see how first the learning-based systems choose simulations at random to train the model, and as soon as the model is trained (after 200 simulations for $LEGCS_{Emb+RF}$, and 5000 simulations for $LEGCS_{GCN}$), they speed up the process of simulating the best configurations with their suggestions of which graphs to simulate next.

For our HVAC example, it takes more time to train for $LEGCS_{GCN}$ partially due to the higher computational complexity of training the model, but more importantly due to the number of necessary measurements to gather training data, i.e., $LEGCS_{Emb+RF}$ reaches a point where more training data does not lead to significantly more precise predictions already after 200 graphs, while $LEGCS_{GCN}$ takes 5000 labelled graphs to be trained. On a Linux Mint 21 Cinnamon machine with 16GB RAM and an Intel Core i7-6700 CPU where one simulation of the MATLAB engine takes about one minute, we now need about half a week to simulate 5000 graphs for training the GCN model instead of 3h 20min to simulate 200 graphs for the Emb+RF model. When we follow the suggestions of the learning models after training, we hit the point after 10,000 simulations (i.e., one week of running time) where we have identified more k best graph configurations using the GCN than with the approach relying on graph embeddings.

6.3 Implications for Product Line Engineering

We applied strategies that are established in Software Product Line Engineering and adapt them to configuration problems where we explore graph structures as configurations instead of Boolean and numerical decisions. With such an approach, a ML-supported configuration process can go beyond Euclidean configurations and be applied to other types of systems. In doing so, we can operate on more complex and ubiquitous graph structures that can capture relations within configurations. We made further advancements to the state-of-the-art by operating directly on the graph data to achieve more accurate predictions. This approach is, however, limited by the amount of training data necessary to confidently predict graph properties. Furthermore, future work on larger graphs is needed to understand the scalability of this approach and its applicability in the industry.

7 Conclusion

In this paper, we suggested that we can interpret graphs representing the structure of systems as a graph-oriented configurations of such systems. We have then drawn parallels to earlier work that links product configuration and machine learning to learn configuration spaces [29], and proposed a corresponding framework for graph-oriented configuration.

We then investigated particular techniques to allow handling such configuration spaces and optimise complex graph structures. This requires regression analysis on graphs to explore and narrow down the number of possible candidates through learning graph configuration spaces. Recent state-of-the-art work used graph embeddings to translate the graphs into a vector space as input for ML-based regression analysis–which limits the information available to the ML model.

We then proposed a new approach (i.e., $LEGCS_{GNN}$) to learn graph configuration spaces that retain the graph representations throughout the learning of graph configuration spaces. In $LEGCS_{GNN}$, we replaced the state-of-the-art learning phase (which is based on embedding and random forests) with a GNN. We have shown that $LEGCS_{GNN}$ (particularly with GCN) provides more accurate predictions, and a more efficient graph configuration space exploration than an approach based on graph embeddings. This new approach is, however, limited by a large number of training data that is required to sufficiently train the GNN model.

Future work is necessary to (i) explore potential heuristics to estimate the point when to change from an embedding-based model (that is more efficient for fewer training data) to a GNN-based model (that is more accurate for more training data), and (ii) apply more sophisticated sampling strategies to achieve precise GNN prediction results with fewer training data.

Acknowledgments

This publication has emanated from research conducted with the financial support of Taighde Éireann – Research Ireland under Grant number 13/RC/2094_2.

References

[1] 2024. Building Ventilation - MATLAB Simulink. https://www.mathworks.com/help/simscape/ug/building-ventilation.html. [Accessed 01-October-2024].

[2] 2024. Simulink - Simulation and Model-Based Design — uk.mathworks.com. https://uk.mathworks.com/products/simulink.html. [Accessed 01-October-2024].

[3] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. 2019. Learning very large configuration spaces: What matters for linux kernel sizes. *Inria Rennes-Bretagne Atlantique* (2019).

[4] Emily Alsentzer, Samuel G. Finlayson, Michelle M. Li, and Marinka Zitnik. 2020. Subgraph Neural Networks. In *Proc. of NeurIPS 2020*.

[5] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer. doi:10.1007/978-3-642-37521-7

[6] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Comput. Surv.* 50, 1 (2017), 14:1–14:45. doi:10.1145/3034827

[7] Chen Cai and Yusu Wang. 2018. A simple yet effective baseline for non-attributed graph classification. *arXiv preprint arXiv:1811.03508* (2018).

[8] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Trans. Knowl. Data Eng.* 30, 9 (2018), 1616–1637. doi:10.1109/TKDE.2018.2807452

[9] Manuel Chica, Angel A Juan Pérez, Oscar Cordon, and David Kelton. 2017. Why simheuristics? Benefits, limitations, and best practices when combining metaheuristics with simulation. *Benefits, Limitations, and Best Practices When Combining Metaheuristics with Simulation (January 1, 2017)* (2017).

[10] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proc. of VAMOS 2012*. 173–182. doi:10.1145/2110147.2110167

[11] Manuel Fernández Delgado, Manisha Sanjay Sirsat, Eva Cernadas, Sadi Alawadi, Senén Barro, and Manuel Febrero-Bande. 2019. An extensive experimental survey of regression methods. *Neural Networks* 111 (2019), 11–34. doi:10.1016/j.neunet.2018.12.010

[12] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. 2011. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Autom. Softw. Eng.* 18, 1 (2011), 77–114. doi:10.1007/S10515-010-0076-6

[13] Kien Do, Truyen Tran, and Svetha Venkatesh. 2019. Graph Transformation Policy Network for Chemical Reaction Prediction. In *Proc. of KDD 2019*. 750–760. doi:10.1145/3292500.3330958

[14] Steve Easterbrook, Janice Singer, Margaret-Anne D. Storey, and Daniela E. Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer, 285–311. doi:10.1007/978-1-84800-044-5_11

[15] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Inf. Softw. Technol.* 106 (2019), 1–30. doi:10.1016/J.INFSOF.2018.08.015

[16] David Eppstein and Denis Kurz. 2017. K-Best Solutions of MSO Problems on Tree-Decomposable Graphs. In *Proc. of IPEC 2017 (LIPIcs, Vol. 89)*. 16:1–16:13. doi:10.4230/LIPICS.IPEC.2017.16

[17] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. 2020. A Fair Comparison of Graph Neural Networks for Graph Classification. In *Proc. of ICLR 2020*. https://openreview.net/forum?id=HygDF6NFPB

[18] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[19] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein Interface Prediction using Graph Convolutional Networks. In *Proc. of NIPS 2017*. 6530–6539.

[20] José Angel Galindo, Deepak Dhungana, Rick Rabiser, David Benavides, Goetz Botterweck, and Paul Grünbacher. 2015. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Inf. Softw. Technol.* 62 (2015), 78–100. doi:10.1016/J.INFSOF.2015.02.002

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proc. of NIPS 2017*. 1024–1034.

[22] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. 2008. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer, 201–228. doi:10.1007/978-1-84800-044-5_8

[23] Angel A. Juan, W. David Kelton, Christine S. M. Currie, and Javier Faulin. 2018. Simheuristics Applications: Dealing with uncertainty in Logistics, Transportation, and other supply Chain areas. In *Proc. of WSC 2018*. IEEE, 3048–3059. doi:10.1109/WSC.2018.8632464

[24] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. of ICLR 2017*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[25] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[26] David Liben-Nowell and Jon M. Kleinberg. 2007. The link-prediction problem for social networks. *J. Assoc. Inf. Sci. Technol.* 58, 7 (2007), 1019–1031. doi:10.1002/asi.20591

[27] Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck. 2024. Learning Graph Configuration Spaces with Graph Embedding in Engineering Domains. In *Proc. of LOD 2023*. 334–348. doi:10.1007/978-3-031-53966-4_25

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *J. Syst. Softw.* 182 (2021), 111044. doi:10.1016/j.jss.2021.111044

[30] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs. In *Proc. of CIKM 2020)*. 3125–3132.

[31] Mary Shaw. 2003. Writing Good Software Engineering Research Papers. In *Proc. of ICSE*. 726–737. doi:10.1109/ICSE.2003.1201262

[32] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. 2013. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Inf. Softw. Technol.* 55, 3 (2013), 491–507. doi:10.1016/J.INFSOF.2012.07.020

[33] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Softw. Qual. J.* 20, 3-4 (2012), 487–517. doi:10.1007/S11219-011-9152-9

[34] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. *CoRR* abs/1710.10903 (2017). arXiv:1710.10903 http://arxiv.org/abs/1710.10903

[35] Shunxin Xiao, Shiping Wang, Yuanfei Dai, and Wenzhong Guo. 2022. Graph neural networks in node classification: survey and evaluation. *Mach. Vis. Appl.* 33, 1 (2022), 4. doi:10.1007/S00138-021-01251-0

[36] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proc. of International Conference on Learning Representations (ICLR)*. OpenReview.net. https://openreview.net/forum?id=ryGs6iA5Km

[37] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. 2018. Hierarchical Graph Representation Learning with Differentiable Pooling. In *Proc. of NeurIPS*. 4805–4815.

[38] Jiaxuan You, Zhitao Ying, and Jure Leskovec. 2020. Design Space for Graph Neural Networks. In *Proc. of Annual Conference on Neural Information Processing Systems (NeurIPS)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[39] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Proc. of NeurIPS 2018*. 5171–5181.

[40] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. 2020. GMAN: A Graph Multi-Attention Network for Traffic Prediction. In *Proc. of Conference on Artificial Intelligence (AAAI)*. AAAI Press, 1234–1241. https://ojs.aaai.org/index.php/AAAI/article/view/5477

[41] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. doi:10.1016/j.aiopen.2021.01.001

Modeling and Analysis of Configurable Job-Shop Scheduling Problems

Max Breit
University of Siegen
Siegen, Germany
max.breit@student.uni-siegen.de

Mathis Weiß
University of Siegen
Siegen, Germany
mathis.weiss@uni-siegen.de

Roman Obermaisser
University of Siegen
Siegen, Germany
roman.obermaisser@uni-siegen.de

Malte Lochau
University of Siegen
Siegen, Germany
malte.lochau@uni-siegen.de

Abstract

Scheduling problems constitute a widely considered class of optimization problems with many important practical applications. To solve a scheduling problem, we have to find an assignment of a given set of computational tasks (jobs) to a restricted number of resources (machines) such that all tasks are finished before their deadlines and the overall amount of time is minimal. The frequently considered class of Job-Shop Scheduling Problems (JSSP) is NP-hard and therefore not efficiently solvable for real-world problems. To tackle this issue, many heuristic approaches have been developed for approximating near-optimal solutions. These approaches usually require as input one single JSSP instance with precise knowledge about task execution times and other properties to be fixed in advance. However, many scheduling problems occurring in practice contain not-yet resolved variability (e.g., intervals of possible execution times). The corresponding family of problem instances is not tractable by most recent solution heuristics. To tackle this challenge, we propose to apply established variability modeling and analysis techniques to formally specify configurable scheduling problems. We use feature models to describe the variability within a given JSSP and we describe an encoding into a constraint linear program. This encoding allows us to apply recent CP-SAT solvers to automatically reason about satisfiability and to find optimal solutions for families of JSSP in a single run. Our experimental evaluation shows promising efficiency improvements in comparison to an instance-by-instance solution strategy.

CCS Concepts

• **Software and its engineering → Software configuration management and version control systems**; • **Theory of computation → Scheduling algorithms**.

Keywords

scheduling, feature model, constraint programming, cp-sat

ACM Reference Format:
Max Breit, Mathis Weiß, Roman Obermaisser, and Malte Lochau. 2025. Modeling and Analysis of Configurable Job-Shop Scheduling Problems. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3715340.3715431

1 Introduction

Context and Motivation. In many modern software-intensive application domains, we have to efficiently perform highly critical computations with a limited amount of computational resources. From a more abstract point of view, a *scheduling problem* consists of a finite set of partly interdependent computational tasks and a finite set of shared computational resources. The goal is to assign (schedule) task to resources so that all computations finish within a critical deadline and/or to minimize the overall amount of time [6]. One of the most widely considered classes of scheduling problems are so-called Job-Shop Scheduling Problems (JSSP). Finding an optimal solution for a given JSSP is NP-hard [9, 21] and therefore not efficiently tractable for realistic problem instances. To address this, researchers have proposed many heuristic solution techniques for approximating near-optimal solutions that scale to real-world JSSP [7]. However, most approaches consider JSSP in which all task execution times and other constraints are known and fixed. In contrast, in various application domains, we encounter scheduling problems with *unresolved variability.* Variability within scheduling problems may have multiple different sources [25, 26]. During development, *uncertainty* due to lack of knowledge of task durations some parts of the problem description may be left open. Another reason may be the need for *flexibility* to choose between alternative task implementations with different performance and/or energy consumption. Also, during the *maintainability* and *evolution* of already deployed systems, task durations may be refined due knowledge gained from run-time observations. These examples show that it is crucial to explicitly include variability within JSSP instance descriptions. The resulting *family of JSSP instances* is not tractable by recent heuristic solutions for JSSP in a single run, but instead requires analysis of each separate SSP-instance, which is infeasible for large-scaled problems with many a-priori unresolved variability parameters.

Approach. In this paper, we propose, to the best of our knowledge, the first approach to apply variability-modeling concepts and

VaMoS 2025, Rennes, France

ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715431

variability-analysis techniques to handle families of JSSP instances as one configurable JSSP. To this end, we utilize feature models [12] to describe variability of JSSP. Based on this modeling formalism, we describe an encoding of configurable JSSP into constraint linear programs. This encoding enables us to apply CP-SAT solvers to automatically reason about satisfiability of configurable JSSP [20] and find instances of optimal solutions in a single run.

Contributions. We make the following contributions.

- **Modeling Configurable JSSP.** We model families of JSSPs as configurable JSSP using feature models.
- **Tool Support for Automated Reasoning.** We describe an encoding of configurable JSSP as constraint linear programs, to apply CP-SAT solvers for automated reasoning about satisfiability and optimal solutions in a single analysis run.
- **Experimental Evaluation.** Our evaluation results show that our family-based approach [28] outperforms the instance-by-instance approach in terms of computational effort.
- **Replication Package** A GitHub repository with our tool and the JSSPs used in our evaluation is available here[1].

2 Background and Motivation

There are numerous variants of scheduling problems in the literature. We consider *job-shop scheduling problems* (JSSP), a class of widely considered optimization problems in computer science and operational research [8, 29]. A JSSP consists of a finite set of k *tasks* grouped into *jobs* and a finite set of l computing resources (*machines*) for executing tasks. Each task has an a-priori known *execution time*. The corresponding optimization problem is to find an assignment (*schedule*) of tasks to machines that minimizes the *makespan* (overall duration until every task completed). The complexity of the problem arises from the following additional properties: (1) the number l of machines is usually much smaller than the number k of tasks, which requires smart *resource sharing* and (2) there are further constraints on the *validity* of schedules. The latter may include constraints requiring tasks to be executed on specific machines, as well as precedence dependencies among tasks (e.g., some tasks must be completed before some other tasks can start).

Example 2.1. Let us assume a device with two computing resources (e.g., CPU and GPU). On that device, we run an app for streaming music, which includes as active elements a song progress bar and a volume slider. The job of updating the progress bar consists of three consecutive tasks, $A1, A2$ and $A3$. The job of adjusting the volume consists of two tasks, $B1$ and $B2$. Lastly, the job for playing the music consists of one task $C1$. All three sets of tasks of each job have to be executed in the ordering indicated by the indices. Each task has an a-priori known duration: $A3$ and $B1$ require 2 ms, $A1$ and $A2$ require 3 ms and $B2$ and $C1$ require 4 ms. Each task must be executed on a specific resource: $A1$, $B2$ and $C1$ on the CPU; $A2, A3$ and $B1$ on the GPU. The overall deadline is 14 ms.

This example is an instance from a class of JSSP called *machine scheduling problems* (MSP). To keep the proposed approach graspable, we focus on this class, formally defined as follows.

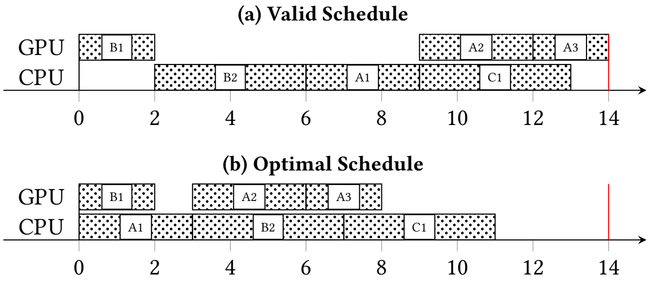


Figure 1: A valid and an optimal schedule for the example.

Definition 2.2 (Machine Scheduling Problem (MSP)). A machine scheduling problem is a tuple $(P, \prec, M, \mu, d, dl)$, where

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *tasks*,
- $\prec \subseteq P \times P$ is a strict partial order denoting *precedence* on P,
- $M = \{m_1, m_2, \ldots, m_n\}$ is a finite set of *machines*,
- $\mu : P \longrightarrow M$ assigns *machines to tasks*,
- $d : P \longrightarrow \mathbb{N}$ assigns *durations* to tasks, and
- $dl \in \mathbb{N} \cup \{\infty\}$ is the overall *deadline*.

In an MSP, every task $p \in P$ is executed on a specific machine $m \in M$, whereas the ordering of tasks of different jobs may vary. By $p \prec p'$ we denote that task p must be completed before task p' starts. By requiring $\prec$ to be a strict partial order, it is non-reflexive (i.e., tasks do not depend on themselves), asymmetric (i.e., no cyclic dependencies) and transitive (i.e., if task p depends task p' and task p' depends on task p'', then task p also depends on task p''). By $dl = \infty$ we denote that the problem has no critical deadline.

A solution s (*schedule*) of an MSP $(P, \prec, M, \mu, d, dl)$ defines starting times $s(p)$ for each task $p \in P$. For a schedule s to be *valid*, it must meet all constraints. First, if $p \prec p'$, then task p must be completed before task p' starts, which requires $s(p) + d(p) \leq s(p')$. Second, all tasks p must be completed before the deadline requiring $s(p) + d(p) \leq dl$. Finally, each machine $m \in M$ may execute at most one task at a time. Thus, any two tasks $p, p' \in P$ assigned to the same machine m must not overlap, which holds if $max(s(p), s(p')) \geq min(s(p) + d(p), s(p') + d(p'))$. In practice, we are interested in an *optimal* schedule with a minimum makespan.

Example 2.3. Two valid schedules for the JSSP from Example 2.1 are shown in Figure 1a and Figure 1b. The deadline of 14 is met by both and both use the same order of the tasks on the GPU, but differ regarding the CPU. In the first schedule, task $B2$ runs first on the CPU, but waits for completion of $B1$ on the GPU. After $B2$ is completed, $A1$ starts on the CPU on which the tasks $A2$ and $A3$ depend on the GPU. In the second schedule, the ordering of $A1$ and $B2$ is swapped, which allows starting all remaining A-tasks earlier thus reducing the makespan from 14 to 11 ms, which is optimal.

Definition 2.4 (Schedule). A *valid schedule* $s : P \longrightarrow \mathbb{N}$ for an MSP $(P, \prec, M, \mu, d, dl)$ assigns starting times to tasks such that

- $\forall p, p' \in P : p \prec p' \longrightarrow s(p) + d(p) \leq s(p')$,
- $\forall p \in P : s(p) + d(p) \leq dl$, and
- $\forall p, p' \in P : \mu(p) = \mu(p') \longrightarrow max(s(p), s(p')) \geq min(s(p) + d(p), s(p') + d(p'))$.

[1] https://github.com/Chair-of-Model-based-Engineering/ConfigurableSchedulingTool

A valid schedule s is *optimal* if $makespan = max\{s(p)+d(p)|p \in P\}$ is minimal.

Finding an optimal schedule for a JSSP is an NP-hard optimization problem [9, 21]. Thus, finding optimal solutions is computationally infeasible for larger-scaled problems. Numerous heuristic approaches have been proposed to approximate near-optimal solutions [11]. These approaches consider a specific JSSP instance. However, in practice, we encounter *variability* within JSSPs. Next, we describe how to use concepts from variability modeling and suitable analysis tools to handle families of JSSP instances.

3 Modeling Configurable Scheduling Problems

In this section, we propose an approach to model families of JSSP.

3.1 Background: Feature Models

Feature models are widely used to specify valid configuration spaces of configurable software [3]. A feature model is defined over a set F of *features*, where each feature $f \in F$ represents a user-configurable (Boolean) yes/no-configuration option. A configuration $C \subseteq F$ is a subset of selected features. A feature model further defines a set D of *dependencies* (constraints) on F to restrict the possible combinations of features in valid configurations. (e.g., selecting feature $f \in F$ in C requires or excludes some other feature $f' \in F$ in the same C).

Definition 3.1 (Feature Model). A *feature model* is a pair (F, D) consisting of a set F of *features* and a set D of *dependencies* on F. By $C(F, D) \subseteq 2^F$, we denote the set of *valid configurations* $C \subseteq F$ satisfying D. A feature model is *satisfiable* if $C(F, D) \neq \emptyset$ holds.

In the following, we employ the graphical FODA notation [12] to model variability in scheduling problems as feature diagrams.

3.2 Modeling Configurable Scheduling Problems with Feature Diagrams

In an MSP according to Def. 2.2, almost all decisions are a-priori fixed. In practice, decisions may be left open for two reasons: *uncertainty* (e.g., a task duration is not exactly known) and *flexibility* (e.g., we may choose between two alternative task implementations). For both, we consider the following types of variability:

- **Alternative task groups** allow to choose exactly one implementation variant for a task.
- **Optional tasks and machines** are not necessarily required.
- **Variable task durations** define ranges of execution times.
- **Cross-dependencies** define constraints between optional tasks and variable durations of other tasks.

To illustrate these types of variability, Figure 2 shows a configurable JSSP extending Example 2.1 as a feature diagram.

Feature Diagram Notation. Feature diagrams organize the set of features in a tree-like hierarchy, representing a child-parent dependency. Singleton child features are either *mandatory* (i.e., to be selected if the parent is selected), or they are *optional*. Sibling features may also be aggregated into groups: in *alternative* groups, exactly one features must be selected together with the parent feature, whereas in *or*-groups at least one feature is required. Dependencies between arbitrary, hierarchically unrelated, features are expressed by cross-tree constraints. Finally, we distinguish between *abstract* and *concrete* features, where abstract features are only used for structuring the feature tree omitted in configurations.

Feature-Tree Structure. In our approach, we subdivide the tree below the (mandatory) root feature, one for variability of tasks and one for the variability of machines. In addition, the singleton child node $dl = 14$ of the root feature defines the overall deadline.

Variability of Tasks. In the sub-tree on the left, the mandatory task $A1$ has a fixed execution time of 3 (attached as mandatory child feature). Task $S0$ is an optional task with a fixed duration 5. In contrast, the execution time of the mandatory task $A2$ may vary within the range of 3..5 time units, denoted by an alternative group. For the optional tasks $C1$ and $C2$ the cross-tree constraint $(C1 \wedge \neg C2) \vee (\neg C1 \wedge C2)$ denotes an alternative.

Variability of Machines. In the sub-tree on the right, the machines *CPU* and *GPU* are mandatory, whereas *RAM* is optional. Further variability among machines is possible, but not used in our example.

Dependencies of Tasks. We use cross-tree constraints for two types of dependencies: (1) tasks require particular machines to be executed on, and (2) tasks require other tasks to be completed before. For instance (1), the constraint *A1* $\Rightarrow$ *CPU* denotes that $A1$ must be executed on the *CPU*. Concerning (2), we slightly abuse cross-tree constraints to also represent the precedence order $\prec$ among tasks (although these dependencies impose temporal rather than logical constraints among tasks). For instance, *A3* $\Rightarrow$ *A2* denotes that task *A2* must end before task *A3* starts. For (2), we also consider dependencies between optional and specific durations of another task. For example the constraint *d(A3)=1* $\Rightarrow$ *S0* denotes that the duration 1 of task *A3* depends on the execution of task *S0*.

3.3 Deriving Problem Instances

A feature model (F, D) as shown in Figure 2 comprises a family of JSSP instances as its configurations. For instance, the JSSP instance shown in Figure 1 corresponds to the configuration:

$$Conf_1 = \{A1, d(A1){=}3, A2, d(A2){=}3, A3, d(A3){=}2, B1, d(B1){=}2,$$
$$B2, d(B2){=}4, C1, d(C1){=}4, CPU, GPU, dl{=}14\}$$

Let us denote this instance as $\mathcal{JSSP}_1 = (P_1, \prec_1, M_1, \mu_1, d_1, dl_1)$. Set P_1 contains tasks whose features are selected below feature *P*. The optional task *S0* is deselected in $Conf_1$ and thus not contained, whereas mandatory tasks *A1*, *B1* and *B2* are selected with fixed durations (i.e., *d(A1)=3*, *d(B1)=2* and *d(B2)=4*). In contrast, the tasks *A2* and *A3* have variable durations, where in $Conf_1$, we select *d(A2)=3* and *d(A3)=2*. Task *C1* is selected with duration *d(C1)=4* thus excluding *C2* because of the cross-tree constraint $C1 \wedge \neg C2 \vee \neg C1 \wedge C2$. Accordingly, the set M_1 contains the mandatory machines *CPU* and *GPU*, whereas the optional machine *RAM* is not selected. For the tasks selected in $Conf_1$, the mapping μ_1 to machines is due to the cross-tree constraints: $\mu_1 : A1 \mapsto CPU, A2 \mapsto GPU, A3 \mapsto GPU, B1 \mapsto GPU, B2 \mapsto CPU, C1 \mapsto CPU$. If $Conf_1$ would also the optional task *S0*, we would have to select *RAM* due to constraint *S0* $\Rightarrow$ *RAM*. Similarly, the precedence order $\prec_1$ is derived from cross-tree constraints such that $\prec_1 = \{(A1, A2), (A2, A3), (B1, B2)\}$ holds. Finally, $dl_1 = 14$ is derived from the feature below the root.

Besides $Conf_1$, the valid configuration space $C(F, D)$ of the feature diagram (F, D) in Figure 2 comprises 126 valid configurations

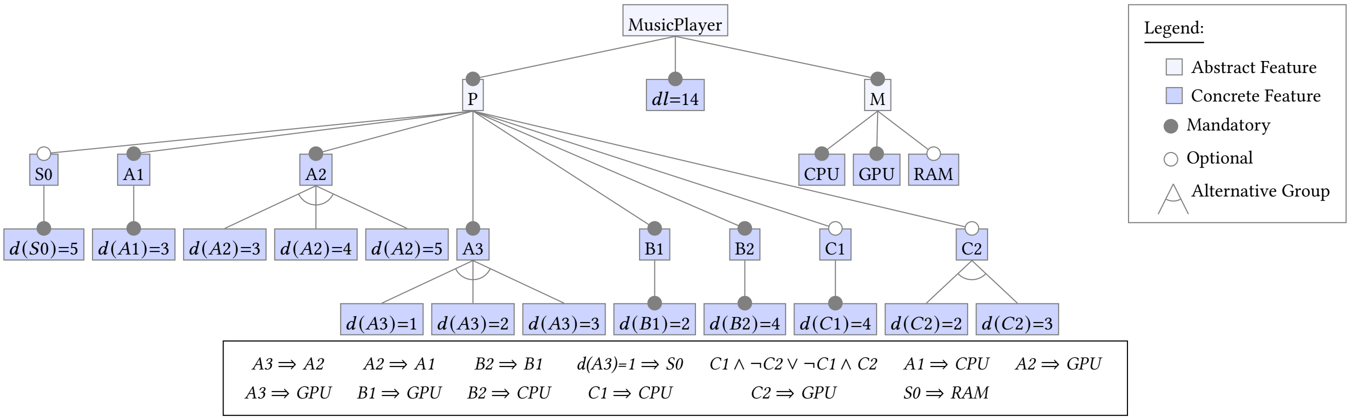


Figure 2: Feature Diagram for a Configurable Scheduling Problem

$Conf_i$. Given a feature diagram, properties of the configuration space can be analyzed with tools like FeatureIDE [13]. Besides general satisfiability of the constraints, dead/core feature analysis can be used to detect tasks, machines or durations which are, although being declared options, part of no or every valid configuration.

4 Automated Analysis of Configurable Scheduling Problems

In this section, we describe how to reason about satisfiability of configurable JSSP and how to find optimal solutions. We revisit the foundations of constraint programming and describe an encoding to constraint satisfaction problems solvable with CP-SAT solvers [20].

4.1 Family-based Analysis of Configurable Scheduling Problems

Given a configurable JSSP, we consider two types of problems:

(1) Find at least one configuration/JSSP which is satisfiable.
(2) Find the configuration/JSSP with an optimal schedule.

A instance-by-instance approach for the configurable JSSP shown in Fig. 2 would be to derive all instances $\mathcal{JSSP}_i$, $1 \leq i \leq n$, and analyze each separately. However, this is practically infeasible as the maximum number of possible instances grows exponentially in the number of features. Furthermore, many of these instances are very similar, leading to many redundant analysis runs. Thus, we aim at performing a so-called family-based analysis on all instances at once [28]. In this paper, we utilize an encoding of JSSP as a constraint satisfaction problem (CSP). Such an CSP encoding also exists for feature models [5]. This leads us to *dynamic* constraint satisfaction problems in which problem variables can be explicitly activated/deactivated by features [16]. Another possible approach would be to use a SAT encoding. However, the CSP encoding enables us to find optimal solutions [20]. The latter is not supported by standard SAT solvers.

4.2 Background: Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) [22] is defined over a finite set $\mathcal{V}$ of typed *variables*. Here, we consider Boolean and integer variables. The type of each variable is defined by a function $\mathcal{D}$ mapping variables to *domains*. We consider domains to comprise any interval $\{x \in \mathbb{N} | l \leq x \leq u\}$ of natural numbers. Boolean variables are a special case with domain $\{0, 1\}$. Third, a CSP consists of a set C of *constraints* restricting the set of valid value assignments.

Definition 4.1 (Constraint Satisfaction Problem). A *constraint satisfaction problem* is tuple $(\mathcal{V}, \mathcal{D}, C)$, where

- $\mathcal{V} = \{v_1, ...v_n\}$ is a finite set of n *variables*,
- $\mathcal{D} : \mathcal{V} \rightarrow \{d \in \mathbb{N} | l \leq d \leq u\}$ maps each variable v_i to a *domain* $D(v_i) = D_i$ of possible values,
- $C \subseteq 2^{D_1 \times \ldots \times D_n}$ is a finite set of *constraints*.

4.3 CSP Encoding of Configurable Scheduling Problems

We now describe the CSP encoding, again, using our running example shown in Figure 2 for illustrative purposes.

Variables. For each optional machine m_i and each optional task p_i, we introduce a Boolean variable $active_i$ indicating whether m_i is used or p_i is executed in a JSSP instance. Applied to our example, this includes the variables $active_{S0}$, $active_{C1}$, and $active_{C2}$ for the optional tasks $S0$, $C1$ and $C2$, and $active_{RAM}$ for the optional machine *RAM*. For mandatory tasks and machines, no such decision variables are required. Moreover, we introduce for *every* task p_i three integer variables, denoting the to-be-decided start time ($start_i$), the (variable) duration ($duration_i$), and the end time (end_i). Finally, we introduce a variable *makespan* for the make-span. Hence, the CSP of our example comprises the following variables:

$$\begin{aligned}\mathcal{V}_{MusicPlayer} = \{&active_{S0}, active_{C1}, active_{C2}, active_{RAM},\\ &start_{S0}, duration_{S0}, end_{S0},\\ &\ldots\\ &start_{C2}, duration_{C2}, end_{C2}, makespan\}.\end{aligned}$$

Domains. The variables $active_i$ are typed over the domain of $\{0, 1\}$. The domain of the integer variables $start_i$, end_i and *makespan* is $\{0, \ldots, dl\}$, where dl is the deadline. If $dl = \infty$, the sum of the highest durations of each task may be used. The domain of the variables $duration_i$ varies for each task p_i. For instance, task *S0* has a fixed execution time of 5 thus the domain of $duration_{S0}$ is $\{5\}$. For task *A3*, having a configurable duration, the domain is $\{1, 2, 3\}$. The CSP for our example comprises the following domains:

$$\begin{aligned}\mathcal{D}_{MusicPlayer} : & active_i \mapsto \{0, 1\}, start_i \mapsto \{0, \dots, dl\},\\ & end_i \mapsto \{0, \dots, dl\}, duration_{S0} \mapsto \{5\},\\ & \dots\\ & duration_{C2} \mapsto \{2, 3\}, makespan \mapsto \{0, \dots, dl\}.\end{aligned}$$

We distinguish two kinds of constraint in our CSP encoding. First, variability constraints ensure that the values assignments of Boolean variables correspond to valid configurations according to Def. 3.1. Second, validity constraints ensure that the value assignments of integer variables correspond to valid schedules according to Def. 2.4.

Variability Constraints. First, we require an optional machine to be active, if at least one job running on this machine is selected. This can be encoded in CSP by considering the maximum value of all variables $active_i$ corresponding to tasks p_i running on the machine. For the optional machine *RAM*, we obtain the constraint:

$$max(active_{S0}) = active_{RAM}.$$

To encode alternative tasks (*C1*, *C2*), we require the sum of the values of $active_i$ variables to be equal to 1. For *C1* and *C2* we get:

$$active_{C1} + active_{C2} = 1.$$

The remaining cross-tree constraints are encoded, accordingly. The require-dependency $duration_{A3} = 1$ to task *S0* is denoted as:

$$(duration_{A3} = 1) \longrightarrow active_{S0}.$$

Validity Constraints. For each task p_i, the values of variables $start_i$, $duration_i$, and end_i define the execution time intervals. To ensure that the intervals respect the given task durations, we add:

$$start_i + duration_i = end_i.$$

If a task is optional, the constraint must only hold if the task is active. For instance, for *S0*, we add the guarded constraint:

$$active_{S0} \longrightarrow (start_{S0} + duration_{S0} = end_{S0}).$$

These constraints are added for each mandatory and optional task:

$$\begin{aligned}active_{S0} \longrightarrow (start_{S0} + duration_{S0} = end_{S0}),\\ start_{A1} + duration_{A1} = end_{A1},\\ \dots\\ active_{C2} \longrightarrow (start_{C2} + duration_{C2} = end_{C2}).\end{aligned}$$

Next, we add constraints to ensure the order $\prec_{MP}$. For instance, task *A1* must precede task *A2* by:

$$end_{A1} \leq start_{A2}$$

and, similarly, for the other tasks:

$$\begin{aligned}end_{A2} \leq start_{A3},\\ end_{B1} \leq start_{B2}.\end{aligned}$$

Moreover, we encode that tasks executed on the same machine do not overlap. For instance, *A1* and *B2* run on the *CPU*. Hence, we require that either *A1* finishes before *B2* starts, or vice versa:

$$end_{A1} \leq start_{B2} \lor end_{B2} \leq start_{A1}.$$

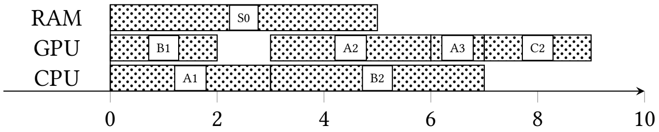


Figure 3: Optimal Solution for the Example

We add such a constraint for each pair of tasks on a machine. For instance, for the machine *CPU*, with tasks *A1*, *B2* and *C1*, we add:

$$\begin{aligned}end_{A1} \leq start_{B2} \lor end_{B2} \leq start_{A1},\\ end_{A1} \leq start_{C1} \lor end_{C1} \leq start_{A1},\\ end_{B2} \leq start_{C1} \lor end_{C1} \leq start_{B2}.\end{aligned}$$

and similarly for the *GPU*. Finally, we add:

$$makespan = max(end_i)$$

to ensure that the variable *makespan* corresponds to the point in time when the last task is finished.

Analysis. Given the CSP encoding

$$CSP_{MusicPlayer} = (V_{MusicPlayer}, D_{MusicPlayer}, C_{MusicPlayer})$$

of our example, we can apply a CP-SAT solver. If we ask the solver for a variable assignment satisfying all constraints, we get one JSSP instance (given by the values of the variables $active_i$) together with a valid schedule for that instance (values of variables $start_i$, $duration_i$ and end_i) such as the one shown in Fig. 1. To find the instance permitting an optimal solution, we ask to minimize the value of *makespan*. Applied to our example, an optimal schedule is shown in Fig. 3. The make-span is 9 which requires the execution of the optional task *S0* which allows a smaller duration of task *A*3.

5 Experimental Evaluation

In this section, we describe the evaluation results gained from applying our tool implementation to a collection of synthetic examples.

5.1 Research Questions

We evaluate the family-based approach in comparison to the instance-based approach. We investigate the following research questions.

- **(RQ1)** What is the computational effort to check satisfiability of a configurable JSSP using family-based analysis as compared to instance-based analysis?
- **(RQ2)** What is the computational effort to find an optimal solution for a configurable JSSP using family-based analysis as compared to instance-based analysis?
- **(RQ3)** How does the amount of optional tasks, alternative tasks and other cross-tree constraints influence the computational effort for family-based analysis?
- **(RQ4)** How does the family-based analysis scale to larger problems

5.2 Methods and Experimental Design

Subject Systems. We first explain the naming conventions for synthetically generated feature diagrams representing configurable JSSP as shown in Figure 2 to serve as our subject systems. Each name starts with *J*, followed by the number of jobs, followed by *T* and the number of non-optional tasks. This is followed by M, denoting the number of machines. Next, an (optional) O is followed by the number of optional tasks, followed by an (optional) A indicating the overall number of tasks being in alternative groups. Finally, D is followed by the number of additional cross-tree-constraints.

We generated the subjects as follows: We start with *J2T10M2* as a base model. For RQ3, we added 1, 2, 4, 8 and 16 optional tasks and use the same scheme for alternative tasks. To investigate additional cross-tree-constraints, we generated corresponding subjects based on the subject *J2T10M2O16A16*, thus creating subjects *J2T10M2O16A16D1* up to *D16*. To investigate scalability we used the subject *J4T25M4O16A16D10* and increased its variable parameters up to the subject *J4T100M4O128A128D40* with 100 tasks. We also injected unsatisfiable subjects into our corpus, marked by postfix U. For this, we used the subjects *J2T10M2O1* to *J2T10M2O16* and reduced the deadline until no more valid schedule exists.

Data Collection. We collected the execution times from the analysis runs on all subject systems, consisting of two steps:

(1) Read-step to import and encode the subjects, being either one configurable JSSP (family-based analysis) or a list of JSSP instances (instance-based analysis).
(2) Solver-step applying a CP-SAT solver either once on a configurable JSSP (family-based analysis) or to each JSSP instance separately (for instance-based analysis).

For the instance-based analysis, we do not necessarily have to analyze every instance for RQ1: once we found a valid schedule, we stopped and recorded the number of instances considered. For RQ2, we must consider all instances. Although CP-SAT works deterministically, we repeated each measurement 5 times after one non-measured warm-up run and used the average value to reduce the influence of fluctuations. We used a timeout of 30 minutes.

Measurement Setup. Our tool is written in Java 19 [19] and built with Maven [2]. We further used CP-SAT [20], version 9.10.4067, the Universal-Variability-Language [27], and OpenCSV [23], version 5.9., and Eclipse 2024-03 with FeatureIDE [14] in version 3.11.1, to create the feature diagrams. All experiments were executed on a computer running Linux Mint 21.1., equipped with an Intel i5-8250U (4x 1,60 GHz) as a CPU and 8 GB (2400 MHz) RAM.

5.3 Results

Results for the family-based approach are shown using solid lines, those of the instance-based analysis use dotted lines.

RQ 1: Valid Solution. First, we consider the results for subjects with optional tasks, shown in Fig. 4 in blue. For the instance-based approach, we also report the number of instances considered until a valid schedule was found. Subject *J2T10M2O1* requires the fewest run-time for both approaches. The family-based analysis took 177.6 ms and the instance-based approach took 126.8 ms. For the subject *J2T10M2O8*, both approaches required the most run-time, where the family-based approach took 200.2 ms and the instance-based approach took 149.6 ms. Figure 4 shows the results for subjects with alternative groups in red. Both approaches were fastest for subject *J2T10M2A4*. The family-based approach took 194.8 ms and the instance-based approach took 127 ms. For subject *J2T10M2A16*, the family-based approach required the longest run-time (231.4 ms), while the instance-based approach required the longest time for *J2T10M2A2* (184.6 ms). The instance-based approach always succeeded on the first instance. The results for subjects with no valid solutions are shown in Figure 5 in blue using a logarithmic scale. Run-time of the family-based approach varies minimally, while the instance-based approach timed out for the subject *J2T10M2O8*.

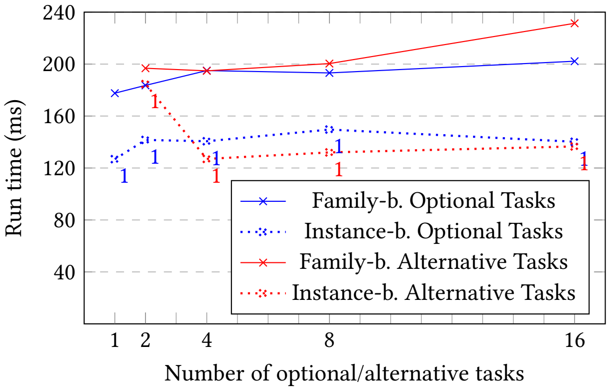


Figure 4: RQ1: Measured run times for determining a feasible solution for problems with optional or alternative tasks.

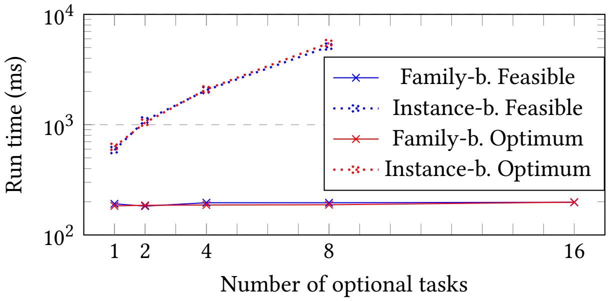


Figure 5: RQ1 & RQ2: The run-times for unsatisfiable JSSP.

RQ 2: Optimal Solution. The results for problems with optional tasks are shown in Figure 6 in blue (logarithmic scale). Subject *J2T10M2O1* was solved the fastest by both approaches (family-based 188.6 ms, instance-based 1.114.4 ms) and subject *J2T10M2O16* took the longest (family-based 200.2 ms, instance-based 30 minutes time-out). The instance-based approach considered between 11 and 1 instance. The result with alternative tasks are shown in red. Subject *J2T10M2A2* was solved fastest (family-based 189.4 ms, instance-based 1,447 ms) and subject *J2T10M2A16* took the longest (family-based 229.8 ms, instance-based 79,909 ms). The instance-based approach considered between 15 and 1 instances. The results for subjects without valid solutions are almost identical to RQ1.

RQ 3: Impact of Dependencies. Figure 7 uses the same representation as the Figures 4 and 6. We omitted the results of the instance-based approach for better readability. Regarding subjects with an increasing number of optional tasks, the run-times increased by 24.6 ms to find a valid schedule and for an optimal

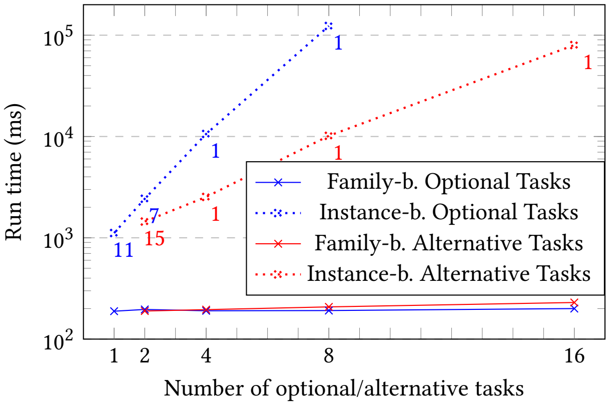


Figure 6: RQ2: Measured run times for determining optimal schedule for problems with optional or alternative tasks.

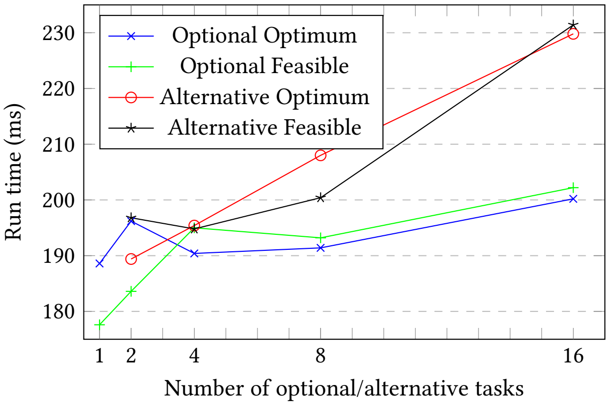


Figure 7: RQ3: Influence of optional tasks and alternative tasks on the family-based search.

Name	Optimum	Feasible
J2T10M2O16A16	276,2 ms	255,8 ms
J2T10M2O16A16D1	272,8 ms	243,8 ms
J2T10M2O16A16D2	266,6 ms	245,4 ms
J2T10M2O16A16D4	274,2 ms	250,0 ms
J2T10M2O16A16D8	259,0 ms	247,8 ms
J2T10M2O16A16D16	282,4 ms	264,6 ms

Table 1: Search times for feasible schedules (every variability)

schedule by 13.6 ms. For subjects with alternative tasks, the runtime for finding a valid schedule increased by 34.6 ms and for an optional schedule by 40.4 ms. Lastly, for subjects with duration cross-tree-constraints, let us consider Tab. 1. The fastest run-time to find a valid schedule was observed for *J2T10M2O16A16D1*, whereas subject *J2T10M2O16A16D16* took the longest time (20.8 ms more). To find an optimal schedule, *J2T10M2O16A16D8* consumed the least time and *J2T10M2O16A16D16* took 23.4 ms longer.

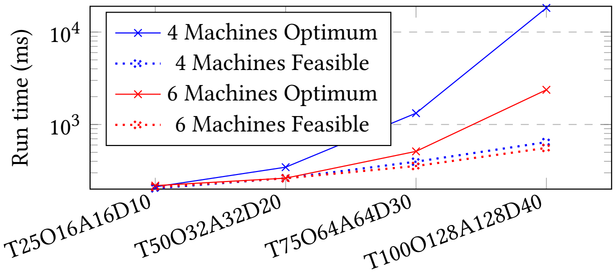


Figure 8: RQ4: Performance of the family-based analysis on bigger problems.

RQ 4: Scalability. The run times for larger problems solved by the family-based analysis are shown in Fig. 8. The run times increase with an increasing problem size and the search for an optimal solution takes longer than the search for a feasible solution. The run times decreased if the number of machines is increased. The optimal analysis for subject *J4T25M4O16A16D10* was the fastest with 204 ms. We observed the longest run time for an optimal schedule for the problem *J4T100M4O128A128D40* with 18,170.4 ms.

5.4 Discussion

RQ1. Considering the result in Figure 4, the differences are small. The instance-based analysis is on average 89.1 ms faster for problems with optional tasks and 59.12 ms faster for alternative tasks. The generally higher overhead in the CSP encoding used for the family-based approach leads to longer run times. Increasing problem size has no noticeable effect. The results look different for unsatisfiable subjects shown in Figure 5, for which the run-times of the instance-based approach strongly increase with problem sizes. The solver has to analyze every JSSP instance individually to finally conclude that no valid solution exists. In contrast, the run-times for the family-based analysis are similar as for subjects with solutions.

Answer to RQ1 (Valid Schedule)

For satisfiable problems, the instance-based approach performs slightly better. For unsatisfiable problems, the instance-based approach fails on large problems. The family-based approach performs similar to satisfiable cases.

RQ2. Let us first consider the results in Figure 6. While the run-times for the family-based analysis varies little, the run-times for the instance-based analysis increases approximately exponentially with problem sizes. Even though an optimal schedule may be found early, all other instances must be always analyzed, too, to prevent from missing a better solution. We observe similar results for unsatisfiable problems as, again, every instance must be considered to finally conclude that no valid (and thus no optimal) solution exists.

Answer to RQ2 (Optimal Schedule)

The run-time of the instance-based approach increases exponentially with increasing problem sizes. The family-based analysis is always faster, with near-stable run-times.

RQ3. The influence of optional and alternative tasks is shown in Figure 7. With an increasing number of alternative tasks, the run-times increase from 189,4 ms to 229,8 ms. For optional tasks we only observe a slight upwards-trend for the run-times, and it is not monotone. Finally, the influence of duration-cross-tree-constraints is shown in Table 1. While the problem causing the longest run-time is the one with 16 duration-cross-tree-constraints, while the baseline-problem took the second-longest. Hence, we cannot observe an obvious impact only by the number of constraints.

Answer to RQ3 (Impact of Dependencies)

Only the number of alternative tasks increases the run-times, whereas other concepts have no clear impact.

RQ4. The results in Fig. 8 show that the measured run times still do not exceed a few seconds but grow much faster for increasing problem sizes. In contrast, the instance-based approach consumes multiple minutes for small-scaled problems. The run time decreases if we increase the number of machines from 4 to 6. Less machines lead to more possible task-order-combinations per machine.

Answer to RQ4 (Scalability to Larger Problems)

With an increased problem size, the run time of the family-based approach rises, but remains within tolerable ranges. The run time decreases with more machines.

5.5 Threats to Validity

Internal Validity. The most serious internal threat is the selection of synthetic subject systems. This allows us to conduct a systematic study for relevant problem parameters. In future work, it needs to be investigated if our selection is representative. This does not diminish the general insights. Also, the fine-grained knowledge required to model configurable scheduling problems may not be available in practice. This is a well-known drawback of all static scheduling approaches, but does not harm the validity of our approach in general. In contrast, our approach explicitly aims at counter-acting this by allowing variability within problem definitions, to reflect uncertainty. Our approach currently only supports MSP, but can be easily extended to other problem classes in future work.

External Validity. Our experiments depend on the validity of the CP-SAT solver and FeatureIDE, which may be considered reliable.

6 Related Work

Recent works scheduling problems are subdivided into two main categories: dynamic scheduling and static scheduling. In dynamic scheduling approaches, schedules are computed at run-time [10, 15, 24] . While this offers precise knowledge about the current run-time context, optimal schedules are not computable on-the-fly due to the NP-hardness of the problem. Various heuristics are used to approximate near-optimal solutions. Our approach is closer to static approaches that aim to precompute optimal schedules at design time [18]. Here, schedules are computed off-line using techniques like integer linear programming and dynamic programming [1]. In a comprehensive survey, recent reasoning techniques of JSSP are summarized [29]. Most of these approaches focus on one specific instance presuming precise knowledge of task durations etc.

There are two research directions to handle forms of variability in scheduling problems: (1) variability as an extension of classic scheduling approaches, and (2) variability within scheduling approaches modeled using techniques as done in our work. For (1), Sorkhpour et al. [25, 26] consider meta-scheduling algorithms based on static approaches, supporting dynamic changes at run-time. Tasks are scheduled with slack times and slowed down at run-time if this is feasible to enable energy-efficient scheduling. This is can be modeled in terms of variable task execution times Variability like optional or alternative tasks as in our approach is not considered. Instead, Sorkhpour et al. focus on optimizing NFP like energy consumption using Mixed-Integer Quadratic Programming, which might also be interesting future extension of our approach. Muoka et al. [17] build upon this to tackle high computational effort at design time as well as high memory consumption of schedule graphs at run-time. To sum up (1), these works consider variability as sporadic changes in the context rather than being part of the definition of a scheduling problem as we do. For (2), Belategi et al. [4] use model-driven concepts in software product-line engineering for modeling real-time embedded systems. They consider variability of hard- and software components and use MARTE model analysis to validate the system design regarding performance and schedulability properties. However, variability at the level of scheduling problems is not explicitly addressed.

7 Conclusion

We proposed a novel approach to model and analyze families of scheduling problems using variability modeling and constraint programming. Our evaluation shows that the approach outperforms an instance-based analysis in terms of computational effort. As a future work, we plan to extend the approach in various directions. First, the class of scheduling problems can be further extended from MSP/JSSP to more complicated ones (e.g., tasks may be assigned to different machines with specific execution times, preemptive scheduling etc.). Moreover, we may extend the modeling capabilities for variability of scheduling problems (e.g., configurable precedence orders and deadlines for tasks, a-priori unbounded configurable execution times for tasks). These extensions also require enhanced encodings and potentially other constraint-solvers. Another interesting branch are (configurable) optimization goals beyond makespans. We intend to conduct real-world case studies to evaluate the effectiveness of our scheduling in different practical contexts.

Acknowledgments

The authors thank our reviewers for their constructive feedback. This work was funded by the German Research Foundation within the project *Co-InCyTe* (LO 2198/4-1).

References

[1] Yasmina Abdeddaïm, Abdelkarim Kerbaa, and Oded Maler. 2003. Task Graph Scheduling Using Timed Automata. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, Nice, France, 237. https://doi.org/10.1109/IPDPS.2003.1213431

[2] Apache Software Foundation. 2024. *Maven*. Apache Software Foundation. https://maven.apache.org/

[3] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-oriented software product lines*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7

[4] Lorea Belategi, Goiuria Sagardui, and Leire Etxeberria. 2011. Model based analysis process for embedded software product lines. In *International Conference on Software and Systems Process, ICSSP 2011, Honolulu, HI, USA, May 21-22, 2011, Proceedings*, David Raffo, Dietmar Pfahl, and Li Zhang (Eds.). ACM, Honolulu, HI, USA, 53–62. https://doi.org/10.1145/1987875.1987886

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35 (09 2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[6] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright. 1999. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications 7*. IEEE Computer Society, San Jose, CA, USA, 361–378. https://doi.org/10.1109/DCFTS.1999.814306

[7] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.* 43, 4 (2011), 35:1–35:44. https://doi.org/10.1145/1978802.1978814

[8] H. Fisher and G.L. Thompson. 1963. Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules. In *Industrial Scheduling*, J.F. Muth and G.L. Thompson (Eds.). Prentice-Hall, Englewood Cliffs, Hoboken, New Jersey, U.S., 225–251.

[9] Joaquim A.S. Gromicho, Jelke J. van Hoorn, Francisco Saldanha da Gama, and Gerrit T. Timmer. 2012. Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research* 39, 12 (2012), 2968–2977. https://doi.org/10.1016/j.cor.2012.02.024

[10] Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. 2019. Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores. *IEEE Trans. Parallel Distributed Syst.* 30, 10 (2019), 2283–2295. https://doi.org/10.1109/TPDS.2019.2910525

[11] Essam H. Houssein, Ahmed G. Gad, Yaser Maher Wazery, and Ponnuthurai Nagaratnam Suganthan. 2021. Task Scheduling in Cloud Computing based on Metaheuristics: Review, Taxonomy, Open Challenges, and Future Trends. *Swarm Evol. Comput.* 62 (2021), 100841. https://doi.org/10.1016/J.SWEVO.2021.100841

[12] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Carnegie Mellon University, Software Engineering Institute. https://insights.sei.cmu.edu/library/feature-oriented-domain-analysis-foda-feasibility-study/ Accessed: 2024-Jul-18.

[13] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, Vancouver, Canada, 611–614. https://doi.org/10.1109/ICSE.2009.5070568

[14] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, Vancouver, BC, Canada, 611–614. https://doi.org/10.1109/ICSE.2009.5070568

[15] Basharat Mahmood, Naveed Ahmad, Majid Iqbal Khan, and Adnan Akhunzada. 2021. Dynamic Priority Real-Time Scheduling on Power Asymmetric Multicore Processors. *Symmetry* 13, 8 (2021), 1488. https://doi.org/10.3390/SYM13081488

[16] Sanjay Mittal and Brian Falkenhainer. 1990. Dynamic Constraint Satisfaction Problems. In *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 - August 3, 1990, 2 Volumes*, Howard E. Shrobe, Thomas G. Dietterich, and William R. Swartout (Eds.). AAAI Press / The MIT Press, Boston, Massachusetts, USA, 25–32. http://www.aaai.org/Library/AAAI/1990/aaai90-004.php

[17] Pascal Muoka, Daniel Onwuchekwa, and Roman Obermaisser. 2021. Adaptive scheduling for time-triggered network-on-chip-based multi-core architecture using genetic algorithm. *Electronics* 11, 1 (2021), 49.

[18] Roman Obermaisser. 2009. Time-Triggered Communication. In *Networked Embedded Systems - Volume 2 of the Embedded Systems Handbook*, Richard Zurawski (Ed.). CRC Press, USA, 14. https://doi.org/10.1201/9781439807620.CH14

[19] Oracle Corporation. 2022. *Java 19*. Oracle Corporation. https://www.oracle.com/java/technologies/javase/jdk19-archive-downloads.html

[20] Laurent Perron and Frédéric Didier. 2024. *CP-SAT*. Google. https://developers.google.com/optimization/cp/cp_solver/

[21] Poria Pirozmand, Ali Asghar Rahmani Hosseinabadi, Maedeh Farrokhzad, Mehdi Sadeghilalimi, Seyed Saeid Mirkamali, and Adam Slowik. 2021. Multi-objective hybrid genetic algorithm for task scheduling problem in cloud computing. *Neural Comput. Appl.* 33, 19 (2021), 13075–13088. https://doi.org/10.1007/S00521-021-06002-W

[22] Francesca Rossi, Peter van Beek, and Toby Walsh. 2008. Constraint Programming. In *Handbook of Knowledge Representation*, Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter (Eds.). Foundations of Artificial Intelligence, Vol. 3. Elsevier, Amsterdam, NL, 181–211. https://doi.org/10.1016/S1574-6526(07)03004-0

[23] Andrew Rucker Jones Scott Conway. 2024. *OpenCSV*. sourceforge. https://opencsv.sourceforge.net/

[24] Tomoki Shimizu, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama. 2022. A Fair-Policy Dynamic Scheduling Algorithm for Moldable Gang Tasks on Multicores. In *11th Mediterranean Conference on Embedded Computing, MECO 2022, Budva, Montenegro, June 7-10, 2022*. IEEE, Budva, Montenegro, 1–4. https://doi.org/10.1109/MECO55406.2022.9797088

[25] Babak Sorkhpour. 2019. *Scenario-based meta-scheduling for energy-efficient, robust and adaptive time-triggered multi-core architectures*. Ph. D. Dissertation. University of Siegen, Germany. http://dokumentix.ub.uni-siegen.de/opus/volltexte/2019/1471/

[26] Babak Sorkhpour, Ayman Murshed, and Roman Obermaisser. 2017. Meta-scheduling techniques for energy-efficient robust and adaptive time-triggered systems. In *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. IEEE, Tehran, Iran, 0143–0150. https://doi.org/10.1109/KBEI.2017.8324961

[27] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet another textual variability language? a community effort towards a unified language. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) *(SPLC '21)*. Association for Computing Machinery, New York, NY, USA, 136–147. https://doi.org/10.1145/3461001.3471145

[28] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47 (07 2014), 1–45. https://doi.org/10.1145/2580950

[29] Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. 2022. A survey of job shop scheduling problem: The types and models. *Computers & Operations Research* 142 (2022), 105731. https://doi.org/10.1016/j.cor.2022.105731

Causality-based Explanations for Feature Model Configuration

Alexander Felfernig
Institute of Software Technology
Graz University of Technology
Graz, Austria
alexander.felfernig@tugraz.at

Damian Garber
Institute of Software Technology
Graz University of Technology
Graz, Austria
dgarber@ist.tugraz.at

Viet-Man Le
Institute of Software Technology
Graz University of Technology
Graz, Austria
vietman.le@ist.tugraz.at

Sebastian Lubos
Institute of Software Technology
Graz University of Technology
Graz, Austria
slubos@ist.tugraz.at

Abstract

Feature model (FM) configuration can be supported on the basis of different reasoning approaches such as SAT solving, constraint solving, and answer set programming (ASP). To better understand the reasons of including or excluding specific features, feature model configurations (or parts thereof) need to be explained to the user. In this paper, we introduce an algorithmic approach to determine *minimal causality-based explanations* which refer to those customer requirements and constraints directly responsible for a feature model configuration. This approach helps to create more transparency and understandability of feature model configuration by determining those attributes and constraints directly responsible for a specific configuration result.

ACM Reference Format:
Alexander Felfernig, Damian Garber, Viet-Man Le, and Sebastian Lubos. 2025. Causality-based Explanations for Feature Model Configuration. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715438

1 Introduction

Feature models (FMs) are often used for the representation of variability properties of highly-variant software and hardware systems [1–3, 9, 12, 17]. Formal FM representations (e.g., SAT problems [14] or constraint satisfaction problems (CSPs) [4]), can then be used to support users of feature model configurators to find a solution for the underlying feature model configuration task. Compared to SAT-based representations, CSP-based representations allow a more flexible constraint representation by supporting, for example, a direct representation of logical equivalences and implications [9].

Regardless of the knowledge representation type, it is crucial to provide users with explanations of configurations [5, 11]. These explanations serve various purposes, including building trust in the configuration, enhancing the user's domain knowledge, clarifying situations where no solution exists, taking into account sustainability aspects, supporting consensus, and persuading users to incorporate specific features into the configuration [10, 18, 21, 26–28]. In this paper, we focus on *causality*, i.e., which user requirements and constraints (hierarchical relationships and cross-tree constraints) are responsible for a particular FM configuration. To increase *configuration process transparency*, we provide (subset-) minimal explanations for selected features of an FM configuration.

For example, if a car configurator user selects the feature *adaptive cruise control* and a final configuration also includes a *safety package* due to a constraint specifying that *adaptive cruise control* requires *safety package*, the selection of *adaptive cruise control* is a basic cause for including the *safety package.* We are interested in *minimal causes*, i.e., only constraints *responsible* for the inclusion (or exclusion) of a specific feature are part of an explanation (*explanation minimality*). A *car color* feature might not be part of such an explanation, since it is typically independent of the safety features and could thus be regarded as an irrelevant element when explaining safety features.

The contributions of this paper are the following: (1) we introduce the concept of minimal causal explanations to feature model configuration, (2) we propose an algorithm to generate such explanations, and (3) we demonstrate the applicability of the algorithm in interactive configuration settings.

The remainder of this paper is organized as follows. In Section 2, we introduce a simple example feature model that supports the configuration of survey software instances. In Section 3, we introduce the concept of minimal causality-based explanations. The CXPLain algorithm for the determination of minimal causality-based explanations is introduced in Section 4. In Section 5, we present the results of a performance evaluation showing the applicability of our approach in interactive configuration settings. Furthermore, Section 6 includes a discussion of threats to validity. The paper is concluded with a discussion of open research issues in Section 7.

2 Example Feature Model

In the following, we introduce an example feature model from the domain of *survey software configuration* which will be used as a working example throughout the paper (see Figure 1). The features part of the FM are organized in a hierarchical fashion basically specified by the following relationships: (1) *mandatory* relationships

VaMoS 2025, Rennes, France

ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715438

specify that specific features have to be included in a configuration (e.g., the *payment* feature is part of every configuration), (2) *optional* relationships express that specific features can be part of a configuration (e.g., the *statistics* feature can be included in a configuration), (3) *alternative* relationships specify that from a given set of sub-features exactly one has to be included in a configuration given that the parent feature is included (e.g., one of the alternative licenses has to be included), and (4) *or* relationships specify that at least one of a given set of sub-features has to be included given that the parent feature is included (e.g., question answering (QA) can be supported by multiple choice questions but as well on the basis of multi-media based representations). In addition to the discussed relationships, cross-tree constraints can be applied to specify further properties (constraints): (1) *excludes* constraints between two features indicate that these features must not be included in the same configuration (e.g., if *no license* is selected as payment model, *ABtesting* must not be included in the same configuration), (2) *requires* constraints between two features f_a and f_b indicate that if f_a is included, feature f_b must be included as well in the final configuration. For example, the inclusion of the feature *ABtesting* requires the inclusion of the *statistics* feature.

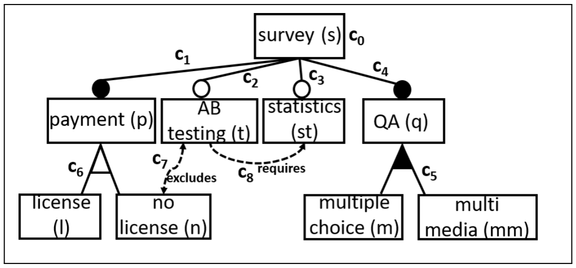


Figure 1: An example feature model (*survey software*).

To support FM configuration, feature models have to be translated into a formal representation. Related example representations are SAT problems [13, 23], answer set programs (ASPs) [24], and constraint satisfaction problems (CSPs) [4, 6, 7]. In this paper, we represent feature models on the basis of CSPs. For a discussion of rules defining the translation of feature models into a logic-based representations we refer to [4, 9]. For generating the constraint-based representations discussed in this paper, we follow those rules.

3 Minimal Causality-based Explanations of FM Configurations

Let us assume that a user of a feature model configurator decides to include the feature *ABtesting* (t) but does not specify intended inclusions or exclusions of the other features. This is a typical situation since specifically in the context of large feature spaces users do not want to specify all features. On the basis of the specified *user requirement* $t = true$, an FM configurator could propose the following FM configuration: $\{s = true, p = true, l = true, n = false, t = true, st = true, q = true, m = true, mm = false\}$.

For the configurator user in our working example, it could be of interest to learn more, for example, about the automated inclusion of the *license* feature (l), i.e., why a license is needed in this context. A corresponding explanation could be: (1) *ABtesting* has to be included since it is a user requirement, (2) it does not allow a free license model (feature *nolicense*) (constraint c_7), (3) every configuration has to include a payment model (p) (constraint c_1), (4) either l or n has to be included (constraint c_6), and (5) every configuration must have the root feature s included (root constraint c_0). With this, we provide a simple explanation (see also Table 2) for the inclusion of a specific feature (in our case, *license*). A similar approach can also be used to explain combinations of included or excluded features.

Before discussing causality-based explanations in detail, we introduce a CSP-based formalization of the feature model shown in Figure 1 – the derived CSP is depicted in Table 1. In this context, c_0 is regarded as root constraint (part of every feature model) assuring that no empty feature configurations can be generated.

Table 1: CSP derived from the feature model in Figure 1. We use abbreviated variable names, e.g., survey (s).

ID	description
c_0	$s = true$
c_1	$p \leftrightarrow s$
c_2	$t \longrightarrow s$
c_3	$st \longrightarrow s$
c_4	$q \leftrightarrow s$
c_5	$q \leftrightarrow (m \vee mm)$
c_6	$p \leftrightarrow (l \wedge \neg n \vee \neg l \wedge n)$
c_7	$\neg(n \wedge t)$
c_8	$t \longrightarrow st$

On the basis of our example CSP (Table 1), we now introduce the concept of a feature model configuration task (see Definition 1) and a corresponding feature model configuration (see Definition 2).

Definition 1. A *feature model configuration task* can be defined as a constraint satisfaction problem (V, C) where V is a set of Boolean variables (features) v_i (domain (v_i)= {true, false}) and $C = REQ \cup KB$ is a set of constraints where $KB = \{c_0..c_n\}$ is a set of domain constraints and $REQ = \{c_{n+1}..c_k\}$ is a set of user requirements.

In our working example, $KB = \{c_0..c_8\}$ (see Table 1) and $REQ = \{c_9 : t = true\}$, i.e., the user has specified *ABtesting* as a feature (t) to be included in the final feature model configuration.

Definition 2. A *feature model configuration* $CONF = \{v_1 = a(v_1), v_2 = a(v_2), .., v_k = a(v_k)\}$ is a set of variable assignments where $a(v_i)$ is the value assignment of variable (feature) v_i. A configuration $CONF$ is consistent if $(\bigcup \{v_i = a(v_i)\} \in CONF) \cup REQ \cup KB$ is consistent (i.e., a solution can be found). $CONF$ is complete if every variable in V has a corresponding assignment in $CONF$. Finally, $CONF$ is valid, if it is consistent and complete.

In our setting, an example configuration is $CONF$=$\{s = true, p = true, l = true, n = false, t = true, st = true, q = true, m = true, mm = false\}$. Since users are often interested in explanations of specific features or feature sets, we introduce the concept of a sub-configuration (see Definition 3).

Definition 3. A *sub-configuration* $SCONF$ is a set of variable assignments with $SCONF \subseteq CONF$.

The basic idea of a minimal causality-based explanation $EXP \subseteq \{REQ \cup KB \cup CONF\}$ is the following. Given a sub-configuration

$SCONF \subseteq CONF$, an explanation for the features in $SCONF$ is $EXP \subseteq REQ \cup KB \cup CONF$ such that $EXP \cup \overline{SCONF}$ is inconsistent. In this context, $\overline{SCONF}$ represents the negation of subconfiguration $SCONF$, i.e., if $SCONF = \{s = true, st = true\}$, $\overline{SCONF} = \neg s \vee \neg st$. With this criteria, we express the property that $EXP \subseteq REQ \cup KB \cup CONF$ is a set of constraints (including variable assignments in REQ and $CONF$) that are able to explain the feature settings in $SCONF$ (see Definition 4).

Definition 4. A causality-based explanation for $SCONF \subseteq CONF$ is a set $EXP \subseteq REQ \cup KB \cup CONF$ such that: inconsistent ($\overline{SCONF} \cup EXP$). EXP is subset-minimal if $\neg\exists EXP' : EXP' \subset EXP$ and EXP' is a causality-based explanation.

This minimality property of explanations EXP is needed to assure that explanations do not contain irrelevant elements, i.e., explanations which include constraints with no causal relationship to the feature model (sub-)configuration ($SCONF$). Table 2 provides examples of different $SCONF$ subsets and related explanations (EXP).

For example, for $SCONF = \{l = true\}$, the corresponding minimal causal explanation is $EXP = \{c_0, c_1, c_6, c_7, c_9\}$, since c_9 is the user requirement triggering the inclusion of $ABtesting(t)$, c_6 requiring the inclusion of either $license(l)$ or $nolicense(n)$, c_7 triggering the inclusion of $license(l)$, and c_0 (the constraint requiring the inclusion of the root feature) needed as logical foundation for the other included features. In this example, explanation constraints stem solely from REQ and KB.

For $SCONF = \{m = true\}$, c_4 requires the inclusion of $QA(q)$, c_5 requires the inclusion of $multiplechoice(m)$ or $multimedia(mm)$ (or both), and $mm = false$ is part of EXP due to the fact that it is part of the configuration but cannot be explained by user requirements in REQ or constraints in KB (we assume that $\{mm = false\} \subset CONF$). In the third entry of Table 2, $|SCONF| = 2$ – the related explanation for $SCONF = \{l = true, st = true\}$ is similar to the explanation for $SCONF = \{l = true\}$.

Table 2: Example minimal causality-based explanations EXP.

$SCONF \subseteq CONF$	EXP
{l=true}	$\{c_0, c_1, c_6, c_7, c_9\}$
{m=true}	$\{c_0, c_4, c_5, mm = false\}$
{l=true,st=true}	$\{c_0, c_1, c_6, c_7, c_8, c_9\}$

4 CXPlain Algorithm

The idea of the CXPlain algorithm is the following: we want to determine a minimal explanation EXP which represents a minimal set of constraints that are the reasons for the variable value settings in $SCONF$. In other words: if EXP entails the constraints directly responsible for the variable settings in $SCONF$, then $EXP \cup \overline{SCONF}$ must be inconsistent, i.e., EXP represents a conflict with regard to the variable value settings in $\overline{SCONF}$. If this is not the case, i.e., $EXP \cup \overline{SCONF}$ is consistent, there is at least one variable setting supported by $EXP \cup \overline{SCONF}$ which is not contained in $SCONF$. If this is the case, EXP is not regarded as a minimal explanation. Since an explanation EXP can be regarded as a minimal conflict, Algorithm 1 and the corresponding recursive function QXP focus on determining a subset-minimal set of elements in $REQ \cup KB \cup CONF$ which are inconsistent with the constraints in $\overline{SCONF}$.

CXPlain (Algorithm 1) determines minimal causal explanations that indicate (explain) which constraints from $REQ \cup KB \cup CONF$ are responsible for the variable assignments in $CONF$. The idea of CXPlain is the following: (1) an explanation can only be found if $REQ \cup KB \cup CONF$ is consistent, i.e., the feature model configuration is consistent with the defined requirements (REQ) and the domain knowledge in KB. (2) if we assume, for example, that $C = \{c_1..c_8\}$ is a constraint set and $C = \{c_1..c_4\}$ is already inconsistent (i.e., contains at least one conflict), we can concentrate our search for an explanation EXP to $C = \{c_1..c_4\}$ and can ignore $C = \{c_5..c_8\}$.

```
Algorithm 1: CXPlain($REQ, KB, CONF, SCONF$) : $EXP$
if Consistent($CONF \cup KB \cup REQ$) then
    return (QXP($\emptyset, REQ \cup KB \cup CONF, \overline{SCONF}$))
else
    print 'no explanation possible'
    return ($\emptyset$)
end if
```

```
Algorithm 2: QXP($\Delta, C = \{c_1..c_n\}, B$) : $EXP$
if $\delta \neq \emptyset$ and InConsistent($B$) then
    return ($\emptyset$)
end if
if $|C| = 1$ then
    return $C$
end if
$k = \lfloor \frac{n}{2} \rfloor$
$C_1 \leftarrow c_1..c_k; C_2 \leftarrow c_{k+1}..c_n$
$\epsilon_2 \leftarrow$ QXP($C_2, C_1, B \cup C_2$)
$\epsilon_1 \leftarrow$ QXP($\epsilon_2, C_2, B \cup \epsilon_2$)
return ($\epsilon_1 \cup \epsilon_2$)
```

This basic principle of divide-and-conquer helps to reduce large sets of constraints in a logarithmic fashion since each consistency check (i.e, InConsistent(B)) can lead to a situation where one half of a constraint set does not have to be analyzed further. This is the basic idea of function QXP (Algorithm 2). In such a situation, QXP is able to identify a conflict in $REQ \cup KB \cup CONF$ with regard to the variable value settings in $\overline{SCONF}$. In the function QXP, the parameter δ help to avoid redundant consistency checks, i.e., the same check (with an identical set of constraints) should only be performed once. The check of $|C| = 1$ represents an invariant as follows: if B is consistent and $|C| = 1$, this remaining element in C can be regarded as an element part of the conflict.

QXP (Algorithm 2) follows the idea of QuickXPlain [16] which determines minimal conflict sets, i.e., sets of constraints that induce an inconsistency. Conflict sets can be used for conflict resolution, i.e., determining constraints that are responsible for conflicts in user requirements, configurations, or even knowledge bases derived from feature models [8, 20, 25, 29]. In contrast to this focus on *conflict resolution*, the major focus of our work is to show how QuickXPlain-style algorithms can be applied for the *explanation of configurations* or parts thereof.

Note that QXP follows a strict preference ordering in the sense that explanations using elements of *REQ* are preferred over explanations on the basis of elements of *KB*. The lowest priority is given to explanations based on elements (variable assignments) in *CONF*. This is due to the fact that feature inclusions/exclusions in *CONF* that are not explainable by user requirements (*REQ*) or domain constraints (*KB*) often have the lowest relevance for the user (they are related to pre-defined solver (configurator) search heuristics).

5 Performance Analysis of CXPlain

To show the applicability of CXPlain in *interactive settings*, we conducted experiments using five real-world feature models that vary in size and complexity [19]. These models stem from the S.P.L.O.T. repository [22] and the Diverso Lab's benchmark [15] (see Table 3).

We have evaluated the performance of CXPlain when identifying explanations for $|SCONF| \in \{1, 2, 4, 8\}$ settings (see Table 4). This evaluation focus is based on the assumption that users are interested in understanding the inclusion/exclusion of specific features – an explanation for all features of a configuration will in many cases include all user requirements and a majority of feature model constraints. To ensure accurate time measurements, we evaluated CXPlain with three random configurations (*CONF*) for each setting in Table 4. Three configurations for each feature model were generated by randomly selecting approximately 50% of the features from the model and assigning random values to them. Then, a solver was used to generate a valid configuration. The random steps help to ensure the diversity of generated configurations. If a generated configuration was inconsistent with the feature model constraints, a new generation process has been triggered.

From the generated configurations, we identified different *SCONF* subsets. For $|SCONF| = 1$, we measured the runtime of CXPlain for each individual feature assignment present in the generated configurations. Consequently, the number of different *SCONF* explanations corresponds directly to the size of *CONF*. For $|SCONF| = 2$, we assessed the runtime of CXPlain across all pairwise assignments within the generated configurations. For $|SCONF| = 4$ and 8, we selected and evaluated a maximum of 10,000 random t-wise assignments for each configuration. All experiments were performed on an Apple M1 Pro (8 cores) with 16GB of RAM, utilizing the Choco Solver[1] for consistency checks.

The findings regarding the performance of CXPlain are presented in *Table 4*. The results indicate that the runtime of CXPlain increases with both, the size of the feature model and $|SCONF|$ – assuming that users are interested in explanations of specific features (or smaller subsets), no related bottlenecks in terms of runtime performance can be expected. Overall, the runtime performance (see Table 4) clearly shows the applicability of the CXPlain algorithm in *interactive configuration settings* (e.g., the maximum runtime needed for determining a causal explanation *EXP* in the case of the $Win8$ feature model is around 86 milliseconds).

The complexity of QXP in terms of the number *nc* of needed consistency checks, i.e., InConsistent(B), follows the complexity of QuickXPlain [16]. In the best case, *nc* is $log_2(\frac{|C|}{m}) + 2m$, the worst case complexity is $2m \times log_2(\frac{|C|}{m}) + 2m$ where is m is the size of the minimal causal explanation and $|C| = |REQ \cup KB \cup CONF|$.

Table 3: Feature models used for CXPlain evaluation (IDE=IDE product line, Arc=Arcade Game PL, FQA=Feature model for Functional Quality Attributes, Ubu=Accessibility options provided by Ubuntu operating systems, Win8=Accessibility options provided by Windows 8 operating systems) [15, 22].

feature model	IDE	Arc	FQA	Ubu	Win8
#features	14	61	178	263	451
#hierarchical constraints	11	32	92	177	267
#cross-tree constraints	2	34	9	84	138

Table 4: Avg. runtime (*msec*) of CXPlain measured with different *feature model sizes* and *sub-configuration sizes*.

$\|SCONF\|$	IDE	Arc	FQA	Ubu	Win8
1	0.045	0.438	1.951	4.918	9.964
2	0.079	0.672	2.630	5.667	12.540
4	0.149	1.113	4.088	9.133	19.384
8	0.324	2.290	11.164	32.948	86.345

6 Threats to Validity

Potential threats to validity related to the work presented in this paper are the following. First, the explanations determined by CXPlain are represented as a set of constraints responsible for a selected set of variables. These constraints represent causal explanations reflecting the reason of a specific feature model configuration (could also be a partial one). Up to now, we did not analyze the best way to present such explanations to users – this could be done, for example, on the basis of large language models proposing a specific textual explanation for the current configuration setting. Second, in this paper we propose a logical foundation of a causality-based minimal explanation, however, more detailed evaluations are needed to better understand which are the preferred explanations from a users point of view (e.g., explanations of single variable settings vs. explanations of configuration subsets). Third, our evaluation is based on a limited set of real-world feature models. Further evaluations have to be performed to develop an improved understanding of the properties of explanations (e.g., in terms of EXP-included constraints depending on the set size of $SCONF$).

7 Conclusions and Future Work

We have introduced the CXPlain algorithm that supports the determination of minimal causal explanations of feature model configurations. Such explanations are subset-minimal sets of constraints that represent the reasons for the given specific variable value assignments of a feature model configuration (or partial configuration). The performance of CXPlain has been evaluated on the basis of a set of real-world feature models. The results of our performance analysis clearly indicate the applicability of CXPlain in interactive feature model configuration settings. Our related future work will include an empirical analysis of alternative ways to present an explanation to a user and further performance analyses of CXPlain with real-world feature models.

[1] choco-solver.org

Acknowledgments

The work presented in this paper has been conducted within the scope of the OpenSpace project funded by the Austrian research promotion agency (FO999891127).

References

[1] M. Acher, P. Temple, J-M. Jézéquel, J. Galindo, J. Martinez, and T. Tiadi. 2018. VaryLaTeX: Learning Paper Variants That Meet Constraints. In *12th Intl. Workshop on Variability Modelling of Software-Intensive Systems*. Madrid, Spain, 83–88.

[2] S. Apel and C. Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (2009), 49–84.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortes. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35 (2010), 615–636. Issue 6.

[4] D. Benavides, P. Trinidad, and A. Cortés. 2005. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China, July 14-16, 2005*, W. Chu, N. Juzgado, and W. Wong (Eds.). 677–682.

[5] S. Dev Gupta, B. Genc, and B. O'Sullivan. 2021. Explanation in Constraint Satisfaction: A Survey. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Zhi-Hua Zhou (Ed.). International Joint Conferences on Artificial Intelligence Organization, 4400–4407. doi:10.24963/ijcai.2021/601 Survey Track.

[6] T. Walsh F. Rossi, P. van Beek. 2006. *Handbook of Constraint Programming*. Elsevier.

[7] A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, and H. Schreiner. 2016. Twenty-Five Years of Successful Application of Constraint Technologies at Siemens. *AI Mag.* 37, 4 (Dec. 2016), 67–80. doi:10.1609/aimag.v37i4.2688

[8] A. Felfernig, D. Benavides, J. Galindo, and F. Reinfrank. 2013. Towards Anomaly Explanation in Feature Models. In *ConfWS-2013: 15th International Configuration Workshop (2013)*, Vol. 1128. 117–124.

[9] A. Felfernig, A. Falkner, and D. Benavides. 2024. *Feature Models: AI-Driven Design, Analysis and Applications*. Springer. doi:10.1007/978-3-031-61874-1

[10] A. Felfernig, M. Wundara, T.N.T. Tran, S. Polat-Erdeniz, S. Lubos, M. El Mansi, D. Garber, and V.M. Le. 2023. Recommender systems for sustainability: overview and research issues. *Frontiers in Big Data* 6 (2023), 16 pages. doi:10.3389/fdata.2023.1284511

[11] G. Friedrich. 2004. Elimination of spurious explanations. In *16th European Conference on Artificial Intelligence* (Valencia, Spain) *(ECAI'04)*. IOS Press, NLD, 813–817.

[12] J. Galindo, J. Horcas, A. Felfernig, D. Fernandez-Amoros, and D. Benavides. 2023. FLAMA: A collaborative effort to build a new framework for the automated analysis of feature models. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B* (Tokyo, Japan) *(SPLC '23)*. ACM, New York, NY, USA, 16–19. doi:10.1145/3579028.3609008

[13] C. Gomes, H. Kautz, A. Sabharwal, and B. Selman. 2008. Satisfiability Solvers. *Handbook of Knowledge Representation* 3 (2008), 89–134.

[14] J. Gu, P. Purdom, J. Franco, and B. Wah. 1996. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 19–152.

[15] R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory. 2022. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering* 27, 2 (2022), 44.

[16] U. Junker. 2004. QuickXPlain: preferred explanations and relaxations for over-constrained problems. In *19th National Conference on Artifical Intelligence (AAAI 2004)*. AAAI, 167–172.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. 1990. Feature-oriented Domain Analysis (FODA) – Feasibility Study. *TechnicalReport CMU – SEI-90-TR-21* (1990).

[18] D. Kramer, C. Sauer, and T. Roth-Berghofer. 2013. Towards Explanation Generation using Feature Models in Software Product Lines. In *9th Workshop on Knowledge Engineering and Software Engineering* (Koblenz, Germany). CEUR, 13–23. https://ceur-ws.org/Vol-1070/

[19] V.M. Le. 2025. *AIG-ist-tugraz/CXPlainEvaluation: v1.0-vamos 2025*. doi:10.5281/zenodo.14678351

[20] V.M. Le, C. Silva, A. Felfernig, D. Benavides, J. Galindo, and T.N.T. Tran. 2023. FASTDIAGP: an algorithm for parallelized direct diagnosis *(AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 723, 8 pages. doi:10.1609/aaai.v37i5.25792

[21] M. Hentze and T. Pett and T. Thüm and I. Schaefer. 2021. Hyper Explanations for Feature-Model Defect Analysis. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems* (Krems, Austria) *(VaMoS'21)*. Association for Computing Machinery, New York, NY, USA, Article 14, 9 pages. doi:10.1145/3442391.3442406

[22] M. Mendonca, M. Branco, and D. Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 761–762. doi:10.1145/1639950.1640002

[23] M. Mendonça, A. Wasowski, and K. Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings (ACM International Conference Proceeding Series, Vol. 446)*, D. Muthig and J. McGregor (Eds.). ACM, 231–240.

[24] V. Myllärniemi, J. Tiihonen, M. Raatikainen, and A. Felfernig. 2014. Using Answer Set Programming for Feature Model Representation and Configuration. In *16th International Workshop on Configuration*. CEUR, Novi Sad, Serbia, 1–8.

[25] R. Reiter. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32, 1 (1987), 57–95.

[26] N. Tintarev and J. Masthoff. 2007. A Survey of Explanations in Recommender Systems *(ICDEW '07)*. IEEE Computer Society, USA, 801–810. doi:10.1109/ICDEW.2007.4401070

[27] T.N.T. Tran, A. Felfernig, and V.M. Le. 2023. An overview of consensus models for group decision-making and group recommender systems. *User Modeling and User-Adapted Interaction* 34, 3 (Sept. 2023), 489–547. doi:10.1007/s11257-023-09380-z

[28] T.N.T. Tran, S. Polat Erdeniz, A. Felfernig, S. Lubos, M. El Mansi, and V.M. Le. 2024. Less is More: Towards Sustainability-Aware Persuasive Explanations in Recommender Systems. In *Proceedings of the 18th ACM Conference on Recommender Systems* (Bari, Italy) *(RecSys '24)*. Association for Computing Machinery, New York, NY, USA, 1108–1112. doi:10.1145/3640457.3691708

[29] J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. 2010. Automated diagnosis of feature model configurations. *Journal of Systems and Software* 83, 7 (2010), 1094–1107.

A Demo of ConfigFix: Semantic Abstraction of Kconfig, SAT-based Configuration, and DIMACS Export

Jude Gyimah
Ruhr University Bochum
Bochum, Germany
jude.gyimah@rub.de

Jan Sollmann
Ruhr University Bochum
Bochum, Germany
jan.sollmann@rub.de

Ole Schuerks
Ruhr University Bochum
Bochum, Germany
ole.schuerks@edu.ruhr-uni-bochum.de

Patrick Franz
Chalmers | University of Gothenburg
Gothenburg, Sweden
patfra71@gmail.com

Thorsten Berger
Ruhr University Bochum and
Chalmers | University of Gothenburg
Bochum, Germany
thorsten.berger@rub.de

Abstract

The Linux kernel and its huge configuration space (>15,000 features) has been a frequent study object. While the research community has developed intelligent software configuration tools, often motivated by the Linux kernel and its configuration language *Kconfig*, the kernel's own configurator xconfig lacks behind. Configuration conflicts need to be resolved manually, which often causes substantial overhead. Unfortunately, *Kconfig* is a complex and intricate language, and while transformations into propositional logic exist, they typically have shortcomings and are difficult to integrate into xconfig. We contribute research results back to the Linux community and present a demo of ConfigFix. It is a plain-C-based extension of xconfig, providing the currently most accurate abstraction of the *Kconfig* semantics into propositional logic. It provides configuration conflict resolution. Integrated into the xconfig UI, it offers configuration fixes to users trying to enable or disable kernel features restricted by dependencies. In addition, researchers benefit from the DIMACS export. Our demo presents the main capabilities of xconfig as well as its evaluation showing the accuracy of it.

Keywords

configuration, Linux Kernel, configuration conflicts, SAT solving

ACM Reference Format:
Jude Gyimah, Jan Sollmann, Ole Schuerks, Patrick Franz, and Thorsten Berger. 2025. A Demo of ConfigFix: Semantic Abstraction of Kconfig, SAT-based Configuration, and DIMACS Export. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3715340.3715445

1 Introduction

The Linux kernel's applicability in many different computing environments—ranging from small Android devices to large scale supercomputer clusters—has contributed to making it one of the world's largest software development projects [8]. Designed as a *highly configurable system* [50] it boasts 28 million lines of code [26] and over 15,000 configuration options (a.k.a. *features* [5, 7, 40]). To this end, it has a configurable build system [6], preprocessor-based variation points [7], a model of its features (*Kconfig model*) and constraints [7, 34], and an interactive configurator tool (*xconfig*) [50].

A plethora of academic studies and techniques on the kernel's variability and its configuration space exist, ranging from variability anomaly detection [2, 31, 35, 52], techniques to identify, extract, and analyze configuration constraints [29, 34, 49, 50, 55], as well as automated support for reviewing and testing patches [27, 28]. In addition to that, many studies have also been conducted on the kernel's evolution [4, 19, 41, 43] and maintenance [1, 18, 20, 56], analyzing its feature model [7, 47], the modeling language (*Kconfig*), the evolution of the model [29], the co-evolution and consistency of its variation points [22, 36, 37, 41, 42, 54], and the representation of feature constraints in its codebase [34]. Many of the results have inspired software configurator tools [13, 15]. Various other tools [51, 58] and techniques [32, 33, 44], including the synthesis of feature models from code or feature constraints [23, 30, 48], have also drawn inspiration from the Linux kernel and its configurator, by borrowing from or directly extending its capabilities.

The kernel's main configurator xconfig, however, lacks behind the state of the art. Kernel users have faced challenges when manually creating their desired configurations, given the kernel's vast configuration space and intricate feature constraints. Furthermore, there is limited support for choice propagation and a lack of intelligent configurator support for resolving configuration conflicts. These challenges are common, because enabling a feature often requires transitively changing many others. Consequently, achieving a desired configuration can in turn be laborious and error-prone. A survey [17] revealed that kernel users commonly struggle with conflict resolution, with 20 % of the respondents needing at least "a few dozen minutes" to do so. Despite this, insofar as we know, no configuration technique stemming from academia, has been specifically developed for the Linux kernel such that it can be officially integrated into the Linux kernel mainline.

In 2015, with the Kconfig-sat initiative [24] kernel developers recognized the need for such a solution. They got in touch with researchers working on kernel configuration studies, to integrate

VaMoS 2025, Rennes, France

ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715445

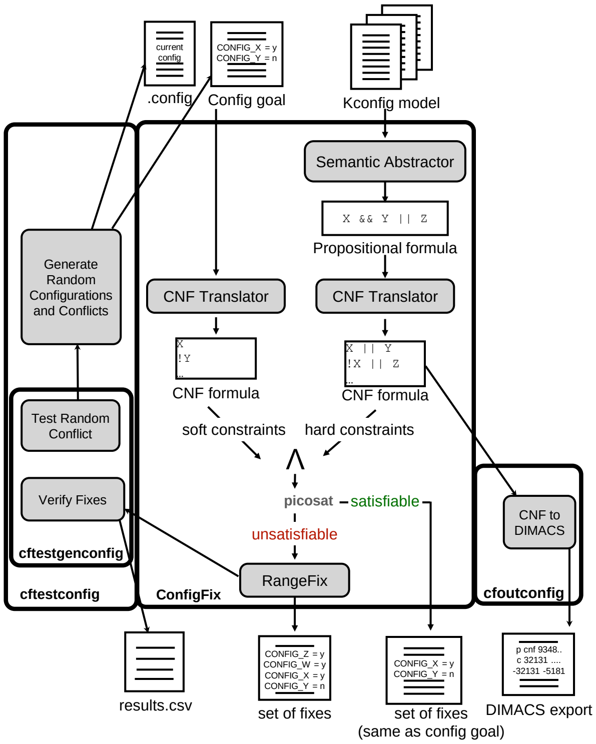


Fig. 1: ConfigFix components

such techniques back into the kernel configurator. Indeed, implementing a sound translation of the kernel's variability model to propositional logic, given the expressiveness and intricate nature of the *Kconfig* language semantics (explained shortly), has long been an open problem in the community. Multiple translation attempts have been proposed to remedy this [10, 21, 46, 53], but none of them have been without limitations [9].

We present a demo of ConfigFix [11, 12], which offers intelligent configuration support integrated into the Linux kernel configurator *xconfig*. It is completely implemented in C, extends *xconfig*'s graphical user interface, and comes with a testing framework. Since researchers can draw value from the DIMACS export, we also discuss the Kconfig's semantics and the implementation details of our semantic abstraction. In contrast to our previous work [11], this paper is a demo, showcasing enhancements in the user interface, test infrastructure, code quality, and presents the DIMACS export functionality. The current version also integrates feedback from the kernel community. Our efforts have led to a stable version of ConfigFix and its test infrastructure [12], that is currently in the process of being integrated in the kernel's source tree (patches submitted and discussed).

2 ConfigFix Design

Figure 1 shows the main components of ConfigFix: translation of the *Kconfig* model into a propositional formula, translation into conjunctive normal form (CNF), conflict resolution algorithm, test infrastructure, and DIMACS export. When invoked, ConfigFix takes the current configuration and the user's configuration goal as input. It then creates a single formula that is a conjunction of all constraints and then queries the SAT solver for satisfiability. In case of a configuration conflict (i.e., the user's goal is not applicable), it triggers our C-based, RangeFix-inspired implementation (discussed shortly) to calculate fixes [11, 59].

Linux Kernel Configuration. The kernel has three different configurators: *make xconfig*, *make menuconfig*, and *make config*. The *xconfig* provides a graphical user interface, while the others are tailored towards shell users. Via them, correct configurations of kernel features (which come with many different characteristics defined in the *Kconfig* model) that adhere to constraints (e.g., types and dependencies) can be created and modified. The underlying DSL [3] *Kconfig* provides feature-modeling concepts [7, 47]. It declares features (called symbols there) of different types (i.e., bool, tristate, string, hex, and int) [45, 46]. Tristate features are often used to control the binding modes of features via three states: "Yes" (compile feature into kernel), "No" (do not compile feature), or "Module" (compile feature as dynamically loadable kernel feature), where constraint semantics follow Kleene's rules for three-state logic [25]. A persistent challenge has been obtaining a sound logical representation (mainly due to its complexity) of the main semantics of *Kconfig*, as a prerequisite to develop analysis and configuration techniques [11]. After we were the first to formalize its semantics [45, 46] (reverse-engineered from *xconfig*'s behavior), many others followed up [9, 24, 38, 45, 61] with their own translations. These works demonstrated that the semantics of *Kconfig* can indeed be abstracted into propositional logic, to be used by SAT solvers.

Translation. Given *Kconfig*'s expressiveness beyond propositional logic, an abstraction is required. Specifically, ConfigFix translates each feature (called symbol in *Kconfig*) into a set of variables that represent the possibility of the feature assuming a specific value. For example, any arbitrary Boolean feature (i.e., symbol) will be translated to a single variable S which is either true or false. Tristate features may only assume the values "Yes," "Module," or "No." For such features (i.e., S_TRISTATE), the constraint set variables C and C_m would be created, where C is true if and only if C is "Yes." and C_m is true if and only if C is "Module". In theory, in case of an integer feature (i.e., C_INT), with default value of 5 and a constraint $2 \leq C \leq 8$, it would be translated into six variables in the propositional formula (which are indeed named as described): C=0, C=1, C=2, C=5, C=8 and C=n. C=0, C=1 and C=n are created for every feature by default with C=n handling the case that C is hidden and has no value (i.e., S_UNKNOWN). C=2, C=5 and C=8 are "known values," which could be values that are either default for the feature or part of a range constraint. The propositional formula is then constructed based on these variables and subsequently translated into CNF using Tseitin transformation [57]. It is noteworthy that this aspect of implementing the ConfigFix translation is the most complicated, given that most features are mainly tristate. Thus, the sheer scale of everything (i.e., features, values, types and especially the CNF conversion), was challenging.

Fix Generation. The ConfigFix GUI is responsible for displaying calculated fixes for configuration conflicts received from users as configuration goals that are not applicable in the current configurator. To calculate and generate fixes from conflicts, a suitable conflict-resolution is required. Various conflict-resolution algorithms exist [16], but many were not applicable to the scope of ConfigFix.

They either produce just a single fix, suggest a long list of fixes or only offer limited support for non-Boolean constraints [39]. However, our conflict-resolution algorithm of choice, a rangefix-inspired conflict resolver, produces fixes that offer a range of values for features, supports non-Boolean features and constraints, adheres to correctness, contains a maximum range in the event of overlapping ranges and requires minimal feature set changes [11]. In our implementation, it takes a CNF model where features are represented by variables and generates minimal sets of features (refered to as diagnoses) that must be changed. Next, it calculates new values for each variable in a diagnosis. All unchanged variables are then replaced by their current values and any violated constraints are minimized via heuristic rules defined and split into minimal clauses to generate the fixes. In summary, with those *RangeFix* capabilities, when there are infinitely many possible solutions available, the user is only presented with the minimal set of these [60].

3 ConfigFix Demo

We now demonstrate ConfigFix's conflict resolution workflows and DIMACS export functionality. We show the *xconfig* UI, the specification of the configuration goal (the assignment of values for a set of features to be configured), the calculation of adequate *fixes*, and their application.

Workflows. Figure 2 presents the complete workflow of ConfigFix. Upon launch, *Xconfig* provides a view that presents a dependency-based hierarchy of nested menus containing features and allows feature value changes when the requirements for the target values are met (as shown in ①). The *Kconfig* language allows specifying conditions for when a feature should be visible in the view. Users can add any tristate feature to the set of features that should be changed by selecting it in this view and clicking the “Add symbol” button (as shown in ②). The set of features that a user wants to change, but that cannot be changed is also called *conflict*. The features that are currently added to the conflict are displayed in the menu on the bottom left. The desired value of a feature in the conflict can be set by selecting it in the menu on the bottom left and using one of the buttons “Y”, “M”, or “N” to set it to “Yes”, “Module”, or “No,” respectively. Alternatively, users can cycle through the three values by clicking on the corresponding cell in the column “Wanted Value.” The difference between “Yes” and “Module” is that with the former, a feature will be statically linked into the kernel, whereas with the latter it will be built as a dynamically loadable kernel module. If a feature cannot be built as a kernel module, the button “M” is grayed out. A feature (symbol) can be removed from the conflict by selecting it in the menu on the bottom left and clicking the button “Remove Symbol.”

Once the desired feature values have been set, *fixes* can be calculated by clicking “Calculate Fixes.” The fixes are then displayed in the table on the bottom right (as shown in ③). Users can switch between different fixes with a dropdown menu. The fixes associate each feature in the conflict with a new value, and ideally, there is an order of the changes the fix presents in which they can be applied such that after applying all changes, the desired changes would have been achieved. The changes can be made manually, or by clicking on “Apply Selected Solution”, the values can be applied automatically, in which case the user is informed via a dialog when the fix has been applied successfully (as shown in ④). ConfigFix supports the

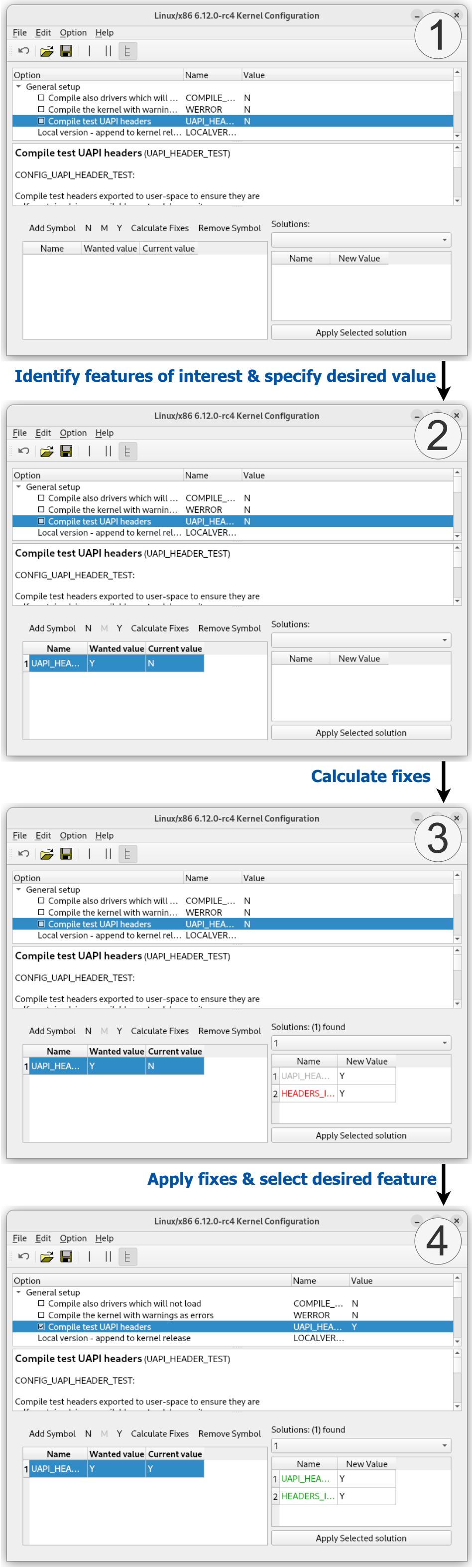

1
Linux/x86 6.12.0-rc4 Kernel Configuration
File Edit Option Help
Option Name Value
General setup
Compile also drivers which will ... COMPILE_... N
Compile the kernel with warnin... WERROR N
Compile test UAPI headers UAPI_HEA... N
Local version - append to kernel rel... LOCALVER...
Compile test UAPI headers (UAPI_HEADER_TEST)
CONFIG_UAPI_HEADER_TEST:
Compile test headers exported to user-space to ensure they are
Add Symbol N M Y Calculate Fixes Remove Symbol
Solutions:
Name Wanted value Current value
Name New Value
Apply Selected solution
Identify features of interest & specify desired value
2
1 UAPI_HEA... Y N
Calculate fixes
3
Solutions: (1) found
1 UAPI_HEA... Y
2 HEADERS_I... Y
Apply fixes & select desired feature
4
Compile also drivers which will not load COMPILE_... N
Compile the kernel with warnings as errors WERROR N
Compile test UAPI headers UAPI_HEA... Y
Local version - append to kernel release LOCALVER...
1 UAPI_HEA... Y Y

Fig. 2: User workflow using ConfigFix inside xconfig

user in applying the changes manually by coloring the names of the features in the table showing the fixes based on when the value can and needs to be set manually. A feature is red if its value can be set to the value provided by the fix and that value differs in the current configuration (as shown in ③). It is green if its value in the current configuration already matches with the value the fix proposes (as shown in ④). It is gray if the fix's value differs from the configuration's value for the feature, but further dependencies need to be fulfilled in order to set the feature to the value of the fix. Figure 2 shows the sequence of the workflow described from the identification of features of interest to be configured, right down to the application of calculated fixes.

DIMACS Export. ConfigFix allows *Kconfig* model exports in DIMACS notation, using "make cfoutconfig". The DIMACS output file offers users the *Kconfig* model in CNF form (see Listing 1). This export functionality supports further research on the kernel's feature model and variability mechanisms. For example, given a configuration with 934872 variables that can be true or false and 2764895 constraints where a user wants to enforce functionalities such as automated isolation testing, build integrity and preprocessor validation via the **UAPI_HEADER_TEST (32131)** feature, Listing 1 indicates that when 32131 is set to false, its dependant 51181 (**UAPI_HEADER_TEST_NPC**) must also be set to false. Such DIMACS exports could be potentially useful in logical inference scenarios where system based conclusions can be drawn from known configuration constraints, using algorithms that verify the satisfiability (SAT) of the underlying propositional logic statements.

Listing 1: DIMACS export snippet for UAPI_HEADER_TEST

```
p cnf 934872 2764895
c 32131 UAPI_HEADER_TEST
-32131 -51181 0
```

4 Test Infrastructure

We evaluated ConfigFix based on the number of conflicts it resolves and the quality of fixes. We created a test infrastructure that generates random configuration samples, introduces conflicts, executes ConfigFix, and then validates the fixes. The *cftestconfig* module (invoked by the command "make cftestconfig") is split into two submodules i.e., the *ConflictFrameworkSetup* responsible for initializing system parameters and the (*ConflictFramework*), which handles the generation of configurations and conflicts.

Specifically, the test infrastructure works as follows. *Cftestconfig* can be used to generate configurations and conflicts in three different modes. ***mode="single"*** gives the user the option to generate several conflicts from an existing configuration. When these conflicts are resolved, the data is saved in a results file. In this mode, "make cftestgenconfig" can be called as many times as required. ***mode="multi"*** provides the option to generate multiple random configurations while simultaneously generating for every configuration, a random set of conflicts for a given architecture. The conflicts are solved and the data is subsequently saved in the results file. In its order of execution, "make randconfig" is called first, followed by "make cftestgenconfig". ***mode="multi_arch"*** gives users the option to generate multiple random configurations and then generate for each, a random set of conflicts for multiple architectures.

First, *it samples the configuration space*. Samples should provide enough coverage of the features contained therein [14]. This requires all important features (such as hardware architecture) in the configuration space to be considered [14]. Our test infrastructure uses `randconfig` (an existing tool in the build system) to generate random configurations. Second, *it introduces conflicts*. For each of the generated configurations, it generates random conflicts containing a specific number of changes requested (i.e., the conflict size, which is configurable).

It then generate a base configuration using `randconfig` with a probability of 100 % of a feature being set with all conflicts chosen as subsets of the base configuration. Third, *it executes the fix generation* for each conflict. The generated configuration sample and conflict become inputs to the test algorithm, which will either produce a single or several configuration fixes, or provide feedback that the conflict cannot be resolved. Conflicts are generated by repeatedly choosing a random Boolean or tristate feature with at least one value to which it cannot be set to, due to unmet dependencies. The conflict then requires this option to be set to one of its blocked values (chosen at random). ConfigFix generates a maximum of three fixes that satisfy the set of violated constraints and can accommodate overlapping ranges using defined heuristics to minimise the number of features value changes. Fourth, *it validates the fixes*. A fix may include some additional features subject to change, in order to set them to their new, desired state. Such features typically bear configuration conflict properties, which may be too hard to reconfigure manually. As such, for every fix returned by the algorithm, it is applied to the sample configuration and verified to ensure that the configuration goal is satisfied, the resultant configuration after applying the fix is valid (via the *xconfig* configurator), as well as no unnecessary changes have been made to the product configuration.

For the actual evaluation, we generated configurations (i.e., in the multi_arch mode) for three different architectures (`x86_64`, `arm64`, and `openrisc`), and for nine different probabilities for a feature to be selected, ranging from 10 % to 90 %. In total, the infrastructure evaluated 27 different configurations. We set it to generate conflict sizes between 1–10 features. Table 1 shows the results. It generated 1350 conflicts. For each of the 27 configurations, and each of the 10 conflict sizes, 5 conflicts were generated. ConfigFix successfully resolved 1,150 conflicts (85.2%) and produced 2,645 fixes in total. Almost all fixes (99.9%) resolved the conflicts, with only 2 fixes (0.1%) failing to do so. Of the fixes, 1,116 (42.2%) were fully applicable (i.e., all the specified values in the fix can be applied) and resolved the conflict, while 1,527 (57.7%) resolved the conflict despite not being fully applicable (i.e., some of the values specified in the fix cannot be applied). Further evaluation details are provided in the ConfigFix repository [12].

Note, a fix could still not resolve a conflict, since our translations abstract the *Kconfig* model into a propositional formula, which may not be able to capture all the constraints of the original model. Consequently, a fix can be fully inapplicable, but still resolve the conflict, since the fix may change the value of a variable that is not directly involved in the conflict, but is a dependency of a variable that is involved in the conflict.

[1] One for each architecture and probability.

[2] For each configuration sample, five conflicts of each size.

Table 1: Evaluation results

metric	value
number of sampled configurations	27[1]
conflict sizes	1–10
total generated conflicts for evaluation	1,350[2] (100.0 %)
conflicts with >= 1 fix produced by ConfigFix	1,150 (85.2 %)
number of resolved conflicts	1,150 (85.2 %)
total number of fixes produced by ConfigFix	2,645 (100.0 %)
fixes that resolve the conflict	**2,643 (99.9 %)**
fully applicable and resolves conflict	1,116 (42.2 %)
not fully applicable, but resolves conflict	1,527 (57.7 %)
does not resolve conflict	2 (0.1 %)

5 Conclusion

We presented a demo of ConfigFix. It helps to configure the Linux kernel by offering automated conflict resolution support. While details are in our preceding conference paper [11], herein we focus on demonstrating capabilities and showing improvements since then.

The results from our evaluation indicates the level of accuracy with which ConfigFix is capable of resolving the conflicts it encounters. Thus, reinforcing the notion that at least one fix is produced in almost 85% of the examples generated. And in those examples, fixes generated resolved the conflict correctly in about 100% of the cases presented. However, despite this impressive level of correctness, ConfigFix has some minor limitations.

For starters, ConfigFix does not explicitly handle string and integer constraints due to how relatively rare they are compared to boolean and tristate constraints. Often, the Linux kernel's features are constrained such that they can be enabled, disabled or set as a loadable kernel. Another slight restriction comes from our choice to use a SAT solver instead of an SMT solver, requiring translation from *Kconfig* to propositional formulas. This means some assignments with integer or string variables are missed by ConfigFix. An SMT solver could handle more complex value sets, like ranges, while ConfigFix only suggests specific integer values. Furthermore, RangeFix was designed for SMT, however based on mailing list discussions with SAT experts, performance was paramount. Additionally, the translation is not perfectly accurate, even excluding strings and integers. This results in smaller constraint sets and faster runtimes. While this could lead to invalid fixes, irrelevant changes, and missed existing fixes, our validation of ConfigFix shows that this is not a problem at all.

Finally, based on mailing-list interactions, a future direction could be to provide a command-line configuration completion tool where users give a partial configuration, and ConfigFix completes it. We hope the C-based integration developed by discussion with Linux developers contributes research results back to the Linux kernel community, supporting users to configure their desired kernel.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *ASE*.

[2] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. 2019. *Learning From Thousands of Build Failures of Linux Kernel Configurations*. Technical Report. Inria ; IRISA. https://inria.hal.science/hal-02147012

[3] Wąsowski Andrzej and Berger Thorsten. 2023. *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer International Publishing.

[4] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. 2002. Analyzing cloning evolution in the linux kernel. *Information and Software Technology* 44, 13 (2002), 755–765.

[5] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Software Product Line Conference (SPLC)*. 16–25.

[6] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines. In *SPLC*.

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Softw. Eng.* 39, 12 (Dec. 2013), 1611–1640.

[8] Swapnil Bhartiya. 2016. Linux is the largest software development project on the planet: Greg Kroah-Hartman. urlhttps://www.cio.com/article/3069529/linux-is-the-largest-software-development-project-on-the-planet-greg-kroah-hartman.html. Accessed: 2020-01-06.

[9] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig semantics and its analysis tools. In *GPCE*.

[10] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A Kconfig Translation to Logic with One-Way Validation System. In *SPLC*.

[11] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[12] Patrick Franz, Ibrahim Fayaz, Thorsten Berger, Sarah Nadi, Evgeny Groshev, Lukas Günther, Dorina Sfirnaciuc, Jude Gyimah, Jan Sollman, and Ole Schürks. 2020. ConfigFix. https://bitbucket.org/easelab/configfix.

[13] José A Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101 (2019), 387–433.

[14] Evgeny Groshev. 2020. *A testing technique for conflict-resolution facilitiesin software configurators*. Master's thesis. University Of Gothenburg and Chalmers University Of Technology.

[15] Arnaud Hubaux, Dietmar Jannach, Conrad Drescher, Leonardo Murta, Tomi Männistö, Krzysztof Czarnecki, Patrick Heymans, Tien Nguyen, and Markus Zanker. 2012. Unifying software and product configuration: a research roadmap. In *CONFWS*.

[16] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker. 2012. Unifying Software and Product Configuration: A Research Roadmap. In *ConfWS*.

[17] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *VaMoS*.

[18] Ayelet Israeli and Dror G Feitelson. 2009. *Characterizing software maintenance categories using the Linux kernel*. Technical Report 2009–10. School of Computer Science and Engineering, The Hebrew University of Jerusalem.

[19] Ayelet Israeli and Dror G Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485–501.

[20] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast? case study on the linux kernel. In *MSR*.

[21] Christian Kästner. 2014. KConfig Reader. https://github.com/ckaestne/kconfigreader.

[22] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*.

[23] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*. 611–614. doi:10.1109/ICSE.2009.5070568

[24] Kernelnewbies. 2017. Linux kconfig SAT integration. https://kernelnewbies.org/KernelProjects/kconfig-sat. Accessed: 2019-12-05.

[25] S. C. Kleene. 1938. On notation for ordinal numbers. *Journal of Symbolic Logic* 3, 4 (1938), 150–155. doi:10.2307/2267778

[26] Michael Larabel. 2020. The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019. https://www.phoronix.com/news/Linux-Git-Stats-EOY2019. Accessed: 2020-01-06.

[27] Julia Lawall and Gilles Muller. 2017. JMake: Dependable Compilation for Kernel Janitors. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 357–366. doi:10.1109/DSN.2017.62

[28] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. https://www.usenix.org/conference/atc18/presentation/lawall

[29] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In *International Conference on Software Product Lines (SPLC)*. 136–150.

[30] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[31] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. 2016. A quantitative analysis of variability warnings in linux. In *Proceedings of the 10th*

International Workshop on Variability Modelling of Software-Intensive Systems. 3–8.

[32] Johann Mortara and Philippe Collet. 2021. Capturing the diversity of analyses on the Linux kernel variability. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) *(SPLC '21)*. Association for Computing Machinery, New York, NY, USA, 160–171. doi:10.1145/3461001.3471151

[33] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *SPLC*.

[34] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. Softw. Eng.* 41, 8 (2015), 820–841.

[35] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux variability anomalies: What causes them and how do they get fixed?. In *MSR*.

[36] Sarah Nadi and Richard C. Holt. 2011. Make it or break it: Mining anomalies from Linux Kbuild. In *WCRE*. 315–324.

[37] Sarah Nadi and Richard C. Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *CSMR*.

[38] Vegard Nossum. 2019. satconfig. https://github.com/vegard/linux-2.6/tree/v4.7+kconfig-sat. Accessed: 2019-12-05.

[39] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of Non-Boolean Constraints in a Variability Model of an Embedded Operating System. In *Feature-Oriented Software Development (FOSD)*.

[40] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *International Conference on Modularity (MODULARITY)*.

[41] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2021. A Study of Feature Scattering in the Linux Kernel. *IEEE Trans. Softw. Eng.* 47 (2021), 146–164. Issue 1.

[42] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *Empirical Softw. Engg.* 21, 4 (Aug. 2016), 1744–1793.

[43] Tu Qiang and Godfrey Michael W. 2000. Evolution in open source software: A case study. In *ICSM*.

[44] Muhammad Ejaz Sandhu. 2021. Comparison of Fault Simulation Over Custom Kernel Module Using Various Techniques. *Lahore Garrison University Research Journal of Computer Science and Information Technology* 5, 3 (2021), 73–83. doi:10.54692/lgurjcsit.2021.0503220

[45] Steven She. 2013. LVAT. https://github.com/shshe/linux-variability-analysis-tools.

[46] Steven She and Thorsten Berger. 2010. Formal Semantics of the Kconfig Language. Technical Note. https://arxiv.org/abs/2209.04916.

[47] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of the Linux Kernel. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010)*.

[48] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *ICSE*.

[49] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is The Linux Kernel a Software Product Line?. In *SPLC-OSSPL*.

[50] Julio Sincero and Wolfgang Schröder-Preikschat. 2008. The Linux Kernel Configurator as a Feature Modeling Tool. In *ASPL*.

[51] Eduard Staniloiu, Razvan Nitu, Cristian Becerescu, and Razvan Rughinis. 2021. Automatic Integration of D Code With the Linux Kernel. In *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. 1–6. doi:10.1109/RoEduNet54112.2021.9638307

[52] Stefan Strueder, Mukelabai Mukelabai, Daniel Strueber, and Thorsten Berger. 2020. Feature-Oriented Defect Prediction. In *24th ACM International Systems and Software Product Line Conference (SPLC)*.

[53] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In *USENIX ATC*.

[54] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*.

[55] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *FOSD*.

[56] Y. Tian, J. Lawall, and D. Lo. 2012. Identifying Linux bug fixing patches. In *ICSE*.

[57] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. doi:10.1007/978-3-642-81955-1_28

[58] Ying-Jie Wang, Liang-Ze Yin, and Wei Dong. 2021. AMCheX: Accurate Analysis of Missing-Check Bugs for Linux Kernel. *Journal of Computer Science and Technology* 36, 6 (Dec. 2021), 1325–1341.

[59] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating range fixes for software configuration. In *34th International Conference on Software Engineering (ICSE)*.

[60] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki. 2015. Range Fixes: Interactive Error Resolution for Software Configuration. *IEEE Trans. Softw. Eng.* 41, 6 (June 2015), 603–619.

[61] Christoph Zengler and Wolfgang Küchlin. 2010. Encoding the Linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*.

Analyzing SCons-Based Industrial Software Product Lines

Pascal Becker
ABB AG Corporate Research Center
Germany
Mannheim, Germany
pascal.becker@de.abb.com

Sten Gruener
ABB AG Corporate Research Center
Germany
Mannheim, Germany
sten.gruener@de.abb.com

Lukas Linsbauer
ABB AG Corporate Research Center
Germany
Mannheim, Germany
lukas.linsbauer@de.abb.com

Abstract

Industrial real-time embedded software is usually developed over decades in heterogeneous, multi-disciplinary development teams, inducing major maintenance effort. In addition to a maintenance strategy, increasing variability of products requires a dedicated strategy for the variability management between products and product families. Software product lines (SPLs) provide a proven methodology to address the variability management. Despite the existence of various tools for SPL engineering, none of existing tool kits was matching our requirements especially regarding hierarchical build system based on SCons. We present a variability analysis toolkit which seamlessly integrates with SCons build system, provides a simple integration with CI/CD pipeline, and an option to graphically explore the structure of build system artifacts. Our toolkit automatically calculates key performance indicators for single and multiple product variant graphs. It also provides cluster analysis to find similarities and differences between different product variants. Our initial findings show the acceptance of our Python-based toolkit by the development teams as well as the ability to automatically detect common and variant-specific feature clusters which are useful for further improvement and maintenance of the whole software product line towards an SPL-aware build system.

Keywords

Software Product Lines, Variability Mining, Visualization Techniques for Variability, Embedded Industrial Software

ACM Reference Format:
Pascal Becker, Sten Gruener, and Lukas Linsbauer. 2025. Analyzing SCons-Based Industrial Software Product Lines. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715430

1 Introduction

As a leading manufacturer of low-voltage variable frequency drives (VFDs), ABB Motion is developing several VFD product families to support the needs of different customers and markets worldwide. The product families are targeting different motor power classes as well as dedicated application areas, e.g., in cranes, heating, ventilation, and air conditioning (HVAC), or water-treatment applications.

Due to lasting trends in the digitalization (Industry 4.0 and Industrial Internet of Things) there is an ongoing need to offer and maintain a number of application-specific (or even customer-specific) product variants while ensuring a high product quality.

Obviously, the use of well-known Software Product Line Engineering (SPLE) techniques to manage the variability between different product families and VFD variants is promising especially due to a large body of knowledge including industrial embedded applications and a variety of tools supporting SPLE.

In prior work there was focus on automatic feature extraction of the feature artifacts from the source code, e.g., locating C++ files implementing end-customer-functionality from the VFD user manual [4, 9]. Subsequently, there was work on the process of a gradual, non-invasive migration towards the usage of commercial off the shelve Software Product Line (SPL) tools for the variability management [10]. As a milestone, a non-invasive continuous monitoring of the build variants was established to assure the compliance of the feature model with the ground truth of the built artifacts was established using the commercial pure::variants SPL tool.

Following the methodology of [10], the next step of the SPL introduction would be the switch to stimulative and operative use of the SPL tooling to generate and execute build files for each variant out of 150% model as build file templates. However, we discovered that an additional desktop or web-based tool didn't lead to the desired success and participation within the development teams. Therefore, an alternative approach to make the used build chain SPL-aware was selected. The analyzed SPL makes use of SCons [13] – a Python-based build tool supporting an arbitrary amount of regular Python code for customization.

In this paper we present our work on build-file-based analysis of an existing industrial SPL based on the SCons build tool chain. We continue prior work in the domain of low voltage VFDs and introduce a new Variability Architecture Analysis Toolkit (VAAT) for analysis of build chains implemented with SCons with significant amounts of custom Python code.

The remainder of this paper is structured as follows: After a short overview of related work in Section 2 we provide an overview of the introduced toolkit in Section 3 including requirements, implementation details and applications within the Continuous Integration / Continuous Delivery (CI/CD) pipeline. Section 4 reviews our first results including examples of VAAT output. Finally, a summary and an overview of future work are presented in Section 5.

2 Related Work

A common challenge during the adoption of SPLE based on an existing set of variants is to analyze their commonalities and differences.

VaMoS 2025, Rennes, France

ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715430

Fischer et al. [6, 7, 14] developed a method and a corresponding tool, called ECCO, that is able to compare the implementation artifacts of large sets of variants to produce clusters of artifacts or fragments thereof. For each cluster a presence condition, which is a Boolean formula with features as literals, is computed that determines in what variants a cluster of artifacts is contained. The clustering approach of ECCO is similar to the clustering of VAAT. However, the former assumes variants as input that are consistent (i.e., the same feature is implemented in exactly the same way in each variant) and disjoint (i.e., no references of files across variants), while VAAT is more robust in these regards but due to that only provides analysis results and does not perform automated refactoring.

Hinterreiter et al. [11] developed a tool called FORCE that integrates ECCO, which does not offer any form of variability modeling (e.g., feature modeling) to express constraints among features, with feature modeling and extends it with revisioning of features [12, 17] as well as with operations for distributed feature-oriented development [12].

Martinez et al. [15, 16] proposed and implemented an approach and tool called BUT4Reuse that covers the entire spectrum of refactoring legacy variants into a software product line. This includes in particular feature identification and feature location, which also VAAT aims to support with the metrics and clustering it provides.

Ananieva et al. [1, 2] analyzed and unified concepts and operations of many more tools, including all of the above, that deal with variability in space (i.e., variants) and time (i.e., revisions of features or variants).

Research on the Linux kernel and its variability has been conducted in various areas. In particular related to this work is the fact that the Linux kernel uses the make build system to implement some of its variability [5, 18]. VAAT aims to analyse variability implemented via the SCons build system, which is particularly challenging as build files allow the use of regular Python code and can thus become very complex.

3 VAAT Architecture

The core of the VAAT is a graph structure consisting of involved build artifacts for each product variant. The graph consists of nodes which represent SCons files, further build artifacts (like C++ headers and source files) and directories containing them. Furthermore, information about the current variant of the compiled firmware is attached as label to the particular node.

There are two major use cases for VAAT:

- Descriptive usage: based on the graph, VAAT can be used to define and calculate Key Performance Indicators (KPIs) for continuous monitoring within the CI/CD pipeline. Additionally, rules can be defined to prevent tool chain architecture erosion (similar to the non-invasive "synchronize" step described in [10]).
- Prescriptive usage: cluster-analysis based on VAAT graphs can provide suggestions to refactor existing build chain artifact (cf. Section 3.4).

Requirements towards VAAT were extracted via interviews with involved and experienced software developers and architects:

- R1: ability to analyse SCons-based build process (including custom Python code, e.g., conditional branching),
- R2: ability to be executed in a headless fashion to be included into CI/CD pipelines,
- R3: extensibility to include additional KPIs or information sources beyond C/C++ code (e.g., additional types of artifacts from in-house code-generation tools),
- R4: arguably short execution times in order to not negatively impact duration of the CI/CD pipeline,
- R5: tool execution results should be achievable and versionable, e.g., as build artifacts of the build pipeline.

The tool is implemented in Python, making a heavy-use of NetworkX graph library which outputs graphs as Graph Exchange XML Format (GEXF) files. For the means of visualisation, an open-source tool named *Gephi* [3] was used which can directly import these graphs (requirement R5).

Python was selected since SCons is already Python-based and hence Python and pip (package manager for Python) infrastructure is available on all development machines and build servers already. Furthermore, many involved developers are proficient in Python allowing for a more simple maintenance of the tool.

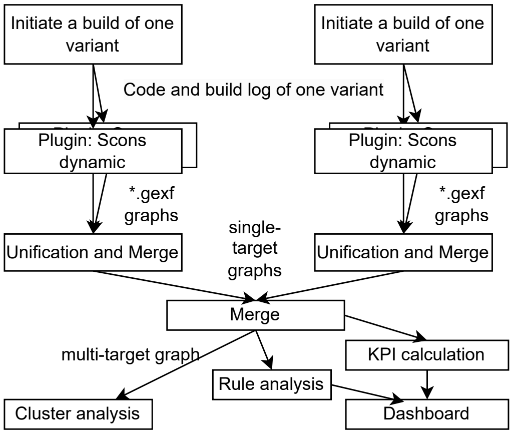


Figure 1: Overview of VAAT analysis steps.

The workflow of VAAT consists of multiple consequent steps which are summarized in Figure 1.

In the first step, a set of "miner" plugins (following R3) construct different graphs covering various aspects of code base to be analyzed is run on each variant. In the subsequent steps those graphs are normalized and merged into one graph containing different analysis aspects.

In the following, we provide a detailed overview of those steps.

3.1 Graph Construction

We use directed, multi-labeled graphs. Nodes represent file-system element like directories, SCons file and artifact files. The type of the node is attached to the node using a property-label. Additionally the references between nodes are denoted using labeled edges, e.g., a “contains" edge between a directory and a target or a “requires" reference between two Python files.

Based on the source code repository, the VAAT toolkit runs a set of “miner" plugins to extract different information aspects from the build artifacts. Miners might have logical dependencies, e.g., a duplicate miner might depend on the discovery of previously involved artifacts. Those dependencies are resolved by the execution order of miners, most of the plugins can be run independently of each other.

In the following, we give an overview of used plugins.

Static SCons Analysis. The core of VAAT is the analysis of SCons build environments and discovery of involved artifacts. We started with a static SCons analysis approach: A script was created to parse the SCons files recursively and to detect the inclusion of other SCons scripts. Furthermore, source files and other artifacts were detected and added into a graph. The initial static plugin was based on a set of regular expressions parsing the SCons build system starting with the top-level `SConscript` file performing a depth-first traversal of all included scripts.

This initial heuristic approach has expectedly shown a number of shortcomings related to the powerful SCons syntax including relative and absolute paths usage options when including dependent SCons files and source code artifacts as well as with mixed-in Python based influencing the build process, e.g., conditional inclusion of specific folders containing source code. Therefore, the static approach was providing an over-approximation of build artifacts which was not sufficient enough for envisioned analysis goals. Based on the experiences we moved on the the dynamic SCons analysis approach.

Dynamic SCons Analysis. Compared to the static approach, the dynamic or trace-based approach is based on markers extracted from the actual build log of a compilation target. The analysis of the build log allows to have accurate results considering actually executed build steps (after the evaluation of mixed-in Python logic, R1). Furthermore, there is no additional overhead in terms of time needed for the compilation due to the envisioned usage of the tool within the build pipeline (R2, R4), where the build process is executed anyways.

Our first approach to modify our own `SConscript` files inheriting from SCons classes failed due to a quite complex internal structure of the Python code within the SCons system. The second approach was to actually modify the SCons build system with log items emitted every time an SCons file is entered (via inclusion) or left by the build system. A subsequent scan of the marks allows to reproduce the inclusion stack of SCons files and build ab the graph of actual SCons file which were traversed during the build of the particular firmware target.

Maintainers of the SCons project have accepted our tracing changes which are available by using the flag `–debug=sconscript` in SCons versions starting with 4.7.0 [8].

Duplicate Analysis. A third miner plugin described in this paper is a simple duplicate detection tool used on the files discovered by the first two plugins. Here we simple calculate md5 hash values of all discovered files and connect respective nodes in the graph with a dedicated relationship in case of the hash equivalence. The duplication shows where files have been simply copied from one variant to another instead of referencing those.

Additional miners. In addition to the three presented miners, we run a number of additional miners which are more tailored to our in-house tooling (see R3) and are of a lesser interest for the community.

3.2 Graph Unification and Merging

Once a graph has been created per miner plugin and a variant, we need to merge these graphs into a graph containing all kind of references originating from various plugins for each variant, so-called *single-variant graphs.* Since graphs represent file system structure with custom relationships and properties, the merging is quite simple.

An iterative algorithm traverses the graph starting with the root build folder and merges the nodes while normalizing the paths strings - making sure all paths are relative to the root build folder and all special symbols like slashes are normalized. Furthermore, nodes representing files system directories are created, such that each file node has a corresponding directory node which it can connect to via a “contains" edge.

Additionally, we merge all single-variant graphs into a *multi-variant graph* where file system objects (graph vertices) additionally contain the name of the particular variant this object was involved into the build process. An example is shown in Figure 2.

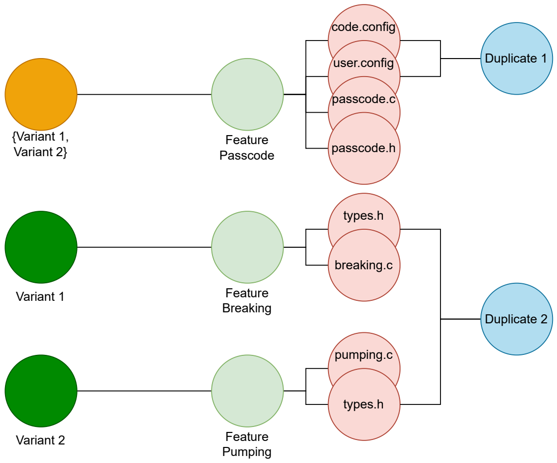


Figure 2: An example showing the graph structure. In total two variants have been analysed and they have one directory in common where one file is even a simple duplication of each other. Also there is one duplication across different directories.

3.3 Analysis of Single-variant Graphs

Single-variant graphs are analyzed using classical graph-theory-based KPIs such as:

- Node degree: maximum number of incoming/outgoing edges for a node. High numbers indicate common hot-spots which are probably the platform code.

- File include depth: the depth of the nesting of SCons build artifacts indicates the complexity of the build system where high numbers may be a hint for a need of refactoring.

We are currently in a process of domain-experts discussions regarding KPIs and their continuous monitoring (cf. Section 3.5)

3.4 Analysis of Multi-variant Graphs

Once all single-variant graphs are merged into a multi-variant graphs, i.e., a graph containing labels for each product variant the previously described KPIs can be put in a different context. For example, a file with a high degree is a hint for shared or platform code and should therefore contain many if not all variant labels.

So far, only monitoring-centring use-cases were covered by existing KPIs. By having a multi-variant graph, we can finally use cluster analysis of the code base according to build variants to initiate a refactoring of existing SCons artifacts. For example, we can build Venn-diagram-like structure for parts of the SCons build system. This allows localizing:

- Platform code, i.e., code which is shared among all variants,
- Multi-variant code, code which is shared between multiple products (e.g., hardware-specific code related to a particular variant or a specific feature like “Pump Cleaning" for VFDs of the water pumping applications),
- Variant-exclusive code, i.e., code which is exclusively used by one specific variant.

Derived clusters were reviewed during interviews with domain experts and are base for a subsequent refactoring of the build system organization and development of a rule-set to prevent erosion of the refactored solution (cf. future work in Section 5).

3.5 CI/CD Integration

In order to receive feedback fast and also to integrate each developer into the process of structuring the variants the calculation of the metrics is integrated into the CI/CDs pipeline. With each commit, the pipeline is executed, the graphs generated and the metrics calculated. This leads to an immediate feedback if the latest commit did improve the software quality and led to less duplications or less dependencies. To align with R4 the execution time of the whole metrics calculation should be as little as possible to not add too much slack time into the whole process.

4 Initial Results

As described in the prior chapters, the analysis contains different graph node types like files, directories or variants. This lead easily to a complex graph. When searching for overlapping software fragments over multiple variants the graph gets huge easily. One real example is shown in Figure 3. Even without being able to read the labels of each node, it is clear that the amount of different clusters across different variants is challenging. In total 4.041 nodes and 8.314 edges represent the analyzed software structure of more than 20 different variants. The following types are represented in the graph in descending order: files (pink), directories (light green), duplicate clusters (blue), target clusters (orange), variants (dark green). Figure 3 *A* shows the code base that is used by all variants - the platform code. Figure 3 *B* is a cluster with a lot of duplicates, especially across configuration files. Figure 3 *C* is just referenced by one variant, so it is a feature of this specific variant.

In total, around 400 duplication clusters with almost 2.000 files have been identified. This means, that there are multiple files copied all over the whole workspace reducing the overall maintainability. This also helps to identify features or common functionality across multiple variants.

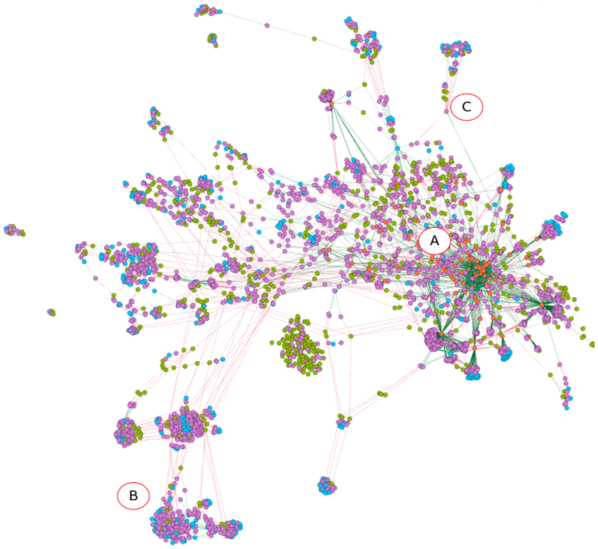


Figure 3: An overview image of a multi-variant graph for considered SPL.

5 Outlook and Future Work

In this work we proposed a newly introduced VAAT tooling tailored for the specific needs of projects using the SCons build tool. We reviewed strategies of coping with unprecedented flexibility of the SCons tool chain like the possibility to include and execute arbitrary Python code as well as our initial results in using graph-based and cluster-based analysis towards improving and monitoring the quality of the SPL.

Future work includes the following topics:

- Extension of context-specific VAAT rules, e.g., ensuring that all shared code should reside in a “shared" or a “common" file system directory or within files with a specific prefix.
- Checking for similarity in files. Not only checking for duplicates, but also find files that have been copied and adapted slightly.
- Integration of the rule-based monitoring into the CI/CD pipeline and dashboards.
- Development of an additional toolkit for a “Pythonic" redesign of the SPL build chain using object-oriented class abstractions in conjunction with SCons.

Acknowledgments

We thank Sergey Ostroumov, Mikko Norrkniivila and Matti Kauhanen for insights and feedback during the project work.

References

[1] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf H. Reussner. 2022. A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications. *Empir. Softw. Eng.* 27, 5 (2022), 101.

[2] Sofia Ananieva, Sandra Greiner, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Thomas Kühn, Christoph Seidl, and Ralf H. Reussner. 2022. Unified Operations for Variability in Space and Time. In *VaMoS*. ACM, 7:1–7:10.

[3] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. 2009. Gephi: an open source software for exploring and manipulating networks. In *Proceedings of the international AAAI conference on web and social media*, Vol. 3.

[4] Andreas Burger and Sten Grüner. 2018. Finalist 2: Feature identification, localization, and tracing tool. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 532–537.

[5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. In *SPLC (1)*. ACM, 21–30.

[6] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *ICSME*. IEEE Computer Society, 391–400.

[7] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *ICSE (2)*. IEEE Computer Society, 665–668.

[8] Sten Grüner. 2023. Adding tracing of scons script calls. https://github.com/SCons/scons/commit/367b0cd3ea065f280daf0835131ac0483b1dfa6a, last visited 10/15/2024.

[9] Sten Grüner, Andreas Burger, Hadil Abukwaik, Sascha El-Sharkawy, Klaus Schmid, Tewfik Ziadi, Anton Paule, Felix Suda, and Alexander Viehl. 2019. Demonstration of a toolchain for feature extraction, analysis and visualization on an industrial case study. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, Vol. 1. IEEE, 459–465.

[10] Sten Grüner, Andreas Burger, Tuomas Kantonen, and Julius Rückert. 2020. Incremental migration to software product line engineering. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 11 pages. https://doi.org/10.1145/3382025.3414956

[11] Daniel Hinterreiter, Lukas Linsbauer, Kevin Feichtinger, Herbert Prähofer, and Paul Grünbacher. 2020. Supporting feature-oriented evolution in industrial automation product lines. *Concurr. Eng. Res. Appl.* 28, 4 (2020).

[12] Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. 2022. Feature-oriented clone and pull operations for distributed development and evolution. *Softw. Qual. J.* 30, 4 (2022), 1039–1066.

[13] Steven Knight. 2005. Building software with SCons. *Computing in Science & Engineering* 7, 1 (2005), 79–88.

[14] Lukas Linsbauer, Stefan Fischer, Gabriela Karoline Michelon, Wesley K. G. Assunção, Paul Grünbacher, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2023. Systematic Software Reuse with Automated Extraction and Composition for Clone-and-Own. In *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer International Publishing, 379–404.

[15] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In *SPLC*. ACM, 101–110.

[16] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2023. Bottom-Up Technologies for Reuse: A Framework to Support Extractive Software Product Line Adoption Activities. In *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer International Publishing, 355–377.

[17] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2022. Evolving software system families in space and time with feature revisions. *Empir. Softw. Eng.* 27, 5 (2022), 112.

[18] Sarah Nadi and Richard C. Holt. 2014. The Linux kernel: a case study of build system variability. *J. Softw. Evol. Process.* 26, 8 (2014), 730–746.

Providing the Variation Control System ECCO as a Service

Tobias Bergthaler
Johannes Kepler University
Linz, Austria
Bergthaler.t@gmx.at

Matthias Preuner
Johannes Kepler University
Linz, Austria
matthias.preuner@gmail.com

Paul Grünbacher
Johannes Kepler University
Linz, Austria
paul.gruenbacher@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

Abstract

Variation control systems (VarCS) like ECCO or SuperMod provide capabilities for uniformly handling revisions and variants of software systems based on features, in that regard going beyond the capabilities of existing version control systems like Git. However, existing VarCSs have not been designed to be provided as a service, which limits their integration in web-based engineering environments and tool pipelines. In this paper, we present Software-as-a-Service (SaaS) extensions for the VarCS ECCO, which also include a REST API to facilitate its integration in different engineering processes. We present ECCO's SaaS architecture and explain its services for managing repositories, as well as local and distributed operations for feature-based development. We demonstrate the ECCO service by showing its integration with ECCOHub, a web-based platform for working with ECCO, as well as ECCO's CI/CD pipeline.

CCS Concepts

• **Software and its engineering → Software product lines**; **Software configuration management and version control systems**; *Software maintenance tools.*

Keywords

Servitization, variation control systems, REST API, SaaS

ACM Reference Format:
Tobias Bergthaler, Matthias Preuner, Paul Grünbacher, and Alexander Egyed. 2025. Providing the Variation Control System ECCO as a Service. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715429

1 Introduction

Servitization is the process of providing existing software solutions as a service, thereby enabling users to access and use software without the need for acquiring and maintaining a specific infrastructure [6, 19]. Providing software as a service (SaaS) is also of high interest for software engineering tools, as it facilitates their use by simplifying installation, updates, and maintenance. A popular example are Version Control Systems (VCSs) like GitHub, GitLab, or Bitbucket [18], which rely to a large extent on web-based services to support the collaboration of teams in open-source or closed-source projects. Such VCSs allow users to create repositories for storing and managing code and other artifacts via cloud-based services, thus facilitating the collaboration of engineers from multiple teams and organizations. However, web-based VCSs also provide integration with services like code review, project management, or the integration with CI/CD pipelines.

Researchers in software product line engineering have been developing Variation Control Systems (VarCSs), which support managing both revisions and variants [17], a capability missing in conventional VCS. VarCSs adopt a feature-oriented approach [1, 3] for developing customer-specific systems and services. Engineers collaboratively evolve product lines by creating new features or by selectively reusing and adapting existing ones [12]. Therefore, the issue of managing revisions and fine-grained, feature-level variants becomes essential for teams in product line engineering. VarCSs thus map features to parts of the code or other artifacts implementing the features. When adding a new feature or revising an existing one, the internal repository representation is updated, thus recording the revision history of each feature, but also augmenting the knowledge base for creating new variants in the future. Features can then be combined to compose new product variants, even if they have not been explicitly committed as such before. VarCS thus go beyond the capabilities of widely used VCS.

Despite these benefits, existing VarCSs like ECCO or SuperMod have not been designed to be provided as a service, so their integration in web-based engineering platforms and tool pipelines is currently limited. In this variability-in-practice paper, we therefore present a technical solution and experiences of extending the VarCS ECCO (Extraction and Composition for Clone-and-Own) [14] with a REST API to allow its integration with the ECCOHub, a web-based platform supporting feature-based engineering with ECCO.

The paper is structured as follows: Section 2 discusses the background on VarCSs and the key concepts of ECCO. Section 3 presents the architecture of our ECCO-based service and the services for managing repositories, support for local feature-based operations (commit, checkout), as well as capabilities for distributed feature-based operations (pull, clone). Section 4 looks at two use cases of the REST API, focusing on ECCOHub and the ECCO CI/CD

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715429

tool pipeline. Section 5 discusses experiences and lessons learned. Finally, Section 6 provides a summary and an outlook.

2 Background

Version control systems track revisions and supports variants (via clones and branches) and thereby support extensional versioning, i.e., they make the entire project history accessible and restorable at any time. For instance, the widely-used VCS Git is at the heart of common platforms like GitHub, GitLab, or BitBucket.

However, it has been pointed out that while VCS are highly capable in handling revisions of artifacts, they have deficiencies with respect to managing variants and features [17], which is also the result of their line-based tracking of differences between revisions.

Feature-based Handling of Revisions and Variants. The available branching and forking mechanisms conceptually create clones. This considerably increases the maintenance effort as changes will need to be propagated manually to all relevant clones. To overcome this issue, variation control systems (VarCS) [17] provide capabilities for uniformly handling both revisions and variants based on decomposing artifacts into finer-grained entities, which are then mapped to features. Similar to VCS, the VarCSs allow to retrieve versions that have been explicitly stored (a.k.a. extensional versioning). However, VarCSs also support intensional versioning, they allow to compose a product variants based on features, even if the desired feature combination has not been submitted explicitly before.

AST-based Diffing. The line-based diffing approach used to determine changes is a serious shortcoming of current VCSs, while VarCSs address the diversity of engineering artifacts encoded in source code, DSLs, or vendor-specific formats. In particular, when computing differences between versions, the line-based comparisons of artifacts will not suffice, as feature-based collaboration in VarCS relies on automatically creating feature-to-artifact mappings for often complex artifacts. VarCS thus compute differences based on the abstract syntax tree representations created from artifacts. This provides higher flexibility regarding the granularity and modularity of features when building the internal representation compared to a line-based approach to determining differences [14].

The VarCS ECCO. The tool supports product line engineering with capabilities for automatic feature tracing and composition [8, 14, 16]. It analyses the code base of multiple product versions (revisions or variants) comprising different feature configurations. The ECCO algorithm identifies associations between code artifacts and features. This knowledge can then be used to create new variants of the product for arbitrary combinations of features. ECCO internally handles artifacts in a graph representation and associates nodes to the features [15]. To enable feature extraction and feature-based composition for different kinds of artifacts, ECCO can be extended with plugins translating artifacts into its internal tree-based structure and vice versa. In particular, since ECCO extracts the features automatically based on the abstract syntax tree of a code file [2], it needs a special parser for each file type. For instance, Hinterreiter et al. [11] extended ECCO to manage mappings between features and their implementation in the DSL IEC 61131-3, while Grünbacher et al. used features managed in ECCO to support the evolution of digital music artifacts encoded in the DSL LilyPond [9, 10].

3 ECCO's SaaS Architecture

Making the existing VarCS ECCO[1] available as a service involved developing several extensions for handling multiple users and repositories simultaneously. The SaaS Architecture extends the existing ECCO architecture to support multiple isolated tenants by ensuring security. Specifically, we separated different functionalities (e.g., authentication, repository management) as shown in Fig. 1. We implemented an authentication mechanisms as well as role-based access control to manage permissions. We also created the user-friendly dashboard EccoHub for managing repositories and for interacting with them through ECCO's operations. Most importantly, we developed a comprehensive API to interact with the VarCS. We adapted and extended ECCO using the REST (Representational State Transfer) architectural style intended for designing scalable, adaptable, and maintainable networked applications [7, 21]. Furthermore, we provide executable Docker images significantly simplifying the deployment of ECCO servers.

3.1 Layers

Figure 1 shows ECCO's three-layered architecture making its capabilities available as a service to both the ECCO CI/CD pipeline and the EccoHub frontend:

Ecco Core. This layer comprises the existing ECCO implementation, extended with capabilities for storing repositories and handling user services. Specifically, the core is separated in multiple parts. The service is the connector to the interfaces handling all requests coming from the user via the UI or Command Line Interface (CLI). The service is responsible for loading and opening projects, and for extracting features and artifacts. The base consists of multiple classes needed to build the repositories committed to ECCO. The most important classes are repository, commits, features, variants, feature revisions and configurations, where each of the classes contains information about the repositories stored in ECCO. Storage handles the storage process of the repositories, containing all data and meta information.

Ecco Rest API. We provide multiple mechanisms for interacting with the ECCO core: the already existing graphical user interface (GUI) and a CLI, as well as the newly developed ECCORest webservice allowing access via the REST API. REST has become the de facto standard for designing web APIs and web-based applications and ensures a broad use across different languages and tools. Components in REST communicate via HTTP requests, usually by sending XML or JSON files. Our REST implementation is integrated in the ECCORest Layer of our architecture. The RESTful backend consists of four controllers handling the REST API requests. It can be accessed by ECCOHub, but is also available for other third-party components. We created a representation of the base classes as so-called models, which encapsulate and restrict functionality. This gave as more control over what we send with the REST API and send only the data needed for the frontend. This also increases performance and makes the REST implementation more flexible against changes in ECCO's base. The so-called controller classes are the interface of the REST API towards the frontend. They define the possible REST API calls, how they can be called, what data they send and receive. In addition, a token-based user

[1] https://github.com/jku-isse/ecco

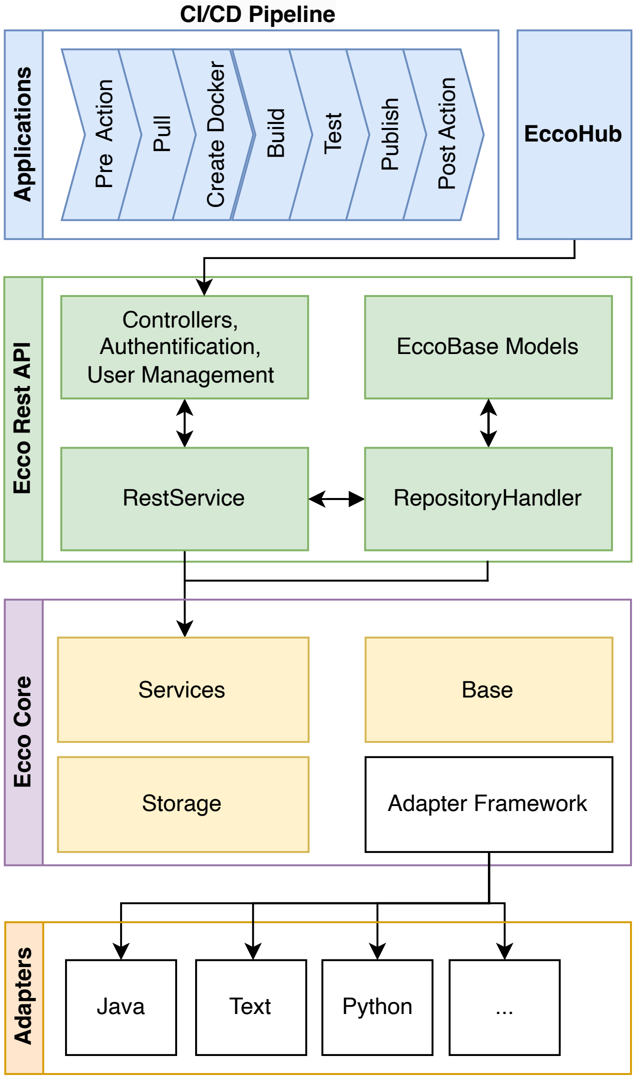


Figure 1: ECCO's SaaS architecture.

authentication ensures that only permitted users have access to ECCOHub and ECCO repositories. The backend implementation runs on a Micronaut server (https://micronaut.io) and is available at https://github.com/jku-isse/ecco/tree/0.1.9.

Adapters. ECCO supports feature extraction and feature-based composition for different kinds of artifacts. It uses adapters to translate different types of artifacts into ECCO's tree-based structure (internalization after commits) and back from that structure to actual artifacts (externalization after checkouts). Existing ECCO adapters work without changes in the SaaS architecture.

3.2 Services

Managing Repositories. ECCO provides services to create new repositories and to delete existing ones, similar to other systems like GitHub or BitBucket.

Feature-based Commits and Checkouts. ECCO services allow to commit new or revised features to ECCO repositories. Variants can be checked out from the repository by defining the required features (and their revision) contained in the repository.

Feature-based Clones and Pulls. ECCO further provides an API with feature-oriented operations to selectively clone a repository from an existing one as well as to pull individual features from one repository to another one. This is useful when working with multiple product variants concurrently. An earlier evaluation also showed this pull operation works for different cases of feature interactions [13].

4 Applications of ECCO SaaS

We demonstrate the application of the ECCO service in two use cases: The ECCOHub, a web-based frontend supporting the ECCO capabilities in collaborative workflows of engineers, and the ECCO CI/CD pipeline using ECCORest to test and deploy the services.

ECCOHub. The web-based tool ECCOHub uses servitization to reduce the effort required of working with ECCO with an intuitive web interface for its users. ECCOHub includes capabilities for managing ECCO repositories, feature-based commits and checkouts supporting individual developers, as well as feature-based clone and pull operations supporting feature-based collaboration. ECCOHub has been developed using the React Framework (https://reactjs.org/) and its implementation is available online[2].

Using the web-based user interface, developers can `Commit` and `Checkout` existing and new variants from a selected repository, either for use in production or to continue development on a feature. The `Pull` command allows retrieving a subset of features from an existing into a new repository, i.e., feature associations of the selected features are copied to the new repository. A special case is the `Clone` command for selectively cloning a repository by defining the required features [13].

Configuration expressions play an important role in these operations. These are comma-separated lists of feature names used to define the combinations of features, also including the revisions of the involved features. To handle configuration expressions more comfortably for checkout operations, ECCOHub allows managing named variants defining the included features and their revisions.

CI/CD pipeline. DevOps practices are nowadays regarded essential for the efficient and effective delivery, management, and operation of services. In particular, DevOps emphasizes automation in the deployment pipeline. This allows teams to deliver updates and new features to services quickly and reliably, based on practices such as infrastructure provisioning, code deployment, testing, and monitoring. Therefore, we developed the ECCO CI/CD pipeline (cf. Figure 1) to improve the development and evolution of the ECCO services. The pipeline process retrieves the current ECCO repository and creates a Docker image for it. The pipeline builds ECCO and ECCORest, thereby also loading all dependencies. It further executes end-to-end tests, to check the integrated operation of ECCOHub and ECCORest. In case the tests are successful, the pipeline

[2] https://github.com/jku-isse/ecco-web-client/tree/release_1.0

creates Docker images[3] for EccoRest and the ECCOHub to facilitate the hosting of ECCO services.

The pipeline was developed for Jenkins. It is manually triggered by a user requesting a new version (Pre-Action). The pipeline pulls the newest version of the ECCO repository from GitHub (Pull). The pipeline creates a Docker image for our virtualization, which contains all needed dependencies. The pipelines builds the pulled ECCO repository within the Docker image. The pipeline tests the created Docker image with the built ECCO repository. The pipeline publishes the created Docker image to the platform Docker Hub. After successfully publishing the Docker image the pipeline ends by cleaning up the pipeline (Post-Action).

The pipeline uses build tool Gradle (https://gradle.org/) for building and testing the ECCORest implementation. For the end-to-end tests cypress (https://www.cypress.io/) is used to execute user actions on ECCOHub, which then calls the REST API provided from ECCORest. The end-to-end tests execute all core functions of the ECCOHub, thereby checking that the docker images are working as expected.

5 Experiences and Lessons Learned

We report several experiences and lessons learned when extending ECCO towards a service.

Design of multi-tenant operations. We had to adapt and extend the original design and implementation of ECCO to allow concurrent multi-user interactions. In particular, the architecture was adapted so that for each opened repository one *ECCO Service* is now instantiated by one *RepositoryHandler*, thereby avoiding the otherwise time-intensive switches between repositories.

Data representation for REST API. When implementing the REST API we created a data representation of ECCO's base classes, which enabled us to send only their essential attributes, e.g., only the name, version, and relations between features. Furthermore, we had to get rid of circular references between Java objects when creating this representation.

Changes of existing features. The servitization process led to significantly reviewing, refactoring, and improving existing features. For instance, our new implementation required changes to store ECCO repositories in user-defined locations. Similarly, the use of the Micronaut server improved error handling and reliability by preventing the termination of the services.

Importance of the CI/CD pipeline. The development process significantly benefits from a CI/CD pipeline for automatically integrating, testing, and deploying ECCO's functionality in an highly iterative manner. In addition, the pipeline provides an fully working containerized version of ECCO, which further simplifies deployment.

6 Conclusions and Outlook

We presented Software-as-a-Service (SaaS) extensions for the Variation Control System (VarCS) ECCO, which significantly enhance its utility and integration capabilities. In this way we want to make ECCO and its extensions better available to a broader audience and developer community to provide the advantages of variation control systems to new fields of applications. This is achieved through two technically distinct implementations: On the one hand, the ECCORest offers a broad range of applications, from simple usage to integration into DevOps processes. On the other hand, the web frontend provides an easy-to-use application that requires no additional installation. Specifically, the ECCO REST API allows its use in the ECCOHub collaboration platform but also facilitates its integration in with third-party tools and tool pipelines [20]. However, more work is needed to facilitate deployment for large-scale environments and in diverse infrastructures.

By providing a REST API and facilitating its incorporation into various engineering processes, ECCO's SaaS architecture addresses the limitations of ECCO, which was not designed a as service allowing the integration in modern, web-based engineering toolchains Future work will involve extending both the frontend and backend to support more specific use cases. Specifically, the work described in this paper is only a first step towards a platform supporting feature-aware collaboration in engineering teams. The collaboration features (e.g., issue tracking, code reviews, as well as user management) of platforms like GitHub or GitLab are certainly a main reason for their success. For instance, they use feature branches as a mechanism to add new or modify existing features, and pull requests to integrate changes of engineers in distributed workflows in a controlled fashion. Feature-based development is well suited for distributed development as, e.g., shown by the extensive use of feature branches in practice. Similar support has also been explored for VarCSs: Hinterreiter et al. [12, 13] extended ECCO with feature-based clone and pull operations for transferring features from one repository to another. However, such advances are only a first step with respect to feature-aware operations for collaborative and distributed development. In particular, the DesignSpace project [4, 5] showed that more flexibility is desirable in supporting different modes of collaboration in engineering workflows. This is certainly the case with features that cut across multiple engineering domains. Providing ECCO as a service allows us to more easily integrate it with the DesignSpace platform in our future research for improving variability management.

This work can benefit other researchers by providing new avenues for using and integrating Variation Control System (VarCS) in diverse engineering and research contexts. This can promote interdisciplinary research, allowing ECCO to be applied in new fields, including software product lines, systems engineering, and even non-software domains where variation control might offer benefits.

Acknowledgments

This research was supported in part by the Austrian Science Fund (FWF) [10.55776/P31989], by the JKU Linz Institute of Technology (LIT) and the state of Upper Austria [LIT-2023-12-SEE-128].

References

[1] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines – Concepts and Implementation.* Springer.

[2] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* 368–377. doi:10.1109/ICSM.1998.738528

[3] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Proceedings 19th Int'l Software Product Line Conference* (Nashville, USA). 16–25.

[3] https://hub.docker.com/r/issejku

[4] Andreas Demuth, Markus Riedl-Ehrenleitner, and Alexander Egyed. 2016. Efficient detection of inconsistencies in a multi-developer engineering environment. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 590–601. doi:10.1145/2970276.2970304

[5] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. 2015. DesignSpace: an infrastructure for multi-user/multi-tool engineering. In *Proc. 30th Annual ACM Symposium on Applied Computing*. 1486–1491. doi:10.1145/2695664.2695697

[6] Yves Ducq, David Chen, and Thècle Alix. 2012. Principles of Servitization and Definition of an Architecture for Model Driven Service System Engineering. In *Enterprise Interoperability*, Marten van Sinderen, Pontus Johnson, Xiaofei Xu, and Guy Doumeingts (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–128.

[7] Roy Thomas Fielding. 2000. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine.

[8] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proceedings 30th IEEE International Conference on Software Maintenance and Evolution*. 391–400. doi:10.1109/ICSME.2014.61

[9] Paul Grünbacher, Rudolf Hanl, and Lukas Linsbauer. 2021. Using Music Features for Managing Revisions and Variants in Music Notation Software. In *Proceedings of the Int'l Conference on Technologies for Music Notation and Representation* (Hamburg, Germany). Hamburg, Germany, 212–220.

[10] Paul Grünbacher, Rudolf Hanl, and Lukas Linsbauer. 2023. Using Music Features for Managing Revisions and Variants of Musical Scores. *Computer Music Journal* 47, 3 (09 2023), 50–68. doi:10.1162/COMJ_a_00691

[11] Daniel Hinterreiter, Lukas Linsbauer, Kevin Feichtinger, Herbert Prähofer, and Paul Grünbacher. 2020. Supporting feature-oriented evolution in industrial automation product lines. *Concurrent Engineering* 28, 4 (2020), 265–279. doi:10.1177/1063293X20958930

[12] Daniel Hinterreiter, Lukas Linsbauer, Paul Grünbacher, and Herbert Prähofer. 2021. Feature-Oriented Clone and Pull for Distributed Development and Evolution. In *Proceedings of the 14th Int'l Conference on the Quality of Information and Communications Technology*. 67–81. doi:10.1007/978-3-030-85347-1_6

[13] Daniel Hinterreiter, Lukas Linsbauer, Paul Grünbacher, and Herbert Prähofer. 2022. Feature-Oriented Clone and Pull Operations for Distributed Development and Evolution. *Software Quality Journal* 30, 4 (2022), 1039–1066.

[14] Lukas Linsbauer, Stefan Fischer, Gabriela Karoline Michelon, Wesley K. G. Assunção, Paul Grünbacher, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2023. Systematic Software Reuse with Automated Extraction and Composition for Clone-and-Own. In *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*, Roberto E. Lopez-Herrejon, Jabier Martinez, Wesley Klewerton Guez Assunção, Tewfik Ziadi, Mathieu Acher, and Silvia Regina Vergilio (Eds.). Springer International Publishing, 379–404. doi:10.1007/978-3-031-11686-5_15

[15] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. 2013. Recovering Traceability between Features and Code in Product Variants. In *Proceedings of the 17th International Software Product Line Conference* (Tokyo, Japan) *(SPLC '13)*. Association for Computing Machinery, New York, NY, USA, 131–140. doi:10.1145/2491627.2491630

[16] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software & Systems Modeling* 16 (Jan 2017), 1179–1199. Issue 4.

[17] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of Variation Control Systems. *Journal of Systems and Software* 171 (2021), 110796.

[18] Stefan Otte. 2009. Version control systems. *Computer Systems and Telematics* (2009), 11–13.

[19] Srinivasa Raghavan R, Jayasimha K.R, and Rajendra V. Nargundkar. 2020. Impact of software as a service (SaaS) on software acquisition process. *Journal of Business and Industrial Marketing* 35, 4 (2020). doi:10.1108/JBIM-12-2018-0382

[20] Alex Rodriguez. 2008. Restful web services: The basics. *IBM developerWorks* 33 (2008), 18.

[21] Erik Wilde and Cesare Pautasso (Eds.). 2011. *REST: From Research to Practice*. Springer, New York, NY. doi:10.1007/978-1-4419-8303-9

Experiences in Developing Configurable Digital Twin-assisted xR Applications for Industrial Environments

Richard May
Harz University of Applied Sciences
Wernigerode, Germany
rmay@hs-harz.de

Simon Adler
Harz University of Applied Sciences
Wernigerode, Germany
simonadler@hs-harz.de

Abstract

The integration of digital twins, i.e., virtual replicas of physical systems, is increasingly transforming manufacturing by enhancing efficiency through real-time monitoring, simulation, and optimization. The 3D-visualization of their data as a core functionality of xR applications (e.g., Virtual Reality) extends their usefulness and can be used as an important tool for teaching, training, and support. However, in addition to already known and well-discussed challenges (e.g., data representation), developing digital twin-assisted xR applications poses various variability challenges due to the complexity of manufacturing processes, data models, and the need for configurability across various scenarios and platforms. In this paper, we share our experiences in developing such applications, focusing on the gap of handling variability. Based on the DigiLehR research project, which also includes three industrial use cases as configurable products of an xR application family, we describe challenges we faced during development and essential lessons learned. Here, we particularly focus on platform specifics, immersion and interaction, digital twin-related data fragmentation, accessibility, and security. Overall, our work aims to create awareness for practitioners and researchers about the challenges of developing digital twin-assisted xR applications and their configurations, encouraging discussions on their efficient application in industrial settings.

CCS Concepts

• **Software and its engineering → Reusability**; • **Computer systems organization → Embedded and cyber-physical systems**; • **Human-centered computing → Interaction paradigms**.

Keywords

extended reality, virtual reality, augmented reality, digital twins, configuration, modularization, manufacturing, industry 4.0

ACM Reference Format:
Richard May and Simon Adler. 2025. Experiences in Developing Configurable Digital Twin-assisted xR Applications for Industrial Environments. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715434

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715434

1 Introduction

With the rise of Industry 4.0, information and communication technologies are increasingly being integrated into manufacturing processes. The value added by industrial machines now extends beyond the manufactured product to include the generation of digital data during manufacturing [36, 38]. This data can be used for various new fields and associated applications, with *digital twins* standing out as an area of particularly high potential. Digital twins are comprehensive virtual replicas of physical systems that facilitate real-time monitoring, simulation, and optimization of manufacturing processes [40, 42]. In more detail, they typically denote a quality criteria about the digital data which should at least reflect the current state of industrial machines but could also include historical signal data, documents, or process data [1, 11]. By utilizing such data, companies can gain valuable insights into their manufacturing processes, leading to significant enhancements in efficiency and productivity, for example, based on predictive maintenance [30, 37].

Although 3D-visualizations are not a key requirement for digital twins, they offer another way to effectively utilize their data [16, 45]. Such visualizations can be particularly valuable for teaching, training, or supportive purposes [23, 39] with a high potential for mobile as well as EXtended Reality (xR) applications, i.e., typically Virtual Reality (VR) and Augmented Reality (AR) [43, 49]. This is why, there is already numerous research on such applications, their challenges, and potentials, ranging from educating students [39] to supporting practitioners in industrial environments [25]. For example, Hazrat et al. [17] reported on utilizing digital twins in engineering education to facilitate the training of human-centric decision-making and Kuts et al. [22] developed a VR-based digital twin-assisted factory environment for learning purposes. Calandra et al. [7] proposed an xR application, which allows collaboratively programming a digital twin-assisted robot. Moreover, Kaarlela et al. [19] presented scenarios of digital twin-assisted safety and emergency training.

Overall, we argue that developing xR applications relying on digital twins is highly challenging — not only due to their increasing complexity (e.g., manufacturing processes, data models, data visualization) [2, 20, 48], but also due to growing demands to efficiently adapt the xR application to dynamic scenarios and users, as well as to deploy them on different platforms (e.g., VR, AR, and mobile devices) [12, 13]. Consequently, digital twin-assisted xR applications must be highly configurable to address such requirements, leading to various common challenges in handling variability, such as valid configurations and their effective verification [8, 21, 32] or reliable and secure evolution [28, 33]. However, due to the unique peculiarities of digital twins (e.g., data fragmentation [14]) and xR technologies (e.g., 3D-visualization [34]), they pose additional challenges in efficiently handling their features.

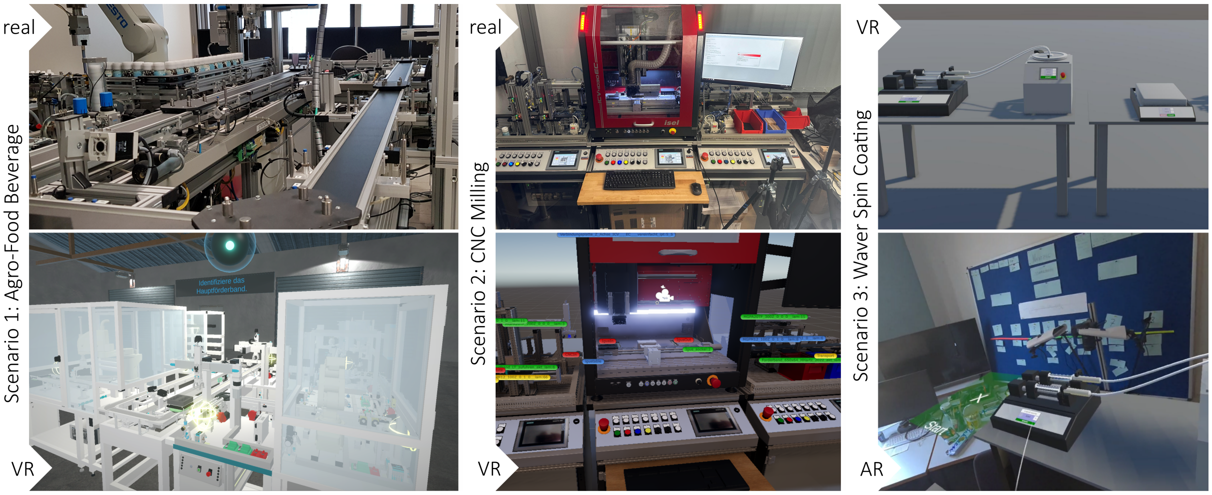

real
Scenario 1: Agro-Food Beverage
Identifiziere das Hauptförderband.
VR
real
Scenario 2: CNC Milling
VR
VR
Scenario 3: Waver Spin Coating
AR

Figure 1: Scenario overview for agro-food beverage (Scenario 1), CNC milling (Scenario 2), and spin coating (Scenario 3).

Although there is not only extensive research related to digital twin-assisted xR applications for Industry 4.0 but also related to managing variability in industrial environments [24, 31, 44], we are not aware of work focusing on both properties, i.e., digital twin-assisted xR applications in Industry 4.0 and their variability. The closest work to ours is by Fernandes and Werner [12, 13] who focused on a web xR application software product line for software engineering education. However, digital twin technology was not part of their study, highlighting the value of our goals. **To address this gap, we aim to share our experiences by reporting challenges we faced and lessons we learned during the development of a family of configurable (i.e., variable) digital twin-assisted xR applications for industrial environments.**

Our insights are based primarily on a research project called *Digital Teaching and Learning in Augmented Realities (DigiLehR)* that investigates the potential of utilizing xR technologies for teaching, training, and supportive purposes. In this context, different industrial scenarios were enhanced by digital twin-assisted xR, including *agro-food beverage*, *CNC milling*, and *waver spin coating* (cf. Sec. 2). To enable reusability of variable features and to restrict additional efforts in resources, we roughly followed a software product line approach [3], i.e., leading to several similar, but adapted applications. Overall, we aim to contribute the following:

- Insights into variability handling in applications for industrial scenarios utilizing both xR and digital twin technologies.
- Challenges and lessons learned in developing configurable digital twin-assisted xR applications.

With our work, we aim to increase the awareness of variability in applications based on xR and digital twin technologies and to spur discussions on handling it efficiently.

2 Industrial Scenarios

While xR applications themselves are already quite complex, the deployment for different scenarios and platforms taking into account digital twin data resulted, not surprisingly, in large efforts. This is why a configurable approach to reuse features was used, which, starting from a basic product with mandatory (i.e., transferable) features and additional optional features, ensured significantly more efficient development. Generally, all scenarios rely on a feature model (Fig. 2) and are implemented based on Unity as well as the xR Interaction Toolkit.

Scenario 1: Agro-Food Beverage *(VR, mobile).* The application consists of an agro-food beverage (AFB) (Fig. 1, left) machine, including seven assembly units, driven by Siemens S300 PLC each. It realizes a circular process with a bottle storage and a module to extract filled, and return empty bottle packages. In addition, it mainly contains discrete actuators and sensors to assert the automated process. With xR-AFB, users should first understand the machine, its parts, their purpose, and how to put the machine into operation. The order in which to start the assembly units is arbitrary; but they depend on each other. So, one learning goal is to differentiate between regular behavior, usual failures, and failures that require expert consultations. For xR the scene is spatially large and geometrically complex. Users must be able to navigate and select components on VR and mobile devices.

Scenario 2: CNC Milling *(VR, mobile).* Here, a CNC milling machine (Fig. 1, middle) was realized, consisting of four assembly units: three are controlled by a Siemens S1500 PLC; one is the isle-CNC main unit. The machine has an input assembly unit with material magazines. Operators can choose a type of material and a drilling recipe. Conveyors will transport the material to the CNC. After completion, the product is transported to an output storage. The operation is standardized, but users must be aware of small steps, that can easily overseen (e.g., hatch must be closed before powering the CNC). The main goal of xR-CNC is the training of operation and how to drill products. The xR scene is less complex than in Scenario 1, but the operation is controlled by human-machine interaction (HMI). Virtual HMI can be easily used on mobile devices by touch gestures; care must be taken in VR due to their small scale.

Scenario 3: Waver Spin Coating *(VR, AR, mobile).* This application contains a waver spin coater (Fig. 1, right), which is a batch processing with manual tasks, including four independent stations. First, users pick up a waver to place it in the spin coater; and select

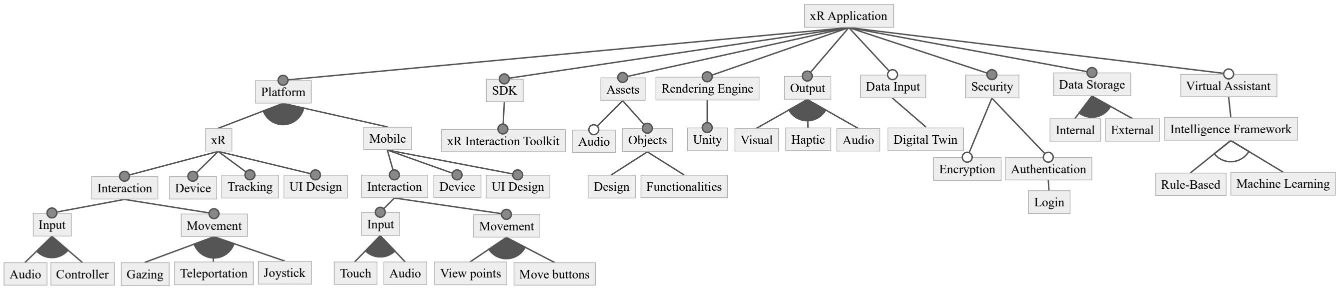


Figure 2: Simplified feature model for configurable xR applications.

a type and amount of liquid from a connected pump station, which will drop the liquid on the waver. Then, users select the temperature, duration, and speed in which the waver will be rotated to distribute the liquid. Finally, the coating quality is measured after selecting the temperature and duration for drying the waver.

Even with professional equipment, each step has variance σ, influencing the outcome. So, users can train how to balance settings (i.e., via HMI), to achieve a result within requirements without wasting material. The experiments are recorded in a protocol monitored live by trainers, if required. The interaction in VR is mostly two-handed (e.g., opening the spin coater with one hand and placing the waver with the other hand). In VR the placement is done manually, for mobile devices animations are used.

3 Challenges and Lessons Learned

Next, experiences and lessons learned we have made during development, in particular related to variability issues, are explained. Fig. 2 provides a general overview of relevant features of the xR applications we relied on. For reasons of simplicity, cross-tree constraints have been excluded here, however, this does not mean that there are none (e.g., joystick movement and mobile platforms).

3.1 Platform Specifics

Challenges. Deploying 3D-visualizations in xR environments and traditional mobile platforms, presents diverse challenges rooted in the individual scenario requirements but also in the distinct hardware capabilities associated with each platform. Even when employing identical visualizations and data, these platforms often require distinct interaction modalities and specific user interface (UI) elements. For instance, VR might require high-resolution graphics, spatial audio, and real-time 3D-rendering to maintain immersion, utilizing motion controllers for interaction with simulated real-world objects [10, 50]. In contrast, AR focuses on overlaying digital information onto the physical environment and typically relies on touchscreens or simple gestures [5, 50]. Meanwhile, mobile devices prioritize touch interactions and responsive design, often employing more traditional 2D-UI elements within simplified 3D-spaces [46]. These differences require the development of platform- as well as device-specific interaction modes, posing challenges in maintaining uniformity in usability and functionality. Integrating and synchronizing these variants while ensuring accurate and consistent data visualization across platforms during evolution adds additional layers of complexity to the development process.

Lessons Learned. To address these challenges, a significant lesson learned is the advantage of using standardized frameworks, such as questionnaires, as a foundation for designing interactions. By categorizing interactions that correspond to assessable questionnaire answers oriented towards each scenario (e.g., CNC milling in Scenario 2), developers can establish structured, platform-specific interaction patterns. This strategy allows the abstraction of essential user engagement components, such as selecting options and navigating environments, which can then be tailored to various input methods available on each platform. Note that although configurability is achieved through questionnaires, the actual possibilities for interaction are limited. Furthermore, leveraging transferable UI technologies (e.g., buttons) through at best platform-independent SDKs (e.g., xR Interaction Toolkit) and engines (e.g., Unity) helps to harmonize interactions and visualizations across devices.

3.2 Immersion and Interaction

Challenges. Typically, xR environments are distinguished by their degree of immersion. High immersion means that user perceive and accept the simulation as realistic and themself as part of the simulation [6]. For this perception of presence, natural interaction methods must be used – in contrast to traditional UIs, which are characterized by more static elements such as buttons and menus [41]. In virtual environments, users can interact, for example, by gestural manipulation of virtual objects, spatial navigation via bodily movements, and virtual assistant support. In this context, there is a high complexity and variability implementing these interactions. Accurate detection and interpretation of variant-rich input modalities requires reliable tracking systems. Furthermore, developers must meet stringent performance demands, balancing high-quality environmental rendering with the need for seamless, responsive interaction processing (cf. Scenario 1).

Lessons Learned. To address these challenges, Hunicke et al. [18] proposed the MDA-framework at concept level, differentiating the development process in phases of Mechanics, Dynamics, and Aesthetics. Developers start by developing mechanics to allow users some dynamics, because users should get into a specific mood. On the other hand, users try to perform a interaction because of a mood and search for mechanics or objects to do so. The processes of developers and users are therefore in contradiction.

Managing variability within the MDA-framework is crucial for accommodating diverse user preferences and device capabilities. This is achieved through user-centric configuration, where users are able to adjust settings like sensitivity or control schemes for personalized experiences, or adaptive systems semi-automatically modifying mechanics based on user behavior to optimize interaction. Reusable designs, objects, and functions that support high immersion facilitate the efficient coverage of variant-rich scenarios.

3.3 Data Fragmentation and Association

Challenges. Developing digital twin-assisted systems is challenging due to the fragmentation of data, their different sources (e.g., different machines or parts of them), and the association between datasets. Because of data fragmentation it is difficult to ensure that data is consistently and accurately represented within the digital twin [14]. Data is collected in multiple formats and update frequencies [51], complicating (real-time) integration into xR applications. So, digital twins and their associated data have a high degree of variability that must be managed accordingly.

Lessons Learned. In our project, different platforms were treated as variants of a Unity base scenario to ensure a centralized data integration. This enables consistent management of data, relationships, and (cross)dependencies between multiple configurations and platforms. Implementing standardized data schemes and APIs facilitate seamless data exchange and transformation. Leveraging middleware solutions (i.e., C# scripts) to harmonize formats allows to create a unified data pipeline within the digital twin framework.

3.4 User Accessibility

Challenges. Motion sickness, or the (temporal) availability of the required xR hardware, requires alternatives to accommodate a diverse user base. This makes configurability an essential requirement in xR applications to tackle this variability issue. Thus, a great challenge is to integrate configuration options into the development project, ensuring that features remain accessible to all users or industrial scenarios regardless of their individual constraints (e.g., grabbing items, such as a waver in Scenario 3).

Lessons Learned. One strategy is to implement visualization techniques guiding users through xR by highlighting interaction targets and providing visual cues (e.g., using specific colors, lights). This makes environments more navigable and less challenging for users prone to motion sickness, as they can control the pacing and intensity of their interactions. We also offered users to switch between xR and offline-working mobile apps, based on their comfort level and restrictions. Additionally, by providing options for different interaction methods based on different platforms as well as additional support by configurable conversational agents [29] we tried to ensure that all users, regardless of their physiological responses or device constraints, can effectively use all scenarios. In our perception, this configurability not only enhanced user satisfaction but also helps in reducing potential discrimination against people.

3.5 Security

Challenges. While digital twins as counterparts to safety-critical machines are known for their ability to reduce functional safety risks [4], there are several challenges related to security, including associated privacy risks. These are typically more related to integrated digital twins based on real-time data than to those based on fixed datasets. Real-time digital twins in xR applications usually involve the continuous collection, processing, and visualization of detailed operational and potentially sensitive data. Such dynamic data streams are vulnerable to unauthorized access, possibly leading to data breaches and compromised system integrity [9, 15]. Additionally, integrating xR applications and digital twins with additional (critical) industrial systems may increase the attack surface even more, making the systems more vulnerable to potential attacks. Even more potential threats can arise due to common configurability issues, ranging from (cross-)dependency issues over unwanted feature interactions to misconfiguring applications [27, 28, 47]. So, the more complex and configurable the system, the greater the attack surface and thus the number of possible attacks [28].

Lessons Learned. To address security challenges, a security engineering approach is recommended, i.e., integrating security into the development process as phase between domain and application engineering [26]. For real-time digital twin-assisted xR applications, implementing a layered security architecture with dynamic encryption and authentication protocols helps mitigate dynamic risks. In non-real-time applications, securing stored data with encryption, access controls (e.g., account systems), and isolating particularly sensitive data is key. Effective dependency and configuration handling as well as defensive configuring prevent configuring issues and helps maintaining system confidentiality, integrity, and availability in different scenarios [35]. In our case, performing not only feature-/product-based verification (i.e., testing), but also family-based verification was useful for addressing issues that originate from the xR application family (e.g., core assets). In addition, isolating (i.e., modularizing) essential features and their source code under consideration of information hiding and optional encryption may reduce the attack surface and associated privacy risks.

4 Conclusion

In this paper, we shared our experiences in developing digital twin-assisted xR applications for industrial environments and efficiently handling their variability. We presented challenges and associated lessons learned, including platform specifics, immersion and interaction, digital twin-related data fragmentation, accessibility, and security. Configurability of xR applications is a key requirement to efficiently develop related variants, which, however, can lead to various issues to be addressed. Several additional research directions arise, taking into account the unique properties of digital twins, xR applications, and industrial environments. For instance, developing guidelines based on software product lines for handling configurability, enhancing data integration and synchronization across platforms, investigating influences related to no-code / low-code, or strengthening security measures tailored to occurring feature interactions and (cross)configurations.

Acknowledgments

The presented work was part of the project “Digital Teaching and Learning in Augmented Realities (DigiLehR)” (FKZ: FMM2020-15 63-Hochschule Harz) and funded by the Foundation for Innovation in Higher Education (https://stiftung-hochschullehre.de/).

References

[1] S. Adler and E. Bayrhammer. 2019. Engineering model linking and ontology linking for production. In *European Conference on Smart Objects, Systems and Technologies (Smart SysTech)*. 1–6.

[2] S. Alizadehsalehi and I. Yitmen. 2023. Digital twin-based progress monitoring management model through reality capture to extended reality technologies (DRX). *Smart and Sustainable Built Environment* 12, 1 (2023), 200–236.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-oriented software product lines*. Springer.

[4] M. Attaran and B. G. Celik. 2023. Digital twin: Benefits, use cases, challenges, and opportunities. *Decision Analytics Journal* 6 (2023), 100165.

[5] M. Billinghurst. 2021. Grand challenges for augmented reality. *Frontiers in Virtual Reality* 2 (2021), 578080.

[6] P. Cairns, A. Cox, and A. I. Nordin. 2014. Immersion in digital games: Review of gaming experience research. *Handbook of Digital Games* (2014), 337–361.

[7] D. Calandra, F. G. Pratticò, A. Cannavò, C. Casetti, and F. Lamberti. 2022. Digital twin-and extended reality-based telepresence for collaborative robot programming in the 6G perspective. *Digital Communications and Networks* (2022), 315–327.

[8] T. Castro, L. Teixeira, V. Alves, S. Apel, M. Cordy, and R. Gheyi. 2021. A Formal framework of software product line analyses. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 1–37.

[9] A. J. G. de Azambuja, T. Giese, K. Schützer, R. Anderl, B. Schleich, and V. R. Almeida. 2024. Digital twins in Industry 4.0 — Opportunities and challenges related to cyber security. *Procedia CIRP* 121 (2024), 25–30.

[10] A. C. C. dos Santos, M. E. Delamaro, and F. L. S. Nunes. 2013. The relationship between requirements engineering and virtual reality systems: A systematic literature review. In *Symposium on Virtual and Augmented Reality*. IEEE, 53–62.

[11] M. Eisenträger, S. Adler, and E. Fischer. 2019. Rethinking software development for collaboration technologies. In *International Conference of Engineering, Technology, and Innovation (ICE/IEE ITMC)*. 1–9.

[12] F. A. Fernandes and C. M. L. Werner. 2022. A scoping review of the metaverse for software engineering education: Overview, challenges, and opportunities. *PRESENCE: Virtual and Augmented Reality* 31 (2022), 107–146.

[13] F. E. Fernandes and C. M. L. Werner. 2022. Software product line for metaverse: Preliminary results. In *Smartworld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles (SmartWorld/UIC/ScalCom/DigitalTwin/PriComp/Meta)*. IEEE, 2413–2420.

[14] G. Fortino and C. Savaglio. 2023. Integration of digital twins & internet of things. In *The Digital Twin*. Springer, 205–225.

[15] S. Guikema and R. Flage. 2024. Digital twins as a security risk. *Perspective* 121 (2024), 1–5.

[16] Z. Han, Y. Li, M. Yang, Q. Yuan, L. Ba, and E. Xu. 2020. Digital twin-driven 3D visualization monitoring and traceability system for general parts in continuous casting machine. *Journal of Advanced Mechanical Design, Systems, and Manufacturing* 14, 7 (2020), 1–15.

[17] M. A. Hazrat, N. M. S. Hassan, A. A. Chowdhury, M. G. Rasul, and B. A. Taylor. 2023. Developing a skilled workforce for future industry demand: The potential of digital twin-based teaching and learning practices in engineering education. *Sustainability* 15, 23 (2023), 16433.

[18] R. Hunicke, M. Leblanc, and R. Zubek. 2004. MDA: A formal approach to game design and game research. *AAAI Workshop - Technical Report* 1 (01 2004).

[19] T. Kaarlela, S. Pieskä, and T. Pitkäaho. 2020. Digital twin and virtual reality for safety training. In *International Conference on Cognitive Infocommunications (CogInfoCom)*. IEEE, 115–120.

[20] H. M. Kamdjou, D. Baudry, V. Havard, and S. Ouchani. 2024. Resource-Constrained eXtended reality operated with digital twin in industrial Internet of Things. *Open Journal of the Communications Society* (2024).

[21] E. Kuiter, A. Knüppel, T. Bordis, T. Runge, and I. Schaefer. 2022. Verification strategies for feature-oriented software product lines. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 1–9.

[22] V. Kuts, T. Otto, E. G. Caldarola, G. E. Modoni, and M. Sacco. 2018. Enabling the teaching factory leveraging a virtual reality system based on the Digital Twin. In *EuroVR Conference*. VTT Technology.

[23] A. Liljaniemi and H. Paavilainen. 2020. Using digital twin technology in engineering education–course concept to explore benefits and barriers. *Open Engineering* 10, 1 (2020), 377–385.

[24] S. Malakuti. 2021. Emerging technical debt in digital twin systems. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 01–04.

[25] A. Martínez-Gutiérrez, J. Díez-González, P. Verde, and H. Perez. 2023. Convergence of virtual reality and digital twin technologies to enhance digital operators' training in industry 4.0. *International Journal of Human-Computer Studies* 180 (2023), 103136.

[26] R. May, C. Biermann, A. Kenner, J. Krüger, and T. Leich. 2023. A product-line-engineering framework for secure enterprise-resource-planning systems. In *International Conference on ENTERprise Information Systems*. Elsevier, 1–8.

[27] R. May, C. Biermann, J. Krüger, and T. Leich. 2025. Asking security practitioners: Did you find the vulnerable (mis)configuration?. In *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 1–10.

[28] R. May, C. Biermann, X. M. Zerweck, K. Ludwig, J. Krüger, and T. Leich. 2024. Vulnerably (mis)configured? Exploring 10 years of developers' Q&As on Stack Overflow. In *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 112–122.

[29] R. May and K. Denecke. 2024. Conversational agents in healthcare: A variability perspective. In *Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 123–128.

[30] R. May., T. Niemand., P. Scholz., and T. Leich. 2023. Design patterns for monitoring and prediction machine learning systems: Systematic literature review and cluster analysis. In *International Conference on Software Technologies (ICSOFT)*. SciTePress, 209–216.

[31] K. Meixner, K. Feichtinger, H. S. Fadhlillah, S. Greiner, H. Marcher, R. Rabiser, and S. Biffl. 2024. Variability modeling of products, processes, and resources in cyber–physical production systems engineering. *Journal of Systems and Software* 211 (2024), 112007.

[32] M. Nieke, C. Seidl, and S. Schuster. 2016. Guaranteeing configuration validity in evolving software product lines. In *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 73–80.

[33] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schuhmayer. 2021. Evolution in dynamic software product lines. *Journal of Software: Evolution and Process* 33, 2 (2021), e2293.

[34] E. M. Raybourn, W. A. Stubblefield, M. Trumbo, A. Jones, J. Whetzel, and N. Fabian. 2019. Information design for xr immersive environments: Challenges and opportunities. In *International Conference on Virtual, Augmented and Mixed Reality. Multimodal Interaction (VAMR)*. Springer, 153–164.

[35] S. Samonas and D. Coss. 2014. The CIA strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security* 10, 3 (2014).

[36] G. Schuh, M. Riesener, A. Gützlaff, C. Dölle, S. Schmitz, J. Ays, S. Wlecke, J. Tittel, and Y. Liu. 2022. Industry 4.0: Agile development and production with internet of production. In *Handbook Industry 4.0: Law, Technology, Society*. Springer, 367–390.

[37] G. Schuh, P. Scholz, T. Leich, and R. May. 2020. Identifying and analyzing data model requirements and technology potentials of machine learning systems in the manufacturing industry of the future. In *ITM)*. IEEE, 1–10.

[38] G. Schuh, P. Scholz, and M. Nadicksbernd. 2020. Identification and characterization of challenges in the future of manufacturing for the application of machine Learning. In *International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS)*. IEEE, 1–10.

[39] S. M. E. Sepasgozar. 2020. Digital twin and web-based virtual gaming technologies for online education: A case of construction management and engineering. *Applied Sciences* 10, 13 (2020), 4678.

[40] M. Singh, E. Fuenmayor, E. P. Hinchy, Y. Qiao, N. Murray, and D. Devine. 2021. Digital twin: Origin to future. *Applied System Innovation* 4, 2 (2021), 36.

[41] D. Stone, C. Jarrett, M. Woodroffe, and S. Minocha. 2005. *User interface design and evaluation*. Elsevier.

[42] F. Tao, B. Xiao, Q. Qi, J. Cheng, and P. Ji. 2022. Digital twin modeling. *Journal of Manufacturing Systems* 64 (2022), 372–389.

[43] X. Tu, J. Autiosalo, R. Ala-Laurinaho, C. Yang, P. Salminen, and K. Tammi. 2023. TwinXR: Method for using digital twin descriptions in industrial eXtended reality applications. *Frontiers in Virtual Reality* 4 (2023), 1019080.

[44] M. P. Uysal and A. E. Mergen. 2021. Smart manufacturing in intelligent digital mesh: Integration of enterprise architecture and software product line engineering. *Journal of Industrial Information Integration* 22 (2021), 100202.

[45] E. Van Der Horn and S. Mahadevan. 2021. Digital twin: Generalization, characterization and implementation. *Decision Support Systems* 145 (2021), 113524.

[46] P. Weichbroth. 2020. Usability of mobile applications: a systematic literature study. *IEEE Access* 8 (2020), 55563–55577.

[47] T. Xu and Y. Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *Comput. Surveys* 47, 4 (2015), 1–41.

[48] C. Yang, X. Tu, J. Autiosalo, R. Ala-Laurinaho, J. Mattila, P. Salminen, and K. Tammi. 2022. Extended reality application framework for a digital-twin-based smart crane. *Applied Sciences* 12, 12 (2022), 6030.

[49] Y. Yin, P. Zheng, C. Li, and L. Wang. 2023. A state-of-the-art survey on augmented reality-assisted digital twin for futuristic human-centric industry transformation. *Robotics and Computer-Integrated Manufacturing* 81 (2023), 102515.

[50] T. Zhan, K. Yin, J. Xiong, Z. He, and S.-T. Wu. 2020. Augmented reality and virtual reality displays: Perspectives and challenges. *iScience* 23, 8 (2020).

[51] H. Zhang, Q. Yan, and Z. Wen. 2020. Information modeling for cyber-physical production system based on digital twin and AutomationML. *The International Journal of Advanced Manufacturing Technology* 107, 3 (2020), 1927–1945.

UVL.js: Experiences on using UVL in the JavaScript Ecosystem

Victor Lamas
CITIC Research Center, Database Lab,
Universidade da Coruña
A Coruña, Spain
victor.lamas@udc.es

Maria-Isabel Limaylla-Lunarejo
CITIC Research Center, Database Lab,
Universidade da Coruña
A Coruña, Spain
maria.limaylla@udc.es

Miguel R. Luaces
CITIC Research Center, Database Lab,
Universidade da Coruña
A Coruña, Spain
miguel.luaces@udc.es

David Romero-Organvidez
I3US, University of Seville
Seville, Spain
drorganvidez@us.es

José A. Galindo
I3US, University of Seville
Seville, Spain
jagalindo@us.es

David Benavides
I3US, University of Seville
Seville, Spain
benavides@us.es

Abstract

The Universal Variability Language (UVL) was developed as a community-driven effort to create a simple yet extensible language for feature modeling, promoting tool interoperability within the software product line community. Although UVL is supported by several tools like FeatureIDE, Flamapy, and Pure::variants, it currently lacks direct support for web environments. To address this, we introduce a JavaScript-based UVL parser built with the ANTLR framework. This parser makes UVL models accessible directly within browser-based environments, eliminating the need for extra installations and enhancing UVL's usability for web-based tools. Furthermore, the parser can be used in back-end environments with JavaScript runtime environments such as Node.js. The parser has been successfully tested with more than 1,000 UVL models available on UVLHub and supports various UVL language levels and conversion strategies. We demonstrate its integration through two use cases: UVLHub, a public repository for UVL models developed using open science principles, and an application lifecycle management tool for software product lines. This JavaScript UVL parser is the first of its kind, unlocking new possibilities for web and JavaScript applications to take advantage of the advancements in UVL technology.

CCS Concepts

• **Software and its engineering → Software product lines**; **Reusability**; **Software libraries and repositories**; *Abstraction, Modeling and Modularity*; *Context specific languages.*

ACM Reference Format:
Victor Lamas, Maria-Isabel Limaylla-Lunarejo, Miguel R. Luaces, David Romero-Organvidez, José A. Galindo, and David Benavides. 2025. UVL.js: Experiences on using UVL in the JavaScript Ecosystem. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3715340.3715437

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715437

1 Introduction

The Universal Variability Language (UVL), a textual notation for variability models, was recently developed due to community effort [2]. The MODEVAR initiative [1] aims to achieve widespread adoption of a single language (UVL), as the variety of languages available limits tool interoperability [6]. To support this, the UVL parser [10] introduces an extension mechanism designed to handle different levels of complexity. It includes two main components: language levels, which define a simple core language with optional and more complex extensions, and conversion strategies that allow tools to translate between these levels. These strategies replace complex constructs with simpler and semantically equivalent expressions, enabling better tool interoperability. The parser allows tool developers to select the language level they support while automatically converting unsupported features.

JavaScript is a dynamic, high-level programming language that has become crucial for modern web development. Its primary role as a web application language is to allow developers to design responsive and interactive user interfaces directly within the browser. Its flexibility extends far beyond scripting on the client side. JavaScript has evolved into a powerful platform for server-side development, supporting a whole ecosystem of server-side components, especially with the introduction of runtime environments like Node.js. This includes frameworks for building services, REST APIs, and full-fledged back-ends. Because of its seamless integration with HTML and CSS, JavaScript is widely used and indispensable for creating intricate, feature-rich web applications that work on various platforms.

In this work, we enhance UVL's accessibility for web platforms by introducing a JavaScript-based parser for the language. Using JavaScript's widespread adoption and client-side runtime capabilities, the parser makes UVL accessible in web-based environments without requiring additional tooling or installations. This facilitates more straightforward integration with current web technologies and encourages wider adoption of UVL by enabling developers to process and work with variability models within web applications. The parser also supports UVL's language levels and conversion strategies to ensure compatibility with different toolchains.

Listing 1 shows an example in UVL format of a feature model to obtain different smartwatch configurations. There are two main blocks: features and constraints. Thanks to the indentations, we can create blocks that group the features under a common hierarchy. For example, in the mandatory block, at the first level of indentation, there are the features screen and energy management. In turn, the feature screen is of the alternative type, i.e., there is a choice between two types of screen: touch or standard. The constraints block allows the definition of restrictions between features, using the features' identifiers as a reference.

```
namespace smartwatch

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
```

Listing 1: Feature model example in UVL

2 JavaScript Parser for UVL

The UVL Parser [10] for Python and Java has already been published in a GitHub repository[1]. The repository encourages community collaboration by accepting contributions in other languages implementing UVL parsing. ANTLR is the foundation for the grammar parsing [7]. Our work involved adding new definitions to the base grammar already established for other languages in the repository, specifically incorporating elements such as opening and closing parentheses and brackets, considering the different features' depth. We then used the ANTLR4 JavaScript generator to generate the corresponding JavaScript classes based on UVL grammar automatically. These classes produce interfaces programmers can implement to process, read, and export the Abstract Syntax Tree (AST) from any UVL feature model definition. The parser initially passed 70% of the UVLs when we first tested it with all UVLs available on UVLHub. The errors were not due to our parser but rather the improper use of quotation marks in certain UVLs. We informed the UVLHub administrators so they could correct the issue. After they resolved it, we revalidated our parser with all UVLs from UVLHub, and it successfully passed their 1,515 feature models.

```
import { FeatureModel } from 'uvl-parser';
const featureModel = new FeatureModel('example.uvl');
const tree = featureModel.getFeatureModel();
```

Listing 2: Example usage of JavaScript UVL Parser

Listing 2 demonstrates a basic usage example. The process begins by importing the `FeatureModel` class. Next, an instance of the class is created, with the UVL file location passed as an argument to the constructor. The location of the UVL file is provided as an argument to the constructor, which creates an instance of the class. The constructor will directly parse the input text if the file is not found at the specified path. In this case, it will treat the parameter as if the UVL were provided in plain text rather than as a file path. The method `getFeatureModel()` within the `FeatureModel` class parses the UVL file into an AST. An error is raised if the file does not conform to the UVL grammar.

To further enhance the usability of this new parser, it has been added to the standard package manager for Node.js: the npm registry [2]. This allows developers to easily include the library as a dependency on their existing Node.js projects.

3 Integration into Web-based Tools

We integrated the parser into two distinct tools: UVLHub and SPLALM. The following sections describe these integrations.

3.1 UVLHub, an Online Repository for UVL Models

UVLHub is a repository for feature models in UVL format [9]. It adheres to open science principles, promoting the dissemination and sharing of knowledge [8]. The repository is integrated with Zenodo [3] to ensure the permanent storage of the models. This integration enables each model to receive a Digital Object Identifier (DOI) for easy citation. UVLHub also provides an REST API, allowing programmatic access to the models and facilitating their integration into other analysis tools. By unifying and standardizing access to feature models, UVLHub contributes to open science. It promotes transparency and replicability in software engineering variability research, enabling researchers to access and share data effectively.

Figure 1 shows the integration of uvl-parser with UVLHub. The uvl-parser package is available through the Node.js NPM manager. UVLHub maintains a package log within its package.json file, including the uvl-parser package. Due to the modular architecture of UVLHub, each module specifies how its JavaScript scripts should be compiled using Webpack. Webpack is a module bundler for JavaScript that combines and optimizes code files and resources—such as CSS, images, and scripts—into one or more files [3]. This process enhances development efficiency and improves the performance of web applications.

The parser integration with the actual JavaScript code has been done thanks to the snippet shown in Listing 2. The integration occurs on the client side to optimize resources. Whenever a UVL

[1] https://github.com/Universal-Variability-Language/uvl-parser

[2] https://npmjs.com/package/uvl-parser

[3] https://zenodo.org/

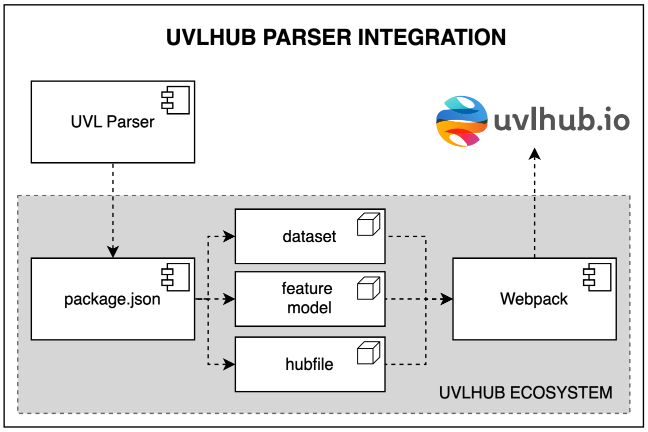
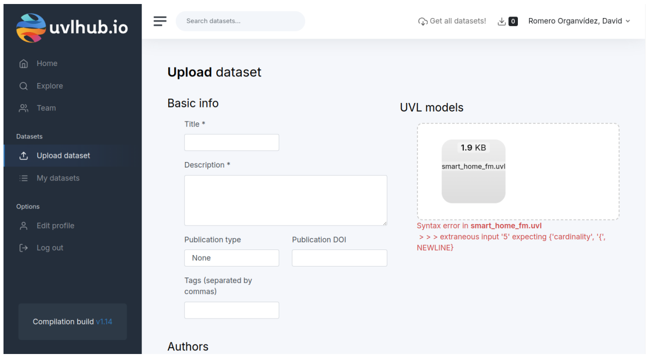


Figure 1: Integration of the UVL parser into UVLHub ecosystem



Upload dataset

Basic info

Title *

Description *

Publication type: None

Publication DOI

Tags (separated by commas)

Authors

UVL models

1.9 KB
smart_home_fm.uvl

Syntax error in **smart_home_fm.uvl**
> > > extraneous input '5' expecting {'cardinality', '{', NEWLINE}

Figure 2: Integration of the UVL parser into UVLHub graphic interface

file is uploaded, the system parses its content. If any syntax errors are detected, they are displayed in the interface, and the upload is canceled. This parser ensures that invalid UVL models are not uploaded. Figure 2 shows the visual interface of UVLHub and the error message that the uvl-parser throws when trying to upload a UVL model with syntactical errors. This integration is already available in the production version of UVLHub [4].

3.2 SPLALM, a Web-based PLE Factory

SPLALM [4] is an application lifecycle management tool designed for software product lines. SPLALM enables users to manage multiple software product lines, each characterized by its feature model in UVL and a collection of source code assets. Within SPLALM, users can also maintain a portfolio of products, with each product having its configuration and associated source code. SPLALM uses GitLab as a version control repository for product source code and product line assets. It implements a Git branching model to track configuration history effectively. SPLALM uses spl-js-engine [5], a JavaScript library created to produce source code for finished products using an annotative approach. This is accomplished by combining a product specification, a feature model for a product line, and annotated code. Spl-js-engine verifies the product specification against the feature model before producing the code.

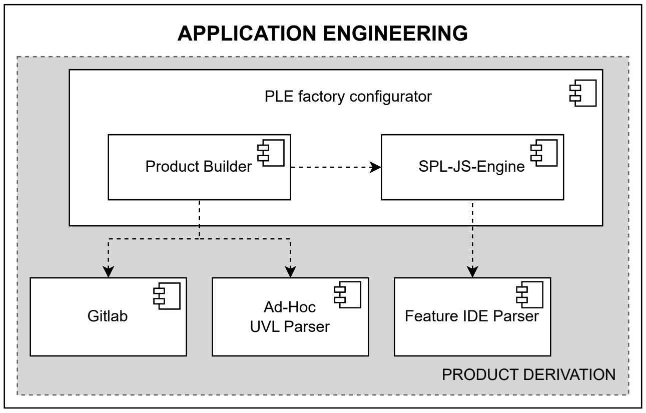


Figure 3: Architecture of SPLALM product derivation before UVL parser changes

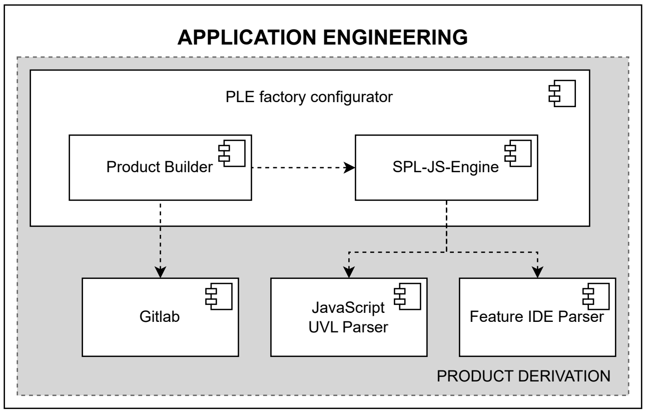


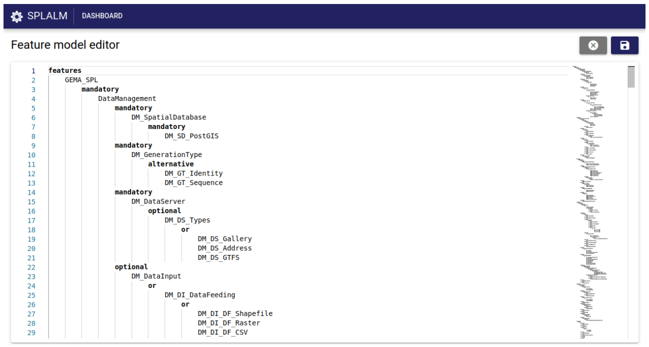
Figure 4: Architecture of SPLALM product derivation after UVL parser changes

In SPLALM, the product derivation component plays a critical role in managing and generating software products based on defined feature models. When providing a feature model for the SPL, SPLALM accepts both FeatureIDE files and UVL files. However, the product derivation component relied on a custom, ad-hoc UVL parser because the official UVL parser did not support JavaScript. Moreover, the component had to convert UVL files into the FeatureIDE language because it was the only format accepted by the spl-js-engine for code generation, as shown in Figure 3.

The introduction of a new, integrated JavaScript UVL parser marked a significant improvement. As shown in Figure 4, the adhoc UVL parser was removed, and the new JavaScript UVL parser was directly integrated into the spl-js-engine derivation engine. This integration allowed the derivation engine to autonomously determine which parser to use based on the input file format, whether FeatureIDE or UVL. As a result, the engine can now directly generate the Abstract Syntax Tree (AST) from the input, improving

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

the process of producing the final source code for the required products. This transition from a standalone ad-hoc UVL parser to an integrated parser within the derivation engine yielded several advantages. It eliminated an unnecessary processing step, reducing dependencies on external components. By integrating the UVL parser, the overall performance and simplicity of SPLAM were significantly improved. This change not only simplifies maintenance but also alleviates some technical debt. Figure 4 shows a screenshot of the Graphical User Interface of the UVL editor in SPLALM, which UVL.js supports.



```
features
    GEMA_SPL
        mandatory
            DataManagement
                mandatory
                    DM_SpatialDatabase
                        mandatory
                            DM_SD_PostGIS
                mandatory
                    DM_GenerationType
                        alternative
                            DM_GT_Identity
                            DM_GT_Sequence
                mandatory
                    DM_DataServer
                        optional
                            DM_DS_Types
                                or
                                    DM_DS_Gallery
                                    DM_DS_Address
                                    DM_DS_GTFS
                optional
                    DM_DataInput
                        or
                            DM_DI_DataFeeding
                                or
                                    DM_DI_DF_Shapefile
                                    DM_DI_DF_Raster
                                    DM_DI_DF_CSV
```

Figure 5: SPLALM Graphical User Interface: UVL editor

4 Conclusions and Future work

In this paper, we improved the accessibility and usability of the Universal Variability Language (UVL) in web-based environments by introducing a parser for UVL based on JavaScript. We showcased the parser's seamless integration into two web-based tools, utilizing JavaScript's widespread application in web development to encourage broader UVL adoption without requiring extra software installations. The parser supports UVL's language levels and conversion strategies, ensuring interoperability across different toolchains.

For future work, we propose developing more complex libraries around the parser to expand its capabilities and utility in diverse applications.

Material

We provide the code related to the paper. You can find the JavaScript version of the parser at this link: https://github.com/Universal-Variability-Language/uvl-parser, and the UVLHub code for integration at https://github.com/diverso-lab/uvlhub.

Acknowledgments

Partly funded by: TED2021-129245B-C21 (PLAGEMIS): partially funded by MCIN/AEI/10.13039/501100011033 and "NextGenerationEU"/PRTR; PID2022-141027NB-C21 (EarthDL): partially funded by MCIN/AEI/10.13039/501100011033 and EU/ERDF A way of making Europe; CITIC is funded by the Xunta de Galicia through the collaboration agreement between the Department of Culture, Education, Vocational Training and Universities and the Galician universities for the reinforcement of the research centers of the Galician University System (CIGUS). This work was also partially supported by FEDER/Ministry of Science, Innovation and Universities/Junta de Andalucía/State Research Agency/CDTI with the following grants: *Data-pl*(PID2022-138486OB-I00) , TASOVA PLUS research network (RED2022-134337-T), AquaIA (GOPG-SE-23-0011) and MIDAS (IDI-20230256). David Romero-Organvidez is supported by PREP2022-000335, financed by MICIN/AEI/10.13039/501100011033 and by FSE+.

References

[1] David Benavides, Rick Rabiser, Don Batory, and Mathieu Acher. 2019. First International Workshop on Languages for Modelling Variability (MODEVAR 2019). In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 323. doi:10.1145/3336294.3342364

[2] David Benavides, Chico Sundermann, Kevin Feichtinger, José A Galindo, Rick Rabiser, and Thomas Thüm. 2024. UVL: Feature Modelling with the Universal Variability Language. *Available at SSRN 4764657* (2024).

[3] Mohamed Bouzid. 2020. *Webpack for Beginners: Your Step-by-Step Guide to Learning Webpack 4*. doi:10.1007/978-1-4842-5896-5

[4] Alejandro Bujan, A. Cortiñas, and M. R. Luaces. 2024. Development of a PLE Factory Environment with GitLab Integration and following ISO/IEC 26580. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference (SPLC 2024)*. Luxemburgo, 34–37.

[5] A. Cortiñas, M. R. Luaces, and O. Pedreira. 2022. spl-js-engine: a JavaScript tool to implement Software Product Lines. In *Proceedings of the 26Th ACM International Systems And Software Product Line Conference (SPLC 2022)*. Graz, 66–69.

[6] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. 2023. FLAMA: A collaborative effort to build a new framework for the automated analysis of feature models. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, SPLC 2023, Tokyo, Japan, 28 August 2023- 1 September 2023*, Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Ivan Machado, Silvia Regina Vergilio, Rick Rabiser, Tao Yue, Xavier Devroey, Mónica Pinto, and Hironori Washizaki (Eds.). ACM, 16–19. doi:10.1145/3579028.3609008

[7] T. J. Parr and R. W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. doi:10.1002/spe.4380250705 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705

[8] Rahul Ramachandran, Kaylin Bugbee, and Kevin Murphy. 2021. From open data to open science. *Earth and Space Science* 8 (2021). doi:10.1029/2020EA001562

[9] David Romero-Organvidez, José A. Galindo, Chico Sundermann, Jose-Miguel Horcas, and David Benavides. 2024. UVLHub: A feature model data repository using UVL and open science principles. *Journal of Systems and Software* 216 (2024), 112150. doi:10.1016/j.jss.2024.112150

[10] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A. Galindo, and David Benavides. 2023. UVLParser: Extending UVL with Language Levels and Conversion Strategies. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B* (Tokyo, Japan) *(SPLC '23)*. Association for Computing Machinery, New York, NY, USA, 39–42. doi:10.1145/3579028.3609013

Delta-Oriented Multi-Level Configuration in Payment Gateway Product Lines

Sulthan Fathurrahman Tsany
Faculty of Computer Science
Universitas Indonesia
Depok, Jawa Barat, Indonesia
sulthan.fathurrahman@ui.ac.id

Rifqi Farel Muhammad
Faculty of Computer Science
Universitas Indonesia
Depok, Jawa Barat, Indonesia
rifqi.farel@ui.ac.id

Dhafin Raditya Juliawan
Faculty of Computer Science
Universitas Indonesia
Depok, Jawa Barat, Indonesia
dhafin.raditya@ui.ac.id

Maya Retno Ayu Setyautami
Faculty of Computer Science
Universitas Indonesia
Depok, Jawa Barat, Indonesia
mayaretno@cs.ui.ac.id

Ichlasul Affan
Faculty of Computer Science
Universitas Indonesia
Depok, Jawa Barat, Indonesia
ichlasul.affan12@cs.ui.ac.id

Abstract

Software product line engineering (SPLE) enables systematic reuse in developing various products within specific domains. Several approaches have been proposed to support SPLE implementation. In this paper, we report our experiences using delta-oriented programming (DOP) to implement a payment gateway product line. The implementation has specific challenges because the services of payment gateway are provided by many vendors. Even though they offer similar core services, each vendor often has distinct characteristics that must be accommodated in the product line application. The core services are modeled as similar features in the feature model, but each vendor's implementation might differ. Therefore, the feature selection must address vendor-specific configurations, requiring a multi-level configuration. This paper discusses the challenges and proposes solutions for achieving a multi-level configuration in the payment gateway domain. We also implement the solution as a tool support to enhance flexibility and promote applicability across other domains.

CCS Concepts

• **Software and its engineering** → *Integrated and visual development environments.*

Keywords

Delta-Oriented Programming, Multi-Level Configuration, Payment Gateway, Software Product Lines

ACM Reference Format:
Sulthan Fathurrahman Tsany, Rifqi Farel Muhammad, Dhafin Raditya Juliawan, Maya Retno Ayu Setyautami, and Ichlasul Affan. 2025. Delta-Oriented Multi-Level Configuration in Payment Gateway Product Lines. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715446

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715446

1 Introduction

A payment gateway is a technology that facilitates secure financial transactions between customers, merchants, and financial institutions. Payment gateway providers (vendors) offer similar core services for processing online payments, enabling merchants to securely transmit payment information. Additionally, they provide customizations, such as support for various payment methods, integration with different financial institutions, and compliance with diverse regulatory requirements across regions. However, managing variability in payment gateways presents several challenges due to the wide-ranging requirements, configurations, and features across different payment providers, merchants, and transaction types.

To address these challenges, Software product line engineering (SPLE) offers a compelling solution by enabling the systematic reuse of software components to produce diverse variants [3, 14]. This approach is particularly suited for domains that exhibit high variability, demand rapid adaptation to market shifts, and potential for reuse of core components across multiple products [1]. The payment gateway domain is well-suited for SPLE to meet varying customer needs while maintaining a shared core of functionality.

In this research, we implement a payment gateway software product line (SPL) using delta-oriented programming (DOP) approach [15]. The languages that support DOP are DeltaJ [10], abstract behavioral specification (ABS) [8], and variability modules for Java (VMJ) [17]. We choose VMJ to implement the payment gateway because it uses standard Java language. VMJ utilizes Java modules and design patterns (decorator and factory) to apply the principles of core and delta modules in Java.

The payment gateway SPL integrates services and relies on application programming interfaces (API) from various vendors. We refer to the domain analysis of the payment gateway product line conducted by [9] and explore public API documentation from several payment gateway vendors. We found that similar features in the feature diagram may have varying implementations across the vendors. Thus, product generation must consider both feature selection and vendor selection.

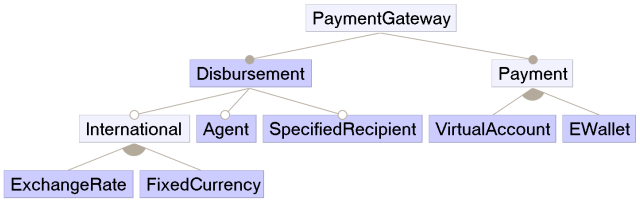


Figure 1: Feature diagram of the payment gateway, adopted from [9]

The multi-level configuration could support the feature selection, which depends on the vendors. It is a form of staged configuration with choices available to each stage [4]. The multi-level configuration is a process where system configuration choices are made in multiple stages, with each stage eliminating some configuration options [5]. Multi-level configuration is realized in some tools, such as V4rdiac for managing variability in cyber-physical production systems [6], ABS tool for managing microservices configurations of web-based SPL [16], Pure::variants tool to create multiple feature models that describe the overall system's variability [2].

We utilize a multi-level configuration approach to manage various variability in implementing the payment gateway SPL. For the payment gateway SPL, the user chooses the vendor in the first stage, and features are selected in the second stage. However, each vendor might also use distinct terminology to describe similar concepts. For instance, while vendor A uses the term *customer name*, vendor B uses *account holder name* to refer to a *client* using their services. The multi-level configuration is not enough because we have to define vendor-specific configurations to solve the different terminology problems.

This paper reports our experiences solving the challenges of implementing the payment gateway SPL using DOP. We propose the solution in the domain implementation and develop a tool to support multi-level configuration by extending FeatureIDE. Although we only present SPL payment gateway as a case study, the tool can be used to support a broad range of domains. Moreover, providing tools to support DOP with standard Java is essential to increase the adoption of this flexible variability implementation approach.

The paper is structured as follows. In Section 2, we propose the multi-level configuration approach for payment gateway SPL. Section 3 explains the usage of a Config *adapter* for unifying different external services from multiple vendors. In Section 4, we summarize insights and challenges as the lesson learned. Section 5 concludes the paper and discusses future work.

2 Multi-Level Configuration

We use the domain analysis conducted in [9] and the snippet of the feature diagram is shown in Figure 1. We explore the public API documentation from four payment gateway vendors: Flip [7], Xendit [19], Oy [13], and Midtrans [12]. Our implementation is based on real-world scenarios using actual API documentation from these vendors, ensuring practical applicability.

We identified an issue where the selection of *Payment* feature variants depends on the chosen vendor. For example, both Midtrans and Flip vendors offer similar payment method variants, such as *credit card*, *ewallet*, and *virtual account*. During the implementation, we connected to real APIs from the vendors, where each vendor provides distinct API services. Consequently, the generated application for the similar *virtual account* feature differs between Midtrans and Flip. To address this, we propose a multi-level configuration approach for payment gateway SPL, which divides product derivation into sequential stages. The first stage involves vendor configuration, while the subsequent stage allows the configuration of features constrained by the selected vendor.

We use FeatureIDE for feature modeling, domain implementation, and feature selection. The domain implementation is developed using VMJ [17]. We extend FeatureIDE by adding a new composer, called the WinVMJ Composer[1], which facilitates working with the VMJ language. The mapping between features and their Java module implementations is defined in a JavaScript Object Notation (JSON) file. This mapping file is structured as a dictionary, where keys represent feature variation names, and values contain lists of modules required for each feature variation. The product generation process uses a configuration file to define selected features. The WinVMJ Composer automatically generates, compiles, and runs valid product modules based on the specified configuration.

We propose two approaches for managing the multi-level configuration using FeatureIDE:

(1) Configuration File
A FeatureIDE `config` file contains a single `<configuration>` element encompassing a list of `<feature>` elements. While this structure effectively supports single-level configurations, it does not inherently support multi-level configuration processes. To enable the multi-level configuration, we propose extending the structure of the FeatureIDE `config` file, as shown in Listing 1. We introduce a new root element, `<multi-configuration>`, which wraps individual configurations. Each `<configuration>` element includes a `model` attribute that references the specific feature model file it corresponds to.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"
     ?>
<multi-configuration>
<configuration model="vendors.uvl">
<feature automatic="selected" name="Vendor"/>
<feature name="Flip"/>
...
</configuration>
<configuration model="paymentfeatures.uvl">
<feature automatic="selected" name="Gateway"/>
<feature name="Disbursement"/>
<feature name="Payment"/>
...
</configuration>
</multi-configuration>
```

Listing 1: Proposed Multi-level XML Configuration Format

[1] https://gitlab.com/RSE-Lab-Fasilkom-UI/PricesIDE/winvmj-composer

The current FeatureIDE project architecture is designed around a single root feature model per project. Implementing comprehensive support for multi-level configuration in the user interface would require substantial modifications to both the core and user interface functionality of FeatureIDE. To address this limitation, we develop a prototype Java program to manage multi-level configuration using the newly proposed Multi-level XML Format. The resulting configuration is then used for product generation in FeatureIDE with the WinVMJ Composer.

(2) Partial Configuration
We use an alternative approach to manage configurations at different levels. FeatureIDE supports deriving a subset SPL from partial configurations, allowing users to progressively narrow down their configuration choices based on prior selections [11]. This feature can be utilized to perform staged configurations for the payment gateway SPL.
At the first stage of configuration, the user selects the required vendors (e.g., Flip and Midtrans) and removes the rest (e.g., Oy and Xendit). Then, the user uses the partial configuration feature to derive a subset SPL based on the selected vendors. In the second stage, the available feature variants are restricted to only those provided by the vendors chosen in the previous stage. In this example, only the features supported by Flip and Midtrans will be included in the newly generated SPL. The user can then select from these available features, and the WinVMJ Composer generates the product variant.
When using the partial configuration approach, there is no strict order for feature selection. Users can select vendors and *Payment Gateway* features in any sequence. Therefore, this does not align with the intended configuration process, where vendor selection should precede feature selection. In future work, FeatureIDE should be extended to support multi-level configuration using Multi-level XML Format defined in the first approach.

3 Unifying Multiple Vendors

The payment gateway SPL is developed using the JAVA language that follows VMJ architectural pattern. VMJ is designed based on the DOP approach, so the implementation is divided into core and delta modules. The core modules implement the common parts, and the delta modules consist of variants' implementation. We analyze the common and variant parts of each feature from the vendor's API documentation. We use the UML diagram with UML-DOP profile (UML-DOP diagram) to model the common and variant parts of each feature.

The UML-DOP profile extends the UML metamodel to represent delta-oriented notation in the UML [18]. For illustration in this section, we use the *Disbursement*. A snippet of the UML-DOP diagram for *Disbursement* core module is shown in Figure 2. Based on the UML-DOP diagram, we develop the payment gateway SPL using the WinVMJ Composer in FeatureIDE. The implementation is available in this repository https://gitlab.com/RSE-Lab-Fasilkom-UI/PricesIDE/payment-gateway/vmj-payment-gateway.

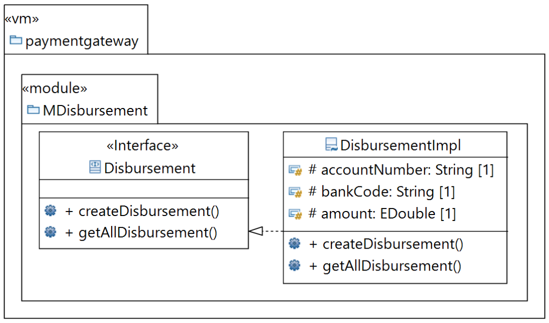


Figure 2: UML-DOP diagram of the *Disbursement* core module

We explore the API documentation to connect the core module to external services provided by vendors. We found that attributes for *Disbursement* features may have different names across different vendors, as illustrated in Table 1. Different vendors could have different terms that serve the same function. For example, there is *bank code* attribute in Figure 2 that represent a beneficiary's bank. Flip uses a similar term, but Xendit uses the term `channelCode` to represent a similar meaning.

Feature	Flip	Xendit
Disbursement	• amount • **bankCode** • accountNumber	• amount • **channelCode** • accountNumber

Table 1: Variant Attributes for the *Disbursement* in Flip and Xendit

Due to the different terms of each vendor, the multi-level configuration is insufficient to support product derivation in the payment gateway SPL. When we choose an attribute's name in the *core* module, we should have a map between this name to the actual attribute name for each vendor. For example, when a user chooses Xendit as the vendor, the Xendit API requires `channelCode` attribute instead of `bankCode`. This could lead to the creation of a new variability within the *Disbursement* feature, namely *Disbursement* and *Disbursement with Xendit*. To preserve *Disbursement* variability, we introduce a Config *adapter* to manage different terms across vendors.

The design of the Config *adapter* is as shown in Figure 3. When users select Flip as the vendor, the `DFlip` delta is applied to the core module. This selection is marked by the stereotype «when» on the abstraction line linking the `Flip` component to the `DFlip` delta module. Within the `ConfigImpl` class of the `DFlip` delta, all functions in the core module is modified to align with the requirements specified by the Flip vendor.

The Config *adapter* consists of functions required to manage the attributes needed for transactions with each vendor. During feature selection, the Config *adapter* is injected based on the vendor selection. This approach allows for the unification of multiple vendors without creating an additional level of variation in the *Payment* and *Disbursement* features.

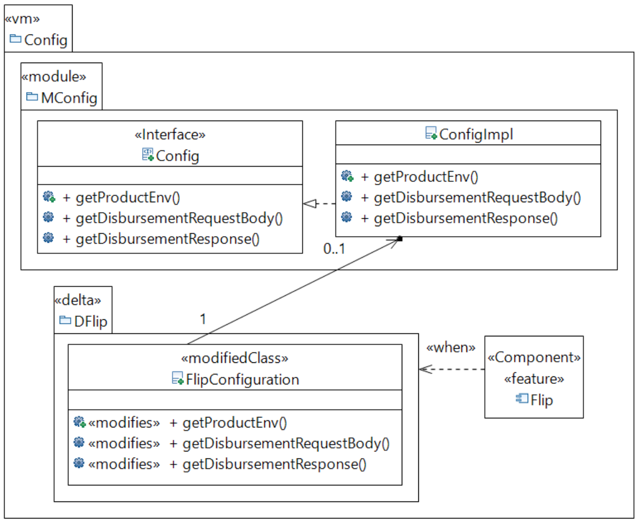

«vm»
Config
«module»
MConfig
«Interface»
Config
+ getProductEnv()
+ getDisbursementRequestBody()
+ getDisbursementResponse()
ConfigImpl
+ getProductEnv()
+ getDisbursementRequestBody()
+ getDisbursementResponse()
0..1
1
«delta»
DFlip
«modifiedClass»
FlipConfiguration
«modifies» + getProductEnv()
«modifies» + getDisbursementRequestBody()
«modifies» + getDisbursementResponse()
«when»
«Component»
«feature»
Flip

Figure 3: UML-DOP diagram of the Config *adapter*

The architecture of our solution is illustrated in Figure 4. As explained in Section 2, we extend the FeatureIDE with a new composer to support DOP implementation with VMJ. We develop a new composer based on VMJ to ease the adoption of the implementation approach using standard Java. The WinVMJ Composer consists of VMJ libraries that currently support web-based SPL development and a source compiler for the product generation. While we demonstrate the use of the WinVMJ Composer for the payment gateway implementation, this tool can be used to develop SPLs in any domain.

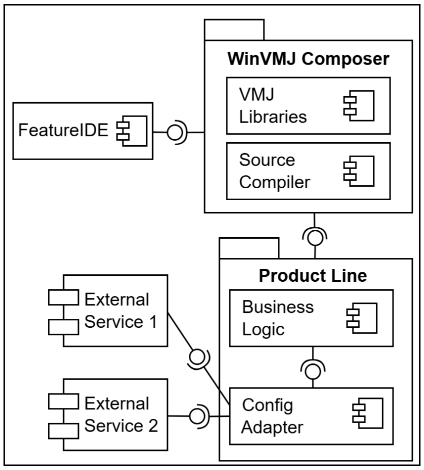


Figure 4: Solution Architecture

Based on the implementation of the payment gateway SPL, external API services use various terms for similar features. The feature's selection also depends on the chosen vendor, but the multi-level configuration is insufficient to address this challenge. This issue can arise in any domain that integrates multiple external services with varying terminologies. Our approach, which defines a Config *adapter*, is designed to support product line implementation that requires external services. As illustrated in Figure 4, the Config *adapter* is a part of product line implementation. The developer could extend the design of Config *adapter*, shown in Figure 3, to integrate with other external services in any domain.

4 Lesson Learned

This section highlights the lessons learned during the implementation of the payment gateway SPL using DOP. We discuss several valuable insights and notable challenges for future research and development.

We observed that SPLs offer great potential for integrating with external services, enabling enhanced flexibility and scalability. This capability allows developers to leverage the strengths of external APIs to create more dynamic and adaptable SPL. However, a recurring issue we identified in the payment gateway SPL is that external services frequently use unique naming conventions for their features.

Features that serve the same purpose across different external services often have different attributes, which complicates their integration. This variability requires the implementation of mechanisms to maintain feature consistency across different systems, ensuring seamless integration and configuration. Our solution is using a *Config* adapter to unify the variability across service providers.

We found that payment gateway APIs vary significantly across vendors, making multi-level configuration a valuable strategy for managing variability. We learned that FeatureIDE still requires extensions to support staged configurations. The partial configuration approach proved effective for narrowing down feature options progressively, but it also revealed gaps in enforcing configuration order. Extending the FeatureIDE configuration file revealed a critical need for adaptability to support complex SPLs.

Our experience with the payment gateway SPL closely simulates real-world scenarios, as we connect to actual APIs of existing payment gateway vendors. However, we did not encounter issues related to "removed elements" from the core modules. In VMJ, the delta *modifier* `remove` is implemented by throwing an exception in Java, as removing elements is not directly supported.

5 Conclusion and Future Work

We successfully implement the payment gateway SPL using DOP. We develop a tool to support multi-level configuration that divides product derivation into sequential stages. The first stage involves vendor configuration, while the second stage allows feature configurations constrained by the selected vendor. Based on our experience, multi-level configuration is insufficient to generate products based on a specific vendor. We design and implement the Config *adapter* to manage vendor-specific configurations.

We plan to improve the tool support by implementing cross-constraint evaluation between different feature models. FeatureIDE has a SAT solver to manage the cross-tree constraint. Furthermore, we plan to implement a support for multi-product line (MPL) in the tool. Therefore, the payment gateway SPL can be integrated to other SPLs.

Acknowledgments

This research is partially funded by Faculty of Computer Science, Universitas Indonesia. We thank Dr. Ade Azurat and Hafiyyan Sayyid Fadhlillah for their constructive feedback.

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer-Verlag, Berlin. https://doi.org/10.1007/978-3-642-37521-7

[2] Danilo Beuche. 2019. Industrial Variant Management with pure::variants. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 37–39. https://doi.org/10.1145/3307630.3342391

[3] Paul Clements and Linda M. Northrop. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA.

[4] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice* 10 (2005), 143–169.

[5] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice* 10 (04 2005), 143–169. https://doi.org/10.1002/spip.225

[6] Hafiyyan Sayyid Fadhlillah, Antonio M. Gutiérrez Fernández, Rick Rabiser, and Alois Zoitl. 2023. Managing Cyber-Physical Production Systems Variability using V4rdiac: Industrial Experiences. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A* (Tokyo, Japan) *(SPLC '23)*. Association for Computing Machinery, New York, NY, USA, 223–233. https://doi.org/10.1145/3579027.3608994

[7] Flip. 2024. *Flip API Documentation.* https://docs.flip.id Accessed: 2024-08-25.

[8] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 142–164.

[9] Erithiana Sisijoan Koesnadi, Maya R. A. Setyautami, and Ade Azurat. 2022. Domain Analysis of Payment Gateway Product Line. In *2022 International Conference on Information Technology Systems and Innovation (ICITSI)*. 237–244. https://doi.org/10.1109/ICITSI56531.2022.9971067

[10] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. 2014. DeltaJ 1.5: Delta-oriented Programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Cracow, Poland) *(PPPJ '14)*. ACM, New York, NY, USA, 63–74.

[11] Lukas Linsbauer, Paul Westphal, Paul Maximilian Bittner, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2022. Derivation of subset product lines in FeatureIDE. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 38–41. https://doi.org/10.1145/3503229.3547033

[12] Midtrans. 2024. *Midtrans API Documentation.* https://docs.midtrans.com/ Accessed: 2024-08-11.

[13] OY! 2024. *OY! API Documentation.* https://api-docs.oyindonesia.com/ Accessed: 2024-08-15.

[14] Klaus Pohl, Gunter Bockle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer-Verlag, Berlin. https://doi.org/10.1007/978-3-642-36583-6_1

[15] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–91.

[16] Maya R. A. Setyautami, Hafiyyan S. Fadhlillah, Daya Adianto, Ichlasul Affan, and Ade Azurat. 2020. Variability Management: Re-Engineering Microservices with Delta-Oriented Software Product Lines. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, Article 23, 6 pages. https://doi.org/10.1145/3382025.3414981

[17] Maya R. A. Setyautami and Reiner Hähnle. 2021. An Architectural Pattern to Realize Multi Software Product Lines in Java. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems* (Krems, Austria) *(VaMoS'21)*. Association for Computing Machinery, New York, NY, USA, Article 9, 9 pages. https://doi.org/10.1145/3442391.3442401

[18] Maya R. A. Setyautami, Reiner Hähnle, Radu Muschevici, and Ade Azurat. 2016. A UML Profile for Delta-oriented Programming to Support Software Product Line Engineering. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16)*. ACM, New York, NY, USA, 45–49.

[19] Xendit. 2024. *Xendit API Reference.* https://developers.xendit.co/api-reference Accessed: 2024-08-30.

UVL web-based editing and analysis with flamapy.ide

Francisco Sebastian Benitez
Universidad de Sevilla
Seville, Spain
fbenitez1@us.es

José A. Galindo
University of Seville
Sevile, Spain
jagalindo@us.es

David Romero Organvídez
Universidad de Sevilla
Seville, Spain
drorganvidez@us.es

David Benavides
University of Seville
Seville, Spain
benavides@us.es

Abstract

Feature modeling is widely used to represent variability in software systems, but as feature models grow in size and complexity, manual analysis becomes infeasible. Automated Analysis of Feature Models (AAFM) is a set of tools and algorithms that enable the computer-aided analysis of such models. Recently, the AAFM community has made an effort to enable the interoperability of tools by means of the UVL language, however, most of the supporting tools need to execute the operations in a server. This have two main drawbacks, first it requires users to upload the model to remote servers, imposing security concerns and second, limits the complexity of the operations that an online tool can offer. In this paper, we introduce `flamapy.ide` , an integrated development environment (IDE) based on the *flamapy* framework, and designed to perform AAFM directly within the browser by relying on WASM technologies. `flamapy.ide` provides SAT and BDD solvers for efficient feature model analysis and offers support for handling UVL files. Also, enables the configuration and visualization of such models relying on a fully client-side approach. This tool brings AAFM capabilities to web-based platforms, eliminating the need for server-side computation while ensuring ease of use and accessibility.

ACM Reference Format:
Francisco Sebastian Benitez, José A. Galindo, David Romero Organvídez, and David Benavides. 2025. UVL web-based editing and analysis with flamapy.ide. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715436

1 Introduction

Feature modeling has become the *de facto* standard for representing variability in software systems [7]. As variability models grow in complexity and size—such as in the Linux kernel [18] or Debian distributions [9]—manual analysis becomes unfeasible. To manage these large models, the only viable approach involves computer-aided techniques and algorithms known as Automated Analysis of Feature Models (AAFM).

Recently, the community have put efforts on trying to ease off the access to FMs [22] and increase the interoperability between models. First, the Universal variability language (UVL) [2] have been proposed to facilitate interoperability between tools and reduce fragmentation in the feature modeling ecosystem. The aim is to enable seamless export and import of feature models across different tools, both online and offline. Second, tools like Feature-IDE [15], `flamapy` [11], UVLS [17], and Travart [6] have adopted and supported this common standard.

Over the past 30 years [10], numerous AAFM tools have been developed, ranging from open-source options like Feature-IDE [15] to proprietary tools such as Gears [16] and pure::variants [3]. Additionally, web-based tools like Glencoe [24] and SPLOT [19] have emerged. A comprehensive review of these tools is available in [8, 13]. However, FMs are not just technical artifacts; they are essential business assets that outline both the current features of products and potential future additions. This introduces security concerns when using online AAFM tools, especially since feature models may contain sensitive strategic information for a company. Moreover, the high computational demands of certain AAFM operations [20] (i.e., find the number of configurations in a FM) have limited the availability of these on online platforms. At the same time, offline tools often require more complex installations requiring different software dependencies thus, making them less attractive to users.

Recently, there has been a shift towards using web browsers as platforms for fully client-side applications. This is, among other techniques, this is achieved by the introduction of WebAssembly (WASM) which allows the execution of high-performance applications directly in the browser offering near-native performance and a sandboxed environment. Modern browsers are now capable of running complex code, including solvers such as pysat [14] and Python-based applications like those enabled by Pyodide [5].

In this paper, we introduce `flamapy.ide` , an Integrated Development Environment (IDE) based on the `flamapy` framework [11], which enables AAFM directly in the browser by means of WASM techniques. The tool integrates both SAT and BDD solvers and supports working with UVL files. Additionally, it provides features for visualizing feature trees in both graphical and textual formats.

Currently `flamapy.ide` provides the following features executed directly in the browser, eliminating the need to install software on the user's machine.:

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715436

(1) Support for UVL files, including code highlighting and error inspection.
(2) Retrieval of more than 10 metrics from FM structural properties and execution of 20 operations, which can be performed using either SAT or BDD solvers.
(3) Dynamic visualization of UVL models using the D3 visualization library.
(4) Exportation of models to different formats, such as Glencoe or SPLOT.

The rest of the paper is structured as follows: Section 2 introduces the background for this research. Then, the architecture of `flamapy.ide` is presented in Section 3. Finally, conclusions and future work are presented in Section 5.

2 Background

flamapy [11] is both a tool and a framework written in Python[1] that offers full support for the UVL [2] feature model serialization. Currently it provides more than 20 AAFM analysis operations. Internally its relying on a plugin-based framework heavily inspired in the MDD frameworks (e.g., modularity, ease of extending the framework). Currently it offers off-the-shelve solvers based analysis using the Pysat [14] and CUDD [25] solvers. Those Python solvers provide either bindings to C++ solvers or direct Python implementations. Also, `flamapy` offers easy-to-use interfaces such as a command line, a FAMILIAR [1] inspired Python interface and a REST API.

WebAssembly (WASM) and Python. WASM is a binary instruction format designed for high-performance execution in web browsers and other environments. It enables running code written in multiple languages, such as C, C++, and Rust, at near-native speed on web platforms. It offers a low-level bytecode that can be executed by the JavaScript engine present in most modern browsers. It is executed in a secure sandboxed environment, ensuring safety in web applications. Beyond browsers, WASM is increasingly used in other platforms, providing a lightweight, cross-platform runtime environment.

Pyodide[2] is a project that brings the Python runtime to the web by compiling it to WebAssembly (WASM), allowing Python code to execute directly within the browser. It can load any pure Python package from the Python Package Index (PyPI). For `flamapy.ide` , Pyodide enables seamless client-side execution of `flamapy` without requiring users to install additional software.

3 Architecture

`flamapy.ide` is a web application that runs entirely on the client side. This means that the various operations it provides (analysis, validation, import, export, visualization, etc.) are performed without relying on a connection to an external server for execution.

To achieve this,`flamapy.ide` relies on the architecture presented in Figure 1. In this architecture, a server is emulated to handle client requests. To achieve this, `flamapy.ide` relies on Web Workers, a *JavaScript* technology that allows background tasks to be executed [23]. The main benefit about relying on Web Workers is that the application remains responsive and reactive when more than one execution threat. This is the case of AAFM applications as the execution time increases as the model size grows, which would cause the user interface to freeze while waiting for those operations to complete. This is the reason why `flamapy.ide` encapsulate `flamapy` inside a Web Worker so a dedicated thread is created to work on the model at the request of the main thread, which manages the user interface and its interactions.

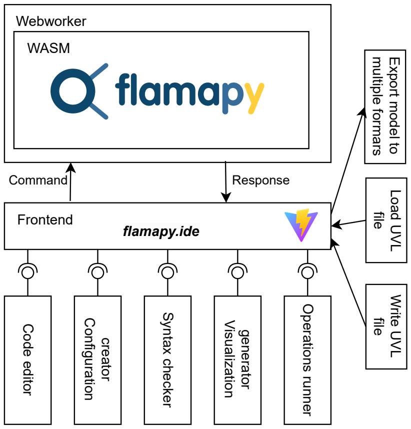


Figure 1: Architecture diagram

Nonetheless, `flamapy.ide` needs to create a user interface to be able to create the components of the IDE. For this purpose, `flamapy.ide` relies on the Vite[3] framework which enable the creation of reusable components at the frontend level.

3.1 Front-end components

To offer a full IDE functionallity, `flamapy.ide` created the components developed shown in Figure 2. Concretely the components are:

(1) The configuration component: where the user can define a configuration for the model present in the code editor. This is important due the need of passing a configuration to certain operations such as Valid Configuration or to check if there is any wrong assignment of values [8].
(2) The operations component: this component presents the user the list of operations to be run. A comprehensive list of operations available in the tool is presented in Section 4
(3) The code editor: where the model is written, whose value is analyzed to validate if the model is well specified. Internally the component rely on the Monaco editor which is the default editor in VScode[4]. The rules for syntax highlighting have been extracted out of the UVLSentinel tool [21].
(4) The result and status component: where the user can see the results of the operations executed in the IDE.

[1] http://docs.flamapy.org
[2] https://pyodide.org/en/stable/
[3] https://vite.dev/
[4] https://microsoft.github.io/monaco-editor/

(5) The information component: where the user can view important information about the model, such as its features. It also show any errors found when parsing of the UVL file.
(6) The visualization component: Is it common that a user wants to visualize in a tree shared graph the feature model encoded in the UVL format. This component enables the visualization of a UVL model by generating a D3 based plot[5].

3.2 Interface and communication with `flamapy`

The user interface of the IDE is inspired on previous tools such as FeatureIDE [15] or Glencoe [24]. Figure 2 shows this interface.

When the UI is initialized. A *Web Worker* is simultaneously created, which runs in another thread. This *Web Worker* instantiates the `flamapy` framework using WASM and communicates with the main thread by exchanging messages. In this way, when the *Web Worker* receives a request, it processes it and performs a specific action, returning the result to the main thread through messages. Concretely, there are two types of messages in `flamapy.ide` :

- Command: this is the type of message sent by the main thread after a user action on the UI. The command and its content vary depending on the type of interaction generated by the user. For example, in the case of checking the satisfiability of the model, it will send the operation name, and the UVL file itself to `flamapy` . However, if we want to check the satisfiability of a configuration, both the model and the configuration will be sent.
- Result: this is the type of message sent by the *Web Worker* to the main thread after processing a command. Its content varies depending on the result obtained by `flamapy` . For example, it could be a boolean indicating if a model or a configuration are satisfiable or it can be the set of errors for a UVL model.

Through this threading architecture, which communicates via messages, it is ensured that all model analysis takes place within the user's own browser, eliminating the need for any external server.

Integrating it with other tools. `flamapy.ide` is prepared to handle requests from another tool. Currently, we have integrated it with the UVLHub [22] repository so its easier to load, analyze and evolve an UVL model. This is achieved by using a query parameter that includes the URL of the UVL file to import. For example the URL https://ide.flamapy.org/?import=https://www.uvlhub.io/hubfile/download/25.uvl will load an instance of `flamapy.ide` with the 25th model existing in the repository. Note that for this to be feasible CORS [4] should be either disabled or allow the exception.

3.3 Solvers

While the `flamapy` framework supports several solvers and enables the integration of other solvers through a plugin system, the default distribution includes two primary solvers: `Pysat` and `CUDD`. These solvers provide different approaches for analyzing feature models that can offer different performance depending on the model and user needs.

Pysat is a Python toolkit for working with Boolean satisfiability (SAT) problems. It is integrated into `flamapy` to perform SAT-based analysis of feature models, allowing for efficient solving of SAT problems and other operations such as detecting dead features, false optional features, and verifying the SAT of feature configurations. When executed in WASM, Pysat uses Glucose v3 as a solver.

CUDD(Colorado University Decision Diagram) is a C library that implements Binary Decision Diagrams (BDDs), which are used to represent Boolean functions. In `flamapy` , `CUDD` is leveraged to perform efficient reasoning over feature models, especially for operations like counting configurations, detecting core features, and determining feature inclusion probability. The BDD-based approach ensures that large feature models with numerous configurations can be analyzed.

The plugin-based architecture of `flamapy` allows researchers and developers to integrate other solvers seamlessly, whether they are SAT solvers, CSP solvers, or even custom algorithms. However, currently, the only two solvers within the WASM distribution are the ones presented above.

4 Operations and solvers

`flamapy.ide` relies on the `flamapy` tool. Concretely on an easy-to-use facade that it offers[6]. This facade is heavily inspired on the FAMILIAR tool [1] and offers a similar syntax for executing operations.

One of the main benefits of relying on this facade is to avoid the need of having to learn the `flamapy` framework underpinnings. This facade provides access to the different operations available in `flamapy` but taking into account if they rely on a solver or not. This is similar to what the FMFactLabel [12] tool does. In this manner, the operations that do not need a solver are called Metrics.

4.1 Operations

Below we present the operations currently supported by `flamapy.ide` . Table 1 show in which solvers are they available:

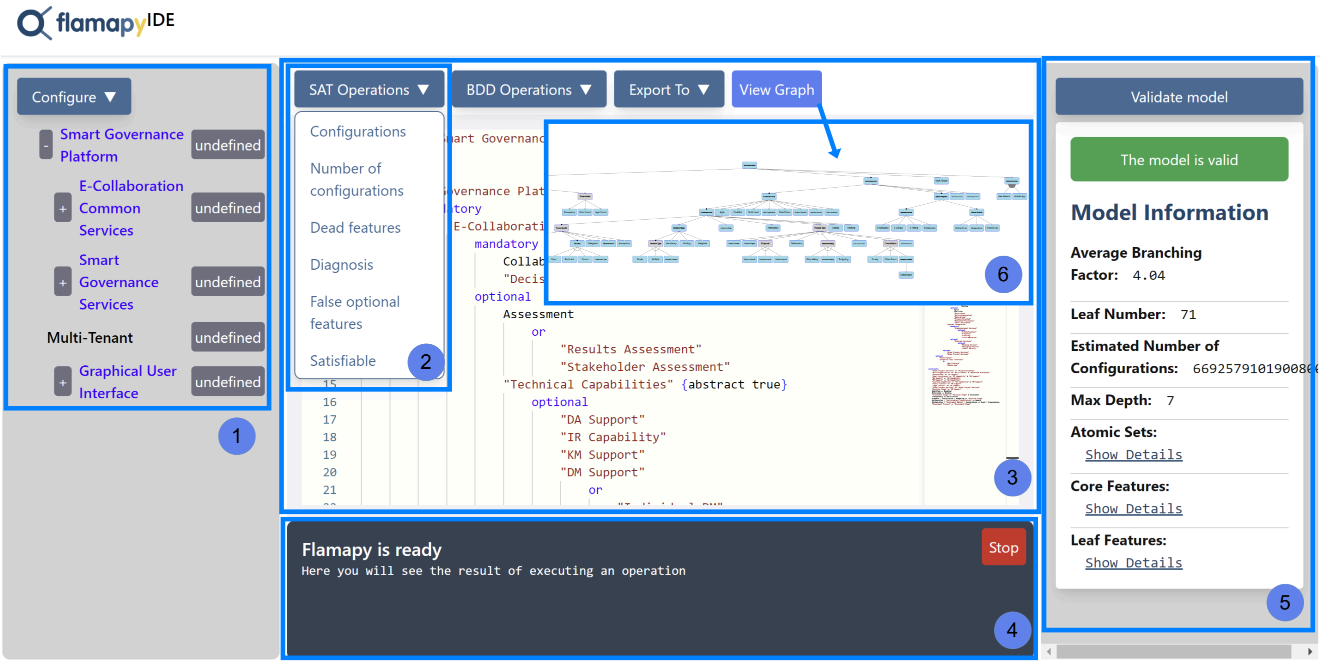
Table 1: Operations Supported by Pysat and CUDD

Operation name	Pysat	CUDD
Configurations	✓	✓
Number of Configurations	✓	✓
Dead Features	✓	✓
Diagnosis	✓	×
False Optional Features	✓	×
Satisfiable	✓	✓
Configuration Distribution	×	✓
Feature Inclusion Probability	×	✓
Unique Features	×	✓
Homogeneity	×	✓
Variability	×	✓
Variant Features	×	✓

The operations on feature models offered by `flamapy.ide` cover various aspects for managing variability: i) **Configurations**, representing valid feature selections conforming to all constraints; ii) **Number of Configurations**, indicating the total number of valid products derivable from the model, which quantifies the size

[5] https://d3js.org/

[6] https://docs.flamapy.org/tool/python_facade/



```
art Governanc
ivernance Plat
itory
E-Collaborati
  mandatory
    Collab
    "Decis
  optional
    Assessment
      or
        "Results Assessment"
        "Stakeholder Assessment"
    "Technical Capabilities" {abstract true}
      optional
        "DA Support"
        "IR Capability"
        "KM Support"
        "DM Support"
        or
```

Flamapy is ready

```
Here you will see the result of executing an operation
```

The model is valid

Model Information

Average Branching Factor: 4.04

Leaf Number: 71

Estimated Number of Configurations: 669257910190080

Max Depth: 7

Atomic Sets:

Core Features:

Leaf Features:

Figure 2: User interface components

of the configuration space; iii) **Dead Features**, features that cannot be part of any valid configuration; iv) **Diagnosis**, identifying inconsistencies or errors in the model to ensure correctness; v) **False Optional Features**, which behave like mandatory features despite being labeled as optional; vi) **Satisfiable**, determining if the model has at least one valid configuration; vii) **Configuration Distribution**, analyzing the frequency of different configurations to understand commonality; viii) **Feature Inclusion Probability**, assessing how likely a feature is included in a valid configuration; ix) **Unique Features**, which appear in only one configuration, indicating specialized setups; x) **Homogeneity**, measuring the similarity between configurations; xi) **Variability**, reflecting the extent of differences between configurations; and xii) **Variant Features**, those appearing in some configurations but not others.

4.2 Metrics

flamapy provides several key metrics to evaluate feature models. These metrics offer insights into FM structure, complexity, and behavior.

The analysis of feature models includes several important metrics: i) **Average Branching Factor**, which measures the average number of child features per feature. This could be used as an indicator of structural complexity; ii) **Leaf Number**, representing the total number of features without children; iii) **Estimated Number of Configurations**, an estimate of valid configurations derivable from the model. While not exact, it could help to understand the size of the variability encoded in a feature model; iv) **Max Depth**, the longest path from the root to any leaf v) **Atomic Sets**, groups of features that always appear together, simplifying the configuration; vi) **Core Features**, which are present in all valid configurations, representing mandatory functionality; and vii) **Leaf Features**, the most concrete decisions in the model, representing features without further refinement.

5 Conclusions and Future Work

In this paper, we introduced flamapy.ide , a web-based integrated development environment designed for AAFM leveraging WASM technologies. flamapy.ide eliminates the need for external servers, thus offering an accessible, platform-independent solution for feature model analysis. flamapy.ide demonstrates the feasibility and advantages of WASM in the context of AAFM enabling analysis within the browser. The tool's integration of metrics, solvers, and visualizations, all performed locally bypasses the common challenges of installation or backend reliance.

For future work, we plan to improve the user interface and extend support for additional feature model formats beyond UVL. Moreover by adding collaborative features, such as real-time coediting.

Material. You can find a deployment of the tool at https://ide.flamapy.org. The code of this tool is available at the Github Orghttps://github.com/flamapy.

Acknowledgments

This work was partially supported by FEDER/Ministry of Science, Innovation and Universities/Junta de Andalucía/State Research Agency/CDTI with the following grants: *Data-pl* (PID2022-138486OB-I00) , TASOVA PLUS research network (RED2022-134337-T), AquaIA (GOPG-SE-23-0011) and MIDAS (IDI-20230256). David Romero-Organvidez is supported by PREP2022-000335, financed by MICIN / AEI / 10.13039 / 501100011033 and by FSE+.

References

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2013. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming* 78, 6 (2013), 657–681.

[2] David Benavides, Chico Sundermann, Kevin Feichtinger, José A Galindo, Rick Rabiser, and Thomas Thüm. 2025. UVL: Feature Modelling with the Universal Variability Language. *Journal of systems and software* (2025).

[3] Danilo Beuche. 2008. Modeling and building software product lines with pure:: variants. In *Software Product Line Conference, International*. IEEE Computer Society, 358–358.

[4] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. 2018. We still {Don't} have secure {Cross-Domain} requests: An empirical study of {CORS}. In *27th USENIX Security Symposium (USENIX Security 18)*. 1079–1093.

[5] Michael Droettboom. 2019. Pyodide: Bringing the Scientific Python Stack to the Browser-Mozilla Hacks-the Web Developer Blog. *Mozilla Hacks–the Web developer blog* 16 (2019).

[6] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. 2021. Travart: An approach for transforming variability models. In *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–10.

[7] Alexander Felfernig, Andreas Falkner, and David Benavides. 2024. *Feature Models: AI-Driven Design, Analysis and Applications*. Springer Nature. https://doi.org/10.1007/978-3-031-61874-1

[8] Alexander Felfernig, Andreas A. Falkner, and David Benavides. 2024. *Feature Models - AI-Driven Design, Analysis and Applications*. Springer. https://doi.org/10.1007/978-3-031-61874-1

[9] José Galindo, David Benavides, and Sergio Segura. 2010. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In *ACoTA*. 29–34.

[10] José Angel Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[11] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. 2023. FLAMA: A collaborative effort to build a new framework for the automated analysis of feature models. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, SPLC 2023, Tokyo, Japan, 28 August 2023- 1 September 2023*, Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Ivan Machado, Silvia Regina Vergilio, Rick Rabiser, Tao Yue, Xavier Devroey, Mónica Pinto, and Hironori Washizaki (Eds.). ACM, 16–19. https://doi.org/10.1145/3579028.3609008

[12] José Miguel Horcas, José A. Galindo, Lidia Fuentes, and David Benavides. 2025. FM fact label. *Sci. Comput. Program.* 240 (2025), 103214. https://doi.org/10.1016/J.SCICO.2024.103214

[13] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023. Empirical analysis of the tool support for software product lines. *Softw. Syst. Model.* 22, 1 (2023), 377–414. https://doi.org/10.1007/S10270-022-01011-2

[14] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: A Python toolkit for prototyping with SAT oracles. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 428–437.

[15] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *2009 ieee 31st international conference on software engineering*. IEEE, 611–614.

[16] Charles W Krueger. 2008. The biglever software gears unified software product line engineering framework. In *2008 12th International Software Product Line Conference*. IEEE, 353–353.

[17] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. 2023. UVLS: A Language Server Protocol For UVL. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, SPLC 2023, Tokyo, Japan, 28 August 2023- 1 September 2023*, Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Ivan Machado, Silvia Regina Vergilio, Rick Rabiser, Tao Yue, Xavier Devroey, Mónica Pinto, and Hironori Washizaki (Eds.). ACM, 43–46. https://doi.org/10.1145/3579028.3609014

[18] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. 2010. Evolution of the Linux kernel variability model. *Software Product Lines: Going Beyond* (2010), 136–150.

[19] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 761–762.

[20] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. 231–240.

[21] David Romero-Organvidez, José A. Galindo, and David Benavides. 2024. UVL Sentinel: a tool for parsing and syntactic correction of UVL datasets. *CoRR* abs/2403.18482 (2024). https://doi.org/10.48550/ARXIV.2403.18482 arXiv:2403.18482

[22] David Romero-Organvidez, José A. Galindo, Chico Sundermann, José Miguel Horcas, and David Benavides. 2024. UVLHub: A feature model data repository using UVL and open science principles. *J. Syst. Softw.* 216 (2024), 112150. https://doi.org/10.1016/J.JSS.2024.112150

[23] Damodaran Chingleput Sathyakumar. 2024. Techniques and practices for optimizing resources in large scale horizontal web applications that deliver cross functional UX components. In *2024 IEEE International Conference on Electro Information Technology (eIT)*. IEEE, 468–479.

[24] Anna Schmitt, Christian Bettinger, and Georg Rock. 2018. Glencoe–a tool for specification, visualization and formal analysis of product lines. In *Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0*. IOS Press, 665–673.

[25] Fabio Somenzi. 2009. CUDD: CU decision diagram package-release 2.4. 0. *University of Colorado at Boulder* 21 (2009).

Behavioral Programming in the Large using Variability Management

Tom Felber
Technische Universität Dresden
Dresden, Germany
tom.felber@t-online.de

Sebastian Götz
Technische Universität Dresden
Dresden, Germany
sebastian.goetz1@tu-dresden.de

Abstract

Behavioral Programming is a paradigm that aims to enable incremental development by utilizing scenarios and use cases for actual software development. The overall behavior is formed by the composition of individual threads (*b-threads*) that run concurrently with each other and use an event-based mechanism to communicate and effect the system's state. However, with a rising number of threads, it gets progressively harder to comprehend state changes and system architecture, making design decisions more difficult. This holds in particular for context-aware programs, i.e., systems that need to react to contextual changes. In this paper, we propose the idea of employing feature modeling techniques from variability management to achieve an architectural representation of context-aware behavioral programs, enabling us to model the relations between behavioral threads and facilitating techniques from context-aware variability management. We illustrate multiple potential advantages and apply the approach to two examples from literature.

CCS Concepts

• **Software and its engineering → Software product lines**.

Keywords

Behavioral Programming, Dynamic Software Product Lines, Context Awareness

ACM Reference Format:
Tom Felber and Sebastian Götz. 2025. Behavioral Programming in the Large using Variability Management. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715443

1 Motivation

Behavioral Programming (BP) [16] is a software paradigm that utilizes the concept of scenarios for actual system development. The main building blocks are behavioral threads (b-threads) that run concurrently to each other, and each of them implements a single behavior the program should or should not follow. They use events at special synchronization points to exchange information and "vote" which event should be triggered. The selected event will then impose its effect on the system. Events can either be requested, blocked or waited-for. Only events that have been requested by at least one b-thread and have not been blocked by any of them are subject for selection. B-threads can wait for events to get notified upon their unfolding.

With ongoing development, the number of b-threads and events in the system can climb rapidly, which makes it hard to keep track of why the system exhibits certain behaviors and makes it easy for bugs to hide. We attempted to tackle this issue by proposing a novel debugger for BP that builds a runtime model of the behavioral system over time and offers analysis and control capabilities [10]. However, this approach only allows for an investigation at runtime, i.e., when bugs have already been introduced and shall be fixed.

In this paper, we aim to introduce an architecture description language (ADL) for BP, which allows for the identification of problems before runtime. Instead of just focusing on the individual b-threads (programming in the small), we aim to enable the developer to focus on their interplay (programming in the large). Due to the inherent variability of behavioral programs, we extend existing variability management techniques to enable the specification and analysis of behavioral programs.

B-threads are only loosely coupled, making it easy to reconfigure them at runtime. Therefore, BP could be particularly suited for the domain of *Dynamic Software Product Lines (DSPLs)* [14]. DSPLs allow users to switch between their variants while the system is running, allowing for adapting the behavior to changes in environment and context. A structured approach to add, remove and exchange b-threads based on contextual conditions has been introduced as *ContextBP* [9]. But, also in ContextBP no architectural description exists, leading to the same complexity problem.

To enable the specification of the architecture of a context-aware behavioral program, we propose to extend feature trees with the three types of events from BP for each feature (waited-for, requested, blocked), while each feature, which does not have sub-features, is mapped to a b-thread. The resulting feature tree provides a comprehensible representation of the application's architecture and enables to reason about feature interaction in terms of conflicting events. We use context-aware variability modelling [11, 17, 20, 22] as an inspiration for our architectural description language to include the specification of contextual conditions.

To exemplify this vision, we first summarize the state of the art of variability management with a particular focus on how it can help to enable an architectural description of behavioral programs. Then we present our vision of event-based b-thread feature trees. Finally, we discuss how well the proposed approach performs for two case studies: the water tank and the tic-tac-toe example from literature [16].

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715443

2 Behavioral Programming

Requirements engineering is a fundamental field in modern software development. Another technique is to define scenarios and use-cases prior to development [15]. *Behavioral Programming* [16] aims to streamline this process by using these scenarios for actual development. Behaviors are implemented individually and composed at runtime, producing a larger compound behavior.

Behavioral Threads. B-threads are the main building block of every behavioral program. Each of them describes a behavior the system should or should not follow. They are implemented as concurrently running threads and usually do not possess knowledge about other b-threads in the system.

Events. Events are the b-thread's way of communicating with other b-threads and, by this, allow them to effect the system state. They carry the behavior a b-thread describes. Each thread may request, block or wait for a set of events.

Synchronization Points. Every b-thread may reach a point in their control flow where they require information exchange or like to effect the system in a certain way. This point in time is the so-called synchronization point. One event is selected from all requested and non-blocked sets of events to unfold the behavior it carries. After the selection process completes, all threads that waited for this particular event are being notified of its arrival.

Contextual Behavioral Programming. In [9], Elyasaf extends the BP approach with the ability to specify the interplay between context and b-threads as well as contextual conditions for the activation of b-threads. The approach extends BP by a context data structure, an effect function that enables events to update the context, and context-aware b-threads which are b-threads bound to a context query. Context-aware b-threads contain b-threads that are spawned when the context query evaluates to true.

Example. A commonly used example for BP is the water tank example. The system state represents the amount and temperature of the water in a tank. B-threads can be used to add hot or cold water to the tank. To reach a desired amount or temperature of the water, a controlling b-thread blocks the hot or cold b-threads if the water is already too hot or too cold, respectively. In ContextBP, multiple tanks in different locations can be modelled, where the location is modelled as context information and individual live copies of the b-threads of the original water tank example are spawned for the respective locations.

3 State Of The Art

Comprehensibility of behavioral programs has not been investigated much. This holds in particular for context-aware BP. In contrast, context-aware (dynamic) software product lines (SPL) have been actively investigated in the last few decades. In this section, we first summarize existing work on BP comprehension, followed by the state of the art in context-aware SPLs.

Trace Visualization. TraceVis [25] is an application to visualize the execution of behavioral programs. It features an interactive, table-like display that contains information on b-threads that were active in the system. The state of a thread at a certain sync point can be read, containing all requested, blocked and waited-for events. A graphical differentiation between threads that advance and those that do not is also presented for each sync point. The tool has been developed for the Java version of BP in particular. The package creates an XML output for a run that includes all necessary trace information and can then be imported into TraceVis.

Debugging Behavioral Programs at Runtime. Recently, we proposed a runtime debugger for behavioral programs [10]. The debugger collects relevant data while the system is active and builds a model of the program's state over time by applying the *Models@run.time* approach [5, 6]. It provides runtime control capabilities such as halting the program, slowing it down or applying complex, conditional breakpoints. Using the derived model, the browser frontend can display the system state at each sync point as a table, highlighting which event was selected and also which events may have been selected to hint at potential bugs. The debugger allows users to look up information of past sync points, also called time-travelling [2]. By leveraging principles from model-based debugging [21], traces can be exported and imported, enabling their comparison to the current execution to display differences.

Context-aware Variability Modelling. A general overview is provided in [22]. Mens et al. identify three classes of approaches: two separate feature models, one feature model with subbranches and a single feature model entangling contextual and non-contextual features. In [17], Hartmann and Trew present a subbranch approach. They use a separate context variability model besides the feature model of the application, while both trees are combined by having the same root feature. Contextual conditions can then be expressed using cross-tree constraints. In [20], an approach is introduced that allows to model an SPL with contextual validity formulas for each feature. The context is described separately from the feature model. Another approach using two separate models was presented in [11]. Here, the feature model and the context feature model are combined using context rules expressed as logic formulas.

Software Product Lines for Distributed Systems. In [23], a feature model for heterogeneous distributed systems is proposed. It introduces categories to the model for different units of composition or middleware inherent to distributed systems. Dos Santos Soares et al. introduce a domain-specific SPL architecture for road traffic management systems in [8]. They model the relationships between system components and use a publish-subscribe middleware.

Research Gap. We found only two approaches that increase the comprehensibility of behavioral programs. Both require the BP to be implemented first, i.e., they provide means to inspect an existing BP. Approaches for other distributed systems have been proposed, too. However, to the best of our knowledge, none exist that enable the developer to model and analyze a BP before implementing it.

As BP features inherent variability, we believe that existing variability management techniques could provide a possible solution. This holds in particular for context-aware behavioral programs, as considerable work on context-aware variability modelling exists, and ContextBP (see sec. 2) does not solve the complexity problem. Thus, in this paper, we leverage context-aware feature modelling to enable the modelling and analysis of contextual BPs before their implementation.

4 A Feature Model for Behavioral Programs

In this section, we investigate the novel approach of mapping individual features to b-threads. We believe this approach will prove advantageous for the comprehensibility of large BPs. We discuss our theoretical findings while also laying emphasis on how the processes of feature interaction reasoning and product derivation would be affected.

4.1 Event-Based B-Thread Feature Trees

We propose to extend the concept of feature trees [19] with BP-specific terminology (fig. 1). In BP, b-threads are the main component of execution, instead of classes and objects in object-oriented programming. Therefore, we map a single b-thread to each childless feature in the model while preserving all feature-model-specific methodology, such as relationship types and cross-tree references. Context awareness can be modeled as well, e.g., by adding context information as feature attributes, as depicted in fig. 1.

As explained in sec. 2, b-threads may request, block or wait-for events. These events define the behavior the program will exhibit. For each b-thread-carrying feature, we attach its respective event sets as feature attributes. By introducing cross-tree references between these attributes, we are able to not only model relations between different features, but also between individual behaviors, including them into any reasoning that might happen on the model.

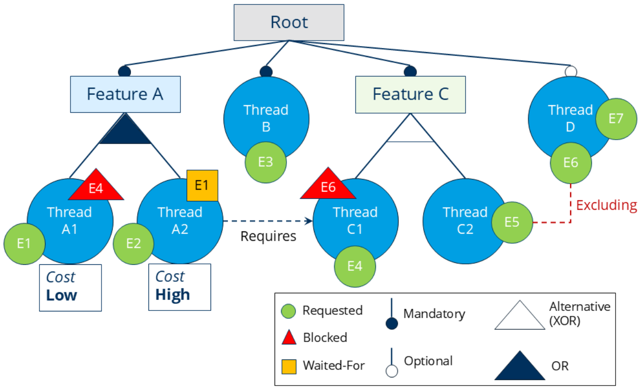


Figure 1: Feature tree with b-thread mapping, cross-tree constrains and context information.

4.2 Discussion

Advantages for BP Comprehension. With the application of feature models, we acquire a comprehensible representation of the program's architecture. State-of-the-art SPL techniques for model-checking and other operations [3, 13] can be reused, as well as conversions to other formats like the UVL [4, 27]. Finding potential issues should be easier, as should any system redesign that might prove to be necessary. Context sensitivity can also be directly included in the model.

In [10], we inserted an artificial bug into an exemplary program, which we then found using the proposed debugger. The bug involved two b-threads that requested one event each, but those events contradicted each other in their targeted behavior. One of them should have been blocked, but this was intentionally "forgotten". In our new approach, one could easily model these two b-threads and their respective events as mutually exclusive, either by certain relationships (like *Alternative*) or by using cross-tree constrains. Therefore, the developer could already see this potential issue by just looking at the feature model itself. Alternatively, a reasoner could detect the issue as well and prohibit the existence of both b-threads at the same time.

Detecting Feature Interactions. Known as "a major thread to feature modularity and compositional reasoning" [1], unwanted feature interactions are usually non-trivial to detect by just looking at each feature separately [26]. Therefore, the only possible way of finding all potential feature interactions is by generating all possible feature combinations. However, this poses an extremely expensive task, as the number of possible combinations (and therefore also possible interactions) rises exponentially with the number of features involved. In addition to measurable effects like behavior or resource consumption, some interactions may be invisible from the outside, like data or control flow, making source code access a requirement for further analysis.

For our proposed model extension, feature interaction analysis may work differently. We have access to all possible events each b-thread feature may generate, and therefore to all behavior and data flow that might occur. This implies that we can reason about possible interactions between features without generating any combination, just by looking at the model, possibly even without source code access. For a specific combination, we can discover, for example, which events may be selected, which may not and also if multiple contradicting events could be subject to being triggered at the same time, but without the need of deriving the product first.

Product Derivation. Product Derivation is the process of deriving the actual software product from selected features. Usually, the communication with all kinds of stakeholders, like customers, forms a large part [24]. This of course stays unchanged with BP, as the decision processes of which features to include have still to be performed. The technical part, however, how the features are selected and composed, is being influenced.

B-threads are only loosely coupled, making them a good instrument in the derivation process, especially for DSPLs, where derivation can happen dynamically during runtime. In [7], three problem dimensions are being identified: "When, How and What to configure". For the *When* dimension, BP offers the synchronization point as a point in time where a manipulation could occur. As all threads essentially halt, it is easy to remove, exchange them or add new ones entirely. The *How* dimension is also trivial, as b-threads run concurrently and independently next to each other, which offers a large degree of separation. Additional glue code that often has to be developed [1] is also not required. B-threads take care of any needed interaction by their own with their event sets.

In addition, our approach would not suffer under the *Optional Feature Problem* [18], where two domain-optional features are not independent in their codebase. The distance between problem and solution space in BP is comparatively small, originating from the use of domain-specific scenarios for actual implementation.

5 Examples

In this section, we validate our proposal on a theoretical basis using two examples from literature.

5.1 Tic-Tac-Toe

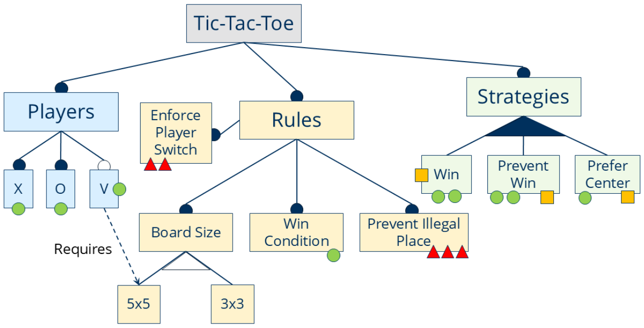


Figure 2: Feature tree for the Tic-Tac-Toe example.

In figure 2, we applied our proposed approach to the Tic-Tac-Toe game, a classic game that is already used as an example in the original BP paper [16]. We subdivide the model into three feature classes: Players, Rules and Strategies.

We imagine different players being actual b-threads in the system where each requests a player event that indicates which player's turn it is. Of course, we need to enforce turn switches which fits well in the Rules compartment from a domain point of view. Therefore, we may implement a b-thread that blocks all player events but the one that should be active right now.

On the right side of the diagram we can see the Strategies branch. B-threads in this category implement a certain strategy a player might use at a given moment in time. Our idea is that these threads wait for a player event that determines for which player they should calculate the next move and thereafter request appropriate move events. Strategy threads may completely ignore rules and request any move they seem fit. Rule threads then block any movement event that would violate the rule their taking care of. When extending strategies with priorities, the system will therefore automatically balance itself to apply the most optimal strategy that is in accordance with all rules in place. In case of a win, the win condition thread will request an event with a priority higher than any other, subsequently forcing the game to finish.

Tic-Tac-Toe is a quite simple game, so we may want to expand its board size, for example, and add new players. This can easily be modelled with the feature tree: We can add the board size as a feature and let the third player depend on the larger size. This prevents the existence of a third player if the board size is insufficient. However, we imagine the implementation of the board size as a b-thread not quite trivial, which could hint us at a potential design issue. Luckily, we are able to discover this problem from the model view alone, without needing actual implementation.

We hereby achieve an architectural view over the game that hints at design issues early on and allows extending the game with new rules, strategies and players.

5.2 Water Tank

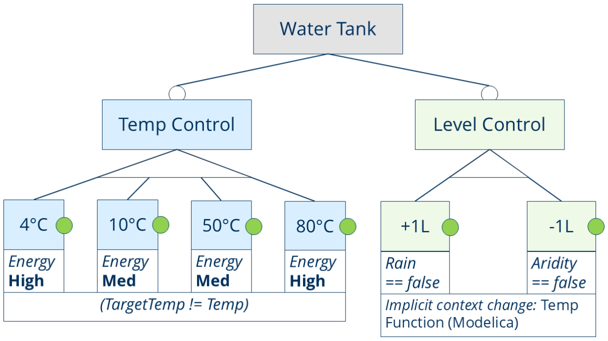


Figure 3: Feature tree for the Water Tank example.

In contrast to the example mentioned in 2, we treat temperature and level control separated here, as this makes more sense when applying the feature model (see figure 3).

The temperature control branch includes features that support a certain temperature. This makes sense on the background of BP, as b-threads and events work on a fine-grained level. There could be, for example, an event that heats with 80° C or cools with 4° C, similar how the original example was constructed. In contrast to Tic-Tac-Toe, the water tank is a cyber-physical system that dependents on some external context. The fine differentiation between heating and cooling temperatures allows to add energy requirements to these features. Naturally, a higher temperature requires more energy that may not be available at a given moment, allowing to deny the use of high temperature threads and falling back on a lower one. In addition, the target water temperature can also be modeled in the context, so that heating or cooling threads get deactivated once the target has been reached to save energy. That is the reason why there are no blocking or waited-for events in the model.

The level control category has context dependencies as well. Features here allow adding or removing water from the tank, depending on the humidity of the environment. If it rains, we may want to use rain water to fill the tank to save water and energy from the pump. Equally, if water is scarce because of aridity, we may want to forbid the removal of water entirely.

Pumping one liter of water into the tank will subsequently alter its temperature, leading to an implicit context change. This brought us to the idea of employing specific modelling techniques and languages, e.g. *Modelica* [12], to calculate this implicit context change, enabling the system to react in advance.

6 Conclusion and Future Work

Large Behavioral Programs suffer from a complexity issue, making system redesigns hard and bugs more likely. We propose to counter this problem by applying feature modeling techniques from Software Product Lines. By extending the classic feature tree approach with BP-specific terminology, we receive an Architecture Description Language for BP, allowing developers to analyze the system architecture before runtime. With two examples from literature, we show the theoretical feasibility of this approach. In future work, we will prototype and validate the approach in practice.

References

[1] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development* (Indianapolis, Indiana, USA) *(FOSD '13)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/2528265.2528267

[2] Earl T. Barr and Mark Marron. 2014. Tardis: affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &amp; Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/2660193.2660209

[3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information systems* 35, 6 (2010), 615–636.

[4] David Benavides, Chico Sundermann, Kevin Feichtinger, José A Galindo, Rick Rabiser, and Thomas Thüm. 2024. UVL: Feature Modelling with the Universal Variability Language. *Available at SSRN 4764657* (2024).

[5] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@ run. time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* 18 (2019), 3049–3082.

[6] Gordon Blair, Nelly Bencomo, and Robert B France. 2009. Models@ run. time. *Computer* 42, 10 (2009), 22–27.

[7] Jackson Raniel F. da Silva, Francisco Airton P. da Silva, Leandro M. do Nascimento, Dhiego A. O. Martins, and Vinicius C. Garcia. 2013. The dynamic aspects of product derivation in DSPL: A systematic literature review. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. 466–473. doi:10.1109/IRI.2013.6642507

[8] Michel dos Santos Soares, Jos Vrancken, and Yubin Wang. 2009. Software Product Line Architecture for Distributed Real-Time Systems. In *International Conference on Software Engineering Theory and Practice*. 8–15.

[9] Achiya Elyasaf. 2021. Context-Oriented Behavioral Programming. *Information and Software Technology* 133 (2021), 106504. doi:10.1016/j.infsof.2020.106504

[10] Tom Felber and Sebastian Götz. 2024. Debugging Behavioral Programs Using Models@run.time. In *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 160–167. doi:10.1109/SEAA64295.2024.00032

[11] Paula Fernandes, Cláudia ML Werner, and Eldânae Teixeira. 2011. An approach for feature modeling of context-aware software product Line. *Journal of Universal Computer Science* 17, 5 (2011), 807–829.

[12] Peter Fritzson. 2015. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons.

[13] José A Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101 (2019), 387–433.

[14] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic software product lines. *Computer* 41, 4 (2008), 93–95.

[15] David Harel and Rami Marelly. 2003. *Come, let's play: scenario-based programming using LSCs and the play-engine*. Vol. 1. Springer.

[16] David Harel, Assaf Marron, and Gera Weiss. 2012. Behavioral programming. *Commun. ACM* 55, 7 (2012), 90–100.

[17] Herman Hartmann and Tim Trew. 2008. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *2008 12th International Software Product Line Conference*. 12–21. doi:10.1109/SPLC.2008.15

[18] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. 2009. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*. 181–190.

[19] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. 2002. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *Software Reuse: Methods, Techniques, and Tools*, Cristina Gacek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 62–77.

[20] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context Aware Reconfiguration in Software Product Lines. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems* (Salvador, Brazil) *(VaMoS '16)*. Association for Computing Machinery, New York, NY, USA, 41–48. doi:10.1145/2866614.2866620

[21] Wolfgang Mayer and Markus Stumptner. 2007. Model-Based Debugging - State of the Art And Future Challenges. *Electronic Notes in Theoretical Computer Science* 174, 4 (2007), 61–82. doi:10.1016/j.entcs.2006.12.030 Proceedings of the Workshop on Verification and Debugging (V&D 2006).

[22] Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. 2017. Modeling and Managing Context-Aware Systems' Variability. *IEEE Software* 34, 6 (2017), 58–63. doi:10.1109/MS.2017.4121225

[23] Subhav Pradhan, Abhishek Dubey, William R Otte, Gabor Karsai, and Aniruddha Gokhale. 2015. Towards a product line of heterogeneous distributed applications. *ISIS* 15 (2015), 117.

[24] Rick Rabiser, Pádraig O'Leary, and Ita Richardson. 2011. Key activities for product derivation in software product lines. *Journal of Systems and Software* 84, 2 (2011), 285–300.

[25] Babooshka Shavazipour, Manuel López-Ibáñez, and Kaisa Miettinen. 2021. Visualizations for decision support in scenario-based multiobjective optimization. *Information Sciences* 578 (2021), 1–21.

[26] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology* 98 (2018), 44–58.

[27] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet another textual variability language? a community effort towards a unified language. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) *(SPLC '21)*. Association for Computing Machinery, New York, NY, USA, 136–147. doi:10.1145/3461001.3471145

Performance-Aware Behaviour Models for Feature-Dependent Runtime Attributes in Product Lines

Birte Friesel
Universität Osnabrück
Osnabrück, Germany
birte.friesel@uos.de

Olaf Spinczyk
Universität Osnabrück
Osnabrück, Germany
olaf@uos.de

Abstract

A product line's features describe its configurable functional properties by means of boolean feature toggles or numeric values. Meanwhile, performance models describe the link between features and non-functional product line properties such as binary size, memory usage, or workload latency. So far, these have focused on static features: model formalisms and learning algorithms assume that the product cannot be reconfigured at runtime and that there is no interaction between features, runtime behaviour, and runtime performance attributes. This is inadequate for real-world configurable software systems: compile-time features may be reconfigured at runtime (e.g. by overriding defaults), and non-featured workload attributes may affect performance attributes and thus must be considered within performance models. We propose performance-aware behaviour models to address this challenge. These build upon existing work on behaviour models and dynamic product lines, and link feature models (static configuration), behaviour models (variable workloads), non-featured runtime variability, and performance models in order to allow engineers to determine arbitrary configuration- and workload-dependent performance attributes of product lines. They are compatible with existing modeling methods and support workload-dependent performance attributes that cannot be expressed as a function of product features alone. We demonstrate the benefits of performance-aware behaviour models in three case studies, and show that they reduce performance prediction error by up to 90 %.

CCS Concepts

• **Software and its engineering → Extra-functional properties**; *Software verification and validation.*

Keywords

Software Product Lines, Non-Functional Properties, Behaviour Models, Performance Models, Annotated Featured Automata

ACM Reference Format:
Birte Friesel and Olaf Spinczyk. 2025. Performance-Aware Behaviour Models for Feature-Dependent Runtime Attributes in Product Lines. In *19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2025), February 04–06, 2025, Rennes, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3715340.3715435

VaMoS 2025, Rennes, France
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1441-2/25/02
https://doi.org/10.1145/3715340.3715435

1 Introduction

At its inception, product line engineering focused on static configuration: engineers can enable or disable individual features, and the resulting product either has or lacks the corresponding functional properties [25]. The field has since grown, with extensions such as numeric features [2], annotations for non-functional product properties (i.e., performance attributes) [4], contextual (runtime-specific) features [23], links between features and runtime behaviour [11], and time- or energy-related behaviour verification [10, 14, 20, 27].

Formalized knowledge about non-functional product properties is crucial for performance-aware product line configuration and optimization [1, 28]. In case a configurable software system provides several implementations that fulfil the same functional properties, engineers can choose the one that best fits their non-functional requirements. For instance, the busybox multi-call binary – which is tailored towards resource-constrained embedded systems – offers "trade bytes for speed" options in several places, allowing for tradeoffs between binary size and execution latency.

However, although behaviour models frequently consider non-functional properties such as timing or energy attributes [16, 29], and performance models can refer to runtime-specific performance attributes such as latency or energy usage [30], there is no link between the two. As such, engineers who want to optimize performance attributes are stuck with performance models that reference a specific, fixed workload, and are incapable of taking workload changes (i.e., variable runtime behaviour) into account.

Consider, for instance, a battery-powered wireless sensor node product line. Battery runtime depends on the time spent in each device state: radio communication is more costly than data processing, which is in turn more costly than a low-power sleep state. It also depends on the sequence of events: five 10 ms transmissions are more energy-intensive than a single 50 ms transmission [24]. A conventional performance model can only predict the effect of product line configuration on total energy usage for a fixed workload, so engineers cannot use it to predict how different workloads (e.g. more/less frequent radio transmission) would affect battery runtime. Addressing workload changes requires rebuilding the entire model from scratch, including time-intensive energy benchmarks.

Workloads also introduce *non-featured* variability. Consider the x264 video codec: while encoding latency depends on codec flags (features) such as target bitrate [33], it also depends on the input file. Higher input resolutions will naturally lead to increased encoding latency, yet input width and height are by no means features in the traditional sense. Indeed, publications that utilize x264 encoding latency as an evaluation target for performance modeling methods do not take variable input files into account, instead using a single file for learning and evaluation [15, 21, 32, 33, 35].

We observe three limitations of existing performance modeling methods: **(a)** each model references a single workload; **(b)** models for runtime behaviour consisting of multiple steps may be inaccurate; and **(c)** they do not support non-featured runtime variability.

We propose state machine-based *performance-aware behaviour models* to address these limitations. They define a link between behaviour models and performance models that can be used to predict arbitrary performance attributes on arbitrary workloads. They also introduce non-featured *runtime parameters* that are not part of the feature model, but still affect runtime performance.

Our performance-aware behaviour models have four benefits over conventional performance models: **(1)** support for variable workloads (i.e., variable event sequences, addressing (a)); **(2)** the ability to decompose complex runtime behaviour into individual steps (→ (b)); **(3)** support for non-featured parameters (→ (c)); and **(4)** the ability to work with partial performance models, reducing benchmarking and model learning overhead when working with inherently runtime-dependent performance attributes.

We contribute a definition of performance-aware behaviour models with support for numeric features, feature-dependent runtime behaviour, non-featured runtime parameters and arbitrary performance models, and three case studies on their application.

In the next section, we discuss the idea behind performance-aware behaviour models and give a definition that builds upon existing performance and behaviour modeling approaches. We follow up with three case studies in Section 3, including a quantitative evaluation of performance prediction accuracy. Afterwards, we discuss related work in Section 4 and conclude in Section 5.

2 Performance-Aware Behaviour Models

Product line engineers typically express behaviour models as labelled transition systems or extensions thereof [9, 11], and include *feature guards* that link boolean product line features with runtime behaviour by means of logic formulas. Although formalisms such as *weighted featured transition systems* (WFTS) and *featured weighted automata* (FWA) extend those with constant weights that can express timing or energy attributes [9, 16, 29], they do not take non-featured runtime parameters into account, and annotated behaviour models are limited to boolean product line features. *Dynamic software product lines* (DSPL), which support contextual (runtime-specific) product line features and runtime reconfiguration of features [23], face the same limitations.

Meanwhile, existing performance modeling approaches are unaware of behaviour models. They are either built into the feature model, annotating features or groups of features with their performance influence [8, 31, 34], or separate $\mathbb{R}^n \to \mathbb{R}$ functions that use n-dimensional *feature vectors* as a link to product line configuration [17]. Separate functions are typically more expressive and maintainable than built-in models, and also follow the notion of separation of concerns: implementation-independent feature and behaviour models should not be intertwined with implementation- and workload-dependent performance annotations [17].

Hence, we propose building performance-aware behaviour models as a combination of four components: a feature model, a set of non-featured runtime parameters, a set of behaviour models, and a set of performance models. Feature model and behaviour models only reference product line features. Each behaviour model component is annotated with a set of performance models; performance models are aware of feature configuration and runtime parameters.

We first define *Dynamic Product Lines with non-featured Runtime Variability* (DPL(rv)) to distinguish between static/dynamic features and non-featured runtime parameters. These build upon dynamic software product lines (DSPL) [23].

Definition 2.1. A dynamic product line with non-featured runtime variability is a tuple (F_s, F_d, P_d) consisting of a finite set of static features F_s, a set of dynamic (runtime-configurable) features $F_d \subseteq F_s$, and a finite set of runtime parameters P_d with $P_d \cap F_s = \emptyset$.

Features can be expressed as a conventional feature model. Runtime parameters are a set of variables without underlying structure.

Our behaviour models decompose finite product actions (e.g. database queries) into individual states and transitions. Thus, each workload whose performance we want to predict is defined as a finite series of events (i.e., a word) combined with a workload configuration $\vec{x} \in F_s \cup P_d$ that describes the corresponding static and dynamic feature configuration and runtime parameters. With this in mind, we define *Annotated Featured Automata* (AFA) as follows.

Definition 2.2. An annotated featured automaton for a DPL(rv) (F_s, F_d, P_d) is a tuple $(Q, q_0, F, \Sigma, \delta, \gamma, P, \lambda)$ consisting of a finite set of states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, a finite set of actions Σ, a transition function $\delta : Q \times \Sigma \rightharpoonup Q$, a guard mapping $\gamma : \delta \to \mathbb{B}(F_s)$, a set of performance attributes P, and a model mapping $\lambda : \delta \times P \rightharpoonup (\mathbb{R}^{|F_s|+|P_d|} \to \mathbb{R})$. The guard mapping γ annotates transitions with optional feature constraints such as $\neg X \vee Y$; see Atlee et al. for details [3]. The model mapping λ annotates transitions with optional performance models.

This definition builds upon featured transition systems [3, 10, 11]. However, we only consider finite, deterministic automata with a single initial state. A DPL(rv) can also be used with performance models $\mathbb{R}^{|F_s|} \to \mathbb{R}$ or $\mathbb{R}^{|F_s|+|P_d|} \to \mathbb{R}$ that are not part of the AFA.

3 Case Studies

We use three case studies to show the benefits of performance-aware behaviour models: throughput/latency prediction for a thumbnail generator with a configurable sub-sampling algorithm, throughput/latency prediction for an aggregating database query with optional offloading to specialized hardware, and energy usage analysis of a wireless sensor node product line with variable software and hardware components and configuration. We compare three kinds of performance models: *baseline* (a single function $\mathbb{R}^{|F_s|} \to \mathbb{R}$), *runtime* (a single function $\mathbb{R}^{|F_s|+|P_d|} \to \mathbb{R}$), and *AFA* (using $\lambda : \delta \times P \rightharpoonup (\mathbb{R}^{|F_s|+|P_d|} \to \mathbb{R})$). Artifacts are available at https://ess.cs.uos.de/git/artifacts/vamos25-behaviour-models.

3.1 Thumbnail Generation

Performance models for thumbnail image generation face similar challenges as the x264 video codec performance models discussed earlier. Hence, we examine a thumbnail generation product line based on `python3-pil`. Its static features (F_s) are default thumbnail width and height (*outW, outH* $\in \mathbb{N}$) and format (*JPEGout, WEBPout,*

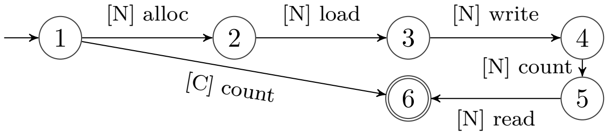


Figure 1: AFA behaviour model for a `count` query that can utilize near-memory computing [N] or conventional CPU threads [C] for query execution.

… $\in \{0, 1\}$) as well as resampling method (*nearest, box, bilinear, hamming, bicubic, lanczos* $\in \{0, 1\}$). All of these are dynamic features (F_d) as well, and can be adjusted at runtime. Input width and height (*inW, inH* $\in \mathbb{N}$) and format (*JPEGin, WEBPin, …* $\in \{0, 1\}$) are workload-dependent runtime parameters (P_d).

Thumbnail generation consists of three steps: loading the input file, scaling it down, and saving the output file. The corresponding AFA is a linear chain of four states with transitions *load*, *scale* and *save*, hence $L(\mathcal{A}) = \{\text{load} \cdot \text{scale} \cdot \text{save}\}$. We use automated benchmarks to measure throughput, total latency, and load/scale/save latency for a variety of workload configurations $\vec{x}$.

We then use our Regression Model Tree (RMT) learning algorithm to automatically build baseline, runtime, and AFA models [19]. We find that considering non-featured runtime parameters is crucial: with a cross-validated prediction error of 112.8 %, the baseline model is of little use. Meanwhile, runtime and AFA models achieve a prediction error of less than 5 %. Here, the AFA model does not provide notable accuracy benefits compared to the runtime model.

3.2 Database Queries

Our next evaluation target is a database product line with optional support for near-memory computing (NMC) hardware, which is seeing growing interest from the database systems community [5, 26]. With NMC, processing can be offloaded to special DRAM modules that incorporate hundreds of dedicated near-memory processing units, thus freeing CPU and memory controller capacity [12, 22]. This is beneficial for memory-intensive operations such as table scans, which are otherwise limited by the CPU–DRAM interface [6].

On currently available NMC hardware, offloading such an operation consists of allocating a suitable set of memory modules, uploading an application that implements the query, transferring the corresponding column, running the query, and reading back the results [22]. Here, the product line and performance modeling perspective becomes relevant: when building database systems with support for CPU and NMC execution, being able to predict query performance for both variants helps engineers choose sane defaults. Runtime placement algorithms can adjust these depending on workload patterns and resource availability.

Our product line's static and dynamic features ($F_s = F_d$) are CPU execution ($C \in \{0, 1\}$), NMC execution ($N \in \{0, 1\}$), the (default) number of threads used for CPU execution (*nThreads* $\in \mathbb{N}$), and the (default) number of DRAM ranks used for NMC execution (*nRanks* $\in \mathbb{N}$). Non-featured runtime parameters (P_d) are the number of database rows read by the query (*nRows* $\in \mathbb{N}$) and the NUMA distance between CPU threads and data (*numaDist* $\in \mathbb{N}$).

Fig. 1 shows the behaviour model for a `count` query. Again, we perform automated benchmarks over $L(\mathcal{A})$ while systematically exploring the configuration space and runtime parameter space, and use the RMT learning algorithm for model generation.

Here, both runtime and behaviour models play a vital role. For CPU execution, the runtime model reduces the cross-validated prediction error from 6.9 % to 2.3 %; AFA model accuracy is identical to runtime model accuracy. For NMC execution, the runtime model reduces the cross-validated prediction error from 50.9 % to 12.3 %, and the AFA model further decreases it to 9.7 %.

3.3 Wireless Sensor Nodes

Our third case study combines performance models and workload-specific benchmarks for trade-off analysis in a wireless sensor node product line. The nodes obtain environmental readings, encode them in a specific data serialization format, and transmit them over a wireless radio – either directly to a central hub or to an intermediate sensor node. As such, sensor nodes may also receive messages and deserialize them. Length and contents of received and sent messages are not known a priori, and must be observed in real-world benchmarks or provided as sample data.

When looking at the energy usage of such a product line, the selection of the data serialization format is especially interesting. Compact data (low transmission energy) often goes along with slow (de)serialization (high processing energy) and vice versa, and it is not clear which aspect is more important when optimizing for low energy usage [18]. Performance-aware behaviour models allow us to find the feature-dependent sweet spot for a typical workload.

The product line's static features (F_s) describe microcontroller hardware (*ATMega*, *MSP430*, *ESP8266*, *STM32* $\in \{0, 1\}$), radio hardware (*CC1200*, *ESP8266* $\in \{0, 1\}$), radio configuration (data rate and transmit power: *dr*, *tp* $\in \mathbb{N}$), and serialization method (*JSON*, *MessagePack*, *ProtoBuf*, *XDR* $\in \{0, 1\}$). Radio configuration features are dynamic ($F_d = \{dr, tp\}$), and the length of received and transmitted data is a non-featured runtime parameter ($P_d = \{len\}$).

In this case, we are looking at two behaviour models: a linear *receive* ⟶ *deserialize* chain and a linear *serialize* ⟶ *transmit* chain. We are interested in two performance attributes: latency t and mean power usage P. Given these, the energy usage E of a workload $w = \sigma_1 \cdots \sigma_k \in L(\mathcal{A})$ with workload configuration $\vec{x} \in F_s \cup P_d$ is $E = Pt = \sum_{i=1}^{k} \lambda(\sigma_i, t)(\vec{x}) \cdot \lambda(\sigma_i, P)(\vec{x})$.

We do not build baseline and runtime performance models here. Data processing latency depends on the data that is being processed, which is neither a product line feature nor a numeric runtime parameter. Using data length as a proxy metric is no good, either: data objects with identical length may have different processing latencies due to different data types or different levels of nesting.

Thanks to performance-aware behaviour models, we can define partial performance model annotations (λ) based on datasheet values and micro-benchmarks, and then use (de)serialize benchmarks to fill the gap. Those benchmarks only need to measure the latency of the (de)serialize operation for each data serialization format; there is no need for radio transmission benchmarks or energy measurements. Combining performance model annotations and latency benchmarks of real-world data objects allows us to analyze energy usage across the entire configuration space.

For instance, with the CC1200 radio operating at the lowest cost per Byte (250 kbit/s and −12 dBm), we find that both ProtoBuf and XDR are well-suited on ATMega, MSP430 and STM32, with XDR being slightly more energy-efficient. Changing radio configuration to the other extreme (highest cost per Byte: 15 kbit/s and +3 dBm) shifts the balance between processing and transmission energy and causes NanoPB to be slightly better. Meanwhile, on ESP8266, XDR is the most energy-efficient choice regardless of radio configuration.

Performing this analysis with conventional performance models would have required latency and energy benchmarks of the entire system, and the resulting models would be limited to the data objects used for the benchmarks. With performance-aware behaviour models, changing the set of data objects only requires new (de)serialization latency benchmarks.

4 Related Work

State machine-based behaviour models and extensions thereof have long been a part of product line engineering research. However, all performance-related publications that we are aware of focus on formal verification such as model checking and real-time analysis rather than performance optimization; they do not utilize generic performance models or support non-featured runtime parameters.

First of all, *Featured Transition Systems* (FTS) express how product line features affect product behaviour [11]. They extend transition systems with feature guards, indicating that certain transitions may only be taken if the corresponding logic formula is satisfied (e.g. if a specific set of features is enabled or disabled).

FTS applications include model checking with featured timed logic extensions (fLTL and fCTL) [10, 11], model-based testing of product families [7], and many similar applications [13]. FTS lack performance annotations, hence none of those relate to performance attributes or performance models.

There are two terms for extending an FTS with constant weights: *Weighted Featured Transition Systems* (WFTS) and *Featured Weighted Automata* (FWA) [16, 29]. These allow for family-based analysis of non-functional product line properties. For instance, Olaechea et al. present an algorithm for long-term behaviour analysis that can be used to determine the long-term average energy usage of all products within a product line [29]. Similarly, Bouyer et al. use energy annotations to verify operation under energy constraints [9]. They do so by proving that all products within a product line satisfy an invariant specifying that the amount of energy available within the system must not be negative, which is a mandatory condition for successful operation of energy harvesting-based embedded systems.

Featured Timed Automata (FTA) and *configurable Parametric Timed Automata* (coPTA) do not build upon FTS, but have similar uses [14, 27]. These extend timed automata with feature-dependent clock constraints, and thus also serve as a utility for family-based analysis and verification. Each feature-dependent clock constraint is a constant value.

These four approaches are more expressive than AFA in terms of support for formal verification of indefinite product operations, and less expressive in terms of lacking support for numeric product line features and non-featured runtime variability. With the former point, they provide an important direction for the evolution of performance-aware behaviour models.

Finally, as already mentioned in Section 2, *Dynamic Software Product Lines* (DSPL) take contextual (runtime-specific) product line features as well as runtime reconfiguration of features into account [23]. For instance, Göttmann et al. use these to verify both functional and non-functional reconfiguration constraints [20]. While this is a hard real-time application, it is also clearly distinct from a performance model. DSPL provide important utilities for runtime reconfiguration, which we have deliberately left out, but do not support non-featured variability.

5 Conclusion

We have presented performance-aware behaviour models for reasoning about performance attributes of feature-dependent runtime behaviour in product lines. In contrast to existing performance engineering approaches, these are capable of taking non-featured runtime variability as well as feature- and workload-dependent runtime behaviour into account. They allow engineers to improve the accuracy of existing performance models by decomposing system behaviour into discrete steps, and also support performance optimization with partial performance models.

We have defined Dynamic Product Lines with non-featured Runtime Variability (DPL(rv)) and Annotated Finite Automata (AFA) to express these models. DPL(rv) build on top of Dynamic Software Product Line research [23], while AFA are closely related to Featured Transition Systems [3, 10, 11]. Thus, they are compatible with many existing family-based formal verification methods for behaviour models with only minor changes.

Our case studies have shown that respecting non-featured runtime variability provides the greatest improvement in performance prediction accuracy, with an up to 90 % reduction in cross-validated performance prediction error. Behaviour decomposition further reduces prediction error by up to 15 %. Finally, even when only partial performance models are available, engineers can reason about workload-dependent performance optimizations with a minimal amount of micro-benchmarks.

While our AFA-based behaviour models do not support indefinite operations, runtime reconfiguration of dynamic features or workloads expressed as timed words, these limitations can be addressed by incorporating concepts from timed automata and encoding knowledge about dynamic features into the behaviour model.

Independent of these limitations, we have shown that using feature and configuration vectors to link behaviour models with separate performance models is beneficial for performance prediction accuracy. In addition to these practical considerations, doing so also follows the principle of separation of concerns: many product line performance attributes depend on implementation details and the workload experienced by the product, including feature-dependent runtime behaviour and non-featured runtime parameters. Hence, performance models should be independent of feature models and behaviour models, and we would like to encourage product line and performance engineers to focus on formalisms that clearly separate these concerns.

Acknowledgments

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 502565817.

References

[1] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. 2022. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *Proceedings of the 26th International Systems and Software Product Line Conference - Volume A* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 85–96. doi:10.1145/3546932.3546997

[2] Timo Asikainen, Tomi Mannisto, and Timo Soininen. 2006. A Unified Conceptual Foundation for Feature Modelling. In *Proceedings of the 10th International Software Product Line Conference* (Baltimore, MD, USA) *(SPLC '06)*. IEEE, 31–40. doi:10.1109/SPLINE.2006.1691575

[3] Joanne M. Atlee, Uli Fahrenberg, and Axel Legay. 2015. Measuring Behaviour Interactions between Product-Line Features. In *Proceedings of the 3rd FME Workshop on Formal Methods in Software Engineering* (Florence, Italy) *(FormaliSE '15)*. IEEE, 20–25. doi:10.1109/FormaliSE.2015.11

[4] Kacper Bąak, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *Proceedings of the 3rd International Conference on Software Language Engineering* (Eindhoven, The Netherlands) *(SLE '10)*. Springer Berlin, Heidelberg, 102–122. doi:10.1007/978-3-642-19440-5_7

[5] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Accelerating Large Table Scan using Processing-In-Memory Technology. In *Proceedings of the 20th Conference on Database Systems for Business, Technology and Web (BTW'23)*. Gesellschaft für Informatik e.V., Bonn, 797–814. doi:10.18420/BTW2023-51

[6] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Processing-in-Memory for Databases: Query Processing and Data Transfer. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 107–111. doi:10.1145/3592980.3595323

[7] Harsh Beohar and Mohammad Reza Mousavi. 2016. Input–output conformance testing for software product lines. *Journal of Logical and Algebraic Methods in Programming* 85, 6 (2016), 1131–1153. doi:10.1016/j.jlamp.2016.09.007 NWPT 2013.

[8] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. 2010. Introducing TVL, a Text-based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems* (Linz, Austria) *(VaMoS '10)*. Universität Duisburg-Essen, 159–162. doi:10.17185/duepublico/47086

[9] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Jiří Srba. 2008. Infinite Runs in Weighted Timed Automata with Energy Constraints. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08)*. Springer Berlin, Heidelberg, 33–47. doi:10.1007/978-3-540-85778-5_4

[10] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming* 80, PB (2 2014), 416–439. doi:10.5555/2748144.2748397

[11] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089. doi:10.1109/TSE.2012.86

[12] Stefano Corda, Gagandeep Singh, Ahsan Jawed Awan, Roel Jordans, and Henk Corporaal. 2019. Platform Independent Software Analysis for Near Memory Computing. In *Proceedings of the 22nd Euromicro Conference on Digital System Design (DSD '19)*. 606–609. doi:10.1109/DSD.2019.00093

[13] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. 2019. *A Decade of Featured Transition Systems*. Springer International Publishing, Cham, 285–312. doi:10.1007/978-3-030-30985-5_18

[14] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2012. Behavioural Modelling and Verification of Real-Time Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1* (Salvador, Brazil) *(SPLC '12)*. Association for Computing Machinery, New York, NY, USA, 66–75. doi:10.1145/2362536.2362549

[15] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2021. Mastering Uncertainty in Performance Estimations of Configurable Software Systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 684–696. doi:10.1145/3324884.3416620

[16] Uli Fahrenberg and Axel Legay. 2017. Featured Weighted Automata. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE '17)*. 51–57. doi:10.1109/FormaliSE.2017.2

[17] Birte Friesel, Michael Müller, Matheus Ferraz, and Olaf Spinczyk. 2022. On the Relation of Variability Modeling Languages and Non-Functional Properties. In *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 140–144. doi:10.1145/3503229.3547055

[18] Birte Friesel and Olaf Spinczyk. 2021. Data Serialization Formats for the Internet of Things. *Electronic Communications of the EASST* 80 (9 2021). doi:10.14279/tuj.eceasst.80.1134

[19] Birte Friesel and Olaf Spinczyk. 2022. Regression Model Trees: Compact Energy Models for Complex IoT Devices. In *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things* (Milan, Italy) *(CPS-IoTBench '22)*. IEEE, 1–6. doi:10.1109/CPS-IoTBench56135.2022.00007

[20] Hendrik Göttmann, Lars Luthmann, Malte Lochau, and Andy Schürr. 2020. Real-time-aware reconfiguration decisions for dynamic software product lines. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line - Volume A* (Montreal, QC, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 11 pages. doi:10.1145/3382025.3414945

[21] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering* 23, 3 (6 2018), 1826–1867. doi:10.1007/s10664-017-9573-6

[22] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608. doi:10.1109/ACCESS.2022.3174101

[23] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (2008), 93–95. doi:10.1109/MC.2008.123

[24] Philipp Hurni, Benjamin Nyffenegger, Torsten Braun, and Anton Hergenroeder. 2011. On the Accuracy of Software-Based Energy Estimation Techniques. In *Proceedings of the 8th European Conference on Wireless Sensor Networks* (Bonn, Germany) *(EWSN '11)*. Springer Berlin, Heidelberg, 49–64. doi:10.1007/978-3-642-19186-2_4

[25] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. https://insights.sei.cmu.edu/library/feature-oriented-domain-analysis-foda-feasibility-study/

[26] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proceedings of the ACM on Management of Data* 1, 2, Article 113 (jun 2023), 27 pages. doi:10.1145/3589258

[27] Lars Luthmann, Andreas Stephan, Johannes Bürdek, and Malte Lochau. 2017. Modeling and Testing Product Lines with Unbounded Parametric Real-Time Constraints. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A* (Sevilla, Spain) *(SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 104–113. doi:10.1145/3106195.3106204

[28] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Transactions on Software Engineering* 46, 7 (7 2020), 794–811. doi:10.1109/TSE.2018.2870895

[29] Rafael Olaechea, Uli Fahrenberg, Joanne M. Atlee, and Axel Legay. 2016. Long-term average cost in featured transition systems. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16)*. Association for Computing Machinery, New York, NY, USA, 109–118. doi:10.1145/2934466.2934473

[30] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software* 182 (2021), 111044. doi:10.1016/j.jss.2021.111044

[31] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems* (Namur, Belgium) *(VaMoS '11)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/1944892.1944894

[32] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE '15)*. Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/2786805.2786845

[33] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G Giarrusso, Sven Apel, and Sergiy S Kolesnikov. 2013. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology* 55, 3 (3 2013), 491–507. doi:10.1016/j.infsof.2012.07.020

[34] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal* 20, 3 (9 2012), 487–517. doi:10.1007/s11219-011-9152-9

[35] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, NE, USA) *(ASE '15)*. IEEE, 365–373. doi:10.1109/ASE.2015.15