Modelling, Reverse Engineering, and Learning Software Variability

Mathieu Acher

Habilitation à diriger des recherches

Mathieu Acher

Université de Rennes 1

Institut Universitaire de France (IUF)

Ecole Doctorale n° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Modelling, Reverse Engineering, and Learning Software Variability

Habilitation présentée et soutenue à Rennes le 16 Novembre 2021
Unité de recherche : IRISA – UMR6074

Composition du Jury :

Rapporteurs

Krzysztof Czarnecki	Professor University of Waterloo, Canada
Ina Schaefer	Professor Technische Universität Braunschweig, Germany
Romain Rouvoy	Professor University of Lille, IUF, France

Examinateurs

Julia Lawall	Senior Research Scientist Inria, France
Christian Kästner	Associate Professor Carnegie Mellon University, USA
Jean-Marc Jézéquel	Professor University of Rennes 1, France

Contents

Chapter 1

Introduction

> Demander quelque chose au sujet d'une chose en termes d'elle-même, ce n'est en rien faire une recherche; mais pour se poser vraiment une question au sujet de quelque chose, il faut s'interroger sur autre chose
>
> *Thomas d'Aquin, Commentaire de la Métaphysique d'Aristote, L. VII, leçon 17, § 1664.*

Since the very beginning of my research journey in 2008, the idea of varying software has never stopped to fascinate me. Without much surprise, I am starting this manuscript and first chapter with a gentle introduction to software variability (Section 1.1). I continue this chapter with the challenges and objectives addressed in the contributions presented in this habilitation (Section 1.2). Then I present an overview of my contributions (Section 1.3). Section 1.4 describes the research methods followed during my research activities. Since the contributions are the result of a collaborative effort, I list in Section 1.5 the various collaborations with students, researchers and software engineers who contributed to the results. Section 1.6 presents the research grants, industrial contracts and collaborative projects which supported the overall research activities. Finally, Section 1.7 describes how to read the document.

Contents

1.1 Context

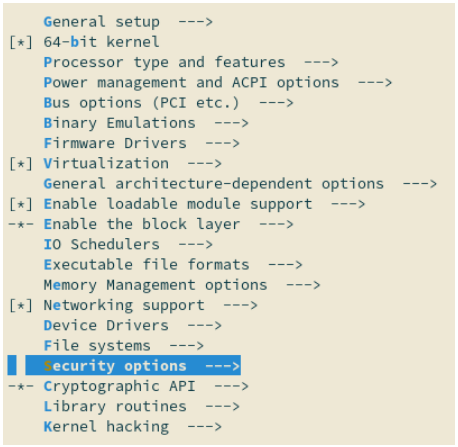
There is no doubt that the world is becoming increasingly dependent on software. It is now an essential element of many organizations (finance, retail, public sectors) and even our daily lives depend on complex software-intensive systems, from banking and communications to transportation and medicine. For decades, the challenge for the research community and the industry has been to provide the right languages, abstractions, models, methods, and tools to assist software developers in building high-quality software capable of fitting various requirements, contexts, and usages. Society expects software to deliver the right functionality, in a short amount of time and with fewer resources, in every possible circumstance whatever are the hardware, the operating systems, the compilers, or the data fed as input.

For fitting such a diversity of needs, it is common that software comes in many **variants** and is **highly configurable** through configuration options, runtime parameters, conditional compilation, command-line options, configuration files, plugins, *etc.* Web browsers like Firefox or Chrome are available on different operating systems, in different languages, while users can configure hundreds of preferences or install numerous 3rd parties extensions (or plugins). Web servers like Apache, operating systems like the Linux kernel (see Figure 1.1(c) and 1.1(d)), or a video encoder like x264 (see Figure 1.2(d) and 1.2(c)) are additional examples of software systems that are highly configurable at compile-time or at run-time for delivering the expected functionality and meeting the various desires of users. As there is no one-size-fits-all solution, **software variability** (*"the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"*) has been studied the last two decades and is a discipline of its own [295, 251, 33, 276]. It is also known in the research community as *Software Product Line (SPL)* engineering. For decades, international conferences like SPLC or VaMoS have brought together numerous academics and industrials. Different problems are considered in the community and target software product lines, configurable systems, dynamic adaptive systems, and variability-intensive systems in general. SPL engineering pursues the goal of developing a family of related products (aka variants), by embracing the ideas of mass customization and software reuse. It focuses on the means of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them.

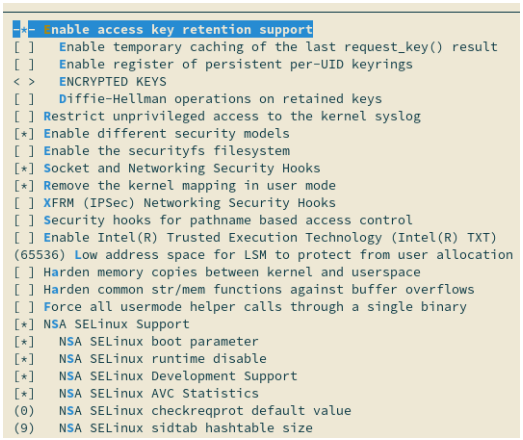
This is analogous to what is practiced in the automotive industry, where the focus is on creating a single production line, out of which many customized but similar variations of a car model are produced. It should be noted that the automotive industry is now facing the challenge of embedding custom software into their car model. Stated differently, product line engineering is also a software product line engineering problem. Figure 1.1(a) and 1.1(b) give another intuition of SPL engineering: when variability is well managed, it is easier to produce variants. Of course, the underlying artefacts as well as the number of possible variants in real-world systems are much more complex than this kids' puzzle.

(a) Kids' puzzle (variability is a mess)

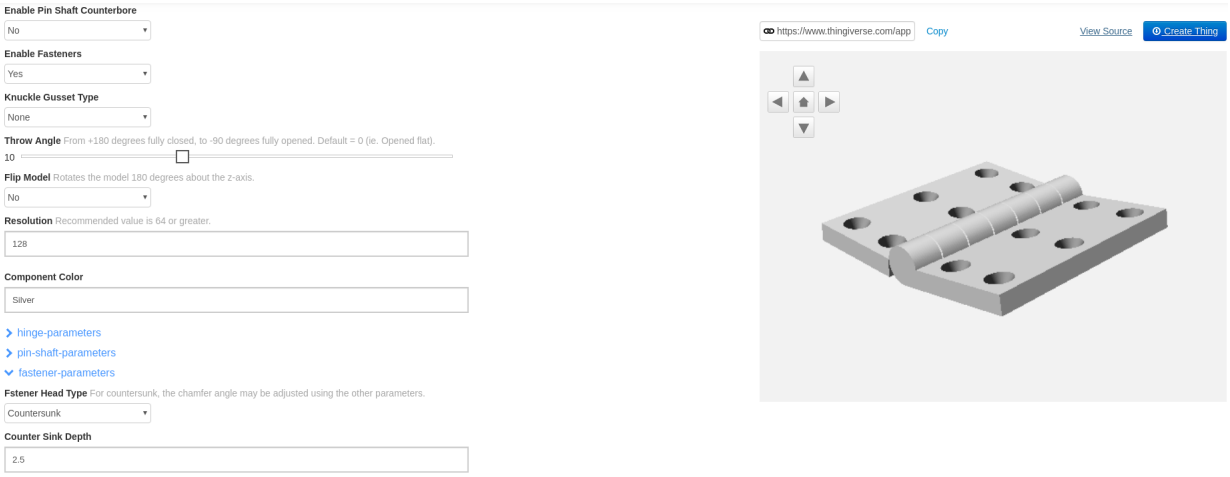
(b) Re-engineering variability (ready to assemble)



(c) Linux menuconfig (general)



(d) Linux menuconfig (security)



(e) Customizable model (3D printing)

Figure 1.1: Software variability in the wild: A gallery

```
    <%_ if (uuidConversionNeeded) { _%>
import java.util.UUID;
    <%_ } _%>
<%_ } _%>

@Configuration
<%_ if (reactive) { _%>
@EnableR2dbcRepositories("<%= packageName %>.repository")
<%_ } else { _%>
@EnableJpaRepositories("<%= packageName %>.repository")
@EnableJpaAuditing(auditorAwareRef = "springSecurityAuditorAware")
<%_ } _%>
@EnableTransactionManagement
<%_ if (searchEngine === 'elasticsearch') { _%>
@Enable<% if (reactive) { %>Reactive<% } %>ElasticsearchRepositories("<%= packageName
<%_ } _%>
public class DatabaseConfiguration {

<%_ if (devDatabaseType === 'h2Disk' || devDatabaseType === 'h2Memory') { _%>
    private final Logger log = LoggerFactory.getLogger(DatabaseConfiguration.class);
```

(a) JHipster variability (Java)

```
  <%_ if (databaseType === 'sql') { _%>
        <!-- The hibernate version should match the one managed by
        https://mvnrepository.com/artifact/org.springframework.boot/spring
        <hibernate.version><%= HIBERNATE_VERSION %></hibernate.version>
    <%_ if (!reactive) { _%>
        <!-- The javassist version should match the one managed by
        https://mvnrepository.com/artifact/org.hibernate/hibernate-core/${
        <javassist.version>3.27.0-GA</javassist.version>
    <%_ } _%>
        <!-- The liquibase version should match the one managed by
        https://mvnrepository.com/artifact/org.springframework.boot/spring
        <liquibase.version><%= LIQUIBASE_VERSION %></liquibase.version>
    <%_ if (!reactive) { _%>
        <liquibase-hibernate5.version>4.3.1</liquibase-hibernate5.version>
    <%_ } _%>
    <%_ if (devDatabaseType === 'h2Disk') { _%>
        <h2.version>1.4.200</h2.version>
    <%_ } _%>
        <validation-api.version>2.0.1.Final</validation-api.version>
  <%_ } _%>
```

(b) JHipster variability (Maven)

```
#if HAVE_GPAC || HAVE_LSMASH
        cli_output = mp4_output;
        param->b_annexb = 0;
        param->b_repeat_headers = 0;
        if( param->i_nal_hrd == X264_NAL_HRD_CBR )
        {
            x264_cli_log( "x264", X264_LOG_WARNING, "cbr nal-hrd is not compatible with mp4\n" );
            param->i_nal_hrd = X264_NAL_HRD_VBR;
        }
#else
        x264_cli_log( "x264", X264_LOG_ERROR, "not compiled with MP4 output support\n" );
        return -1;
#endif
```

(c) Variability in x264 code

```
--asm <integer>         Override CPU detection
--no-asm                Disable all CPU optimizations
--opencl                Enable use of OpenCL
--opencl-clbin <string> Specify path of compiled OpenCL kernel cache
--opencl-device <integer> Specify OpenCL device ordinal
--dump-yuv <string>     Save reconstructed frames
--sps-id <integer>      Set SPS and PPS id numbers [0]
--aud                   Use access unit delimiters
--force-cfr             Force constant framerate timestamp generation
--tcfile-in <string>    Force timestamp generation with timecode file
--tcfile-out <string>   Output timecode v2 file from input timestamps
--timebase <int/int>    Specify timebase numerator and denominator
           <integer>    Specify timebase numerator for input timecode file
                        or specify timebase denominator for other input
--dts-compress          Eliminate initial delay with container DTS hack
```

(d) x264 command-line options

Figure 1.2: Software variability in the wild: A gallery

In SPL engineering as in many software engineering contexts, software variability is a key concern. Different kinds of users are intensively relying on software variability:

- end-users through *e.g.*, menu preferences or configurators [179], potentially non computer experts;
- administrators, release or product managers in charge of configuring, compiling and deploying software systems in variable settings [275];
- developers in charge of implementing, maintaining, and testing software variabilty;
- scientists that rely on software to analyze data and need to calibrate their solutions [77];
- software systems themselves are capable of automatically varying on-demand [137, 221, 169, 298].

Software variability spans multiple application domains (see Figure 1.1, 1.2 and 1.3): operating systems, 3D printing, document generation, video encoding, computer vision, puzzles/games, Web applications, *etc.* Different kinds of artefacts, involving different software languages, are subject to variations (*e.g.*, videos, Java, C, LaTeX, or Python programs, Maven files).

1.2 Challenges and Objectives

Though highly desirable, software variability also introduces an enormous complexity due to the combinatorial explosion of possible variants. For example, the Linux kernel has 15000+ options and most of them can have 3 values: "yes", "no", or "module". Overall, there may be more than 10^{5000} possible variants of Linux (the estimated number of atoms in the universe is 10^{80} and is already reached with 300 Boolean options). Though there are numerous constraints among Linux options, the number of possible variants is enormous. Furthermore, building one configuration of the Linux kernel is costly: around 8 minutes on average on a recent machine [5]. Linux is an extreme case, but other systems quickly induce combinatorial issues related to variability. The video encoder x264 provides a help page of 400 lines, documenting dozens of options (see Figure 1.2(d)). In the 3D printing area, a customizable model may provide dozens of Boolean and numerical options leading to billions of possible variants (see Figure 1.1(e)). In practice, it is hardly possible to fully explore and understand all software options, in all possible settings. This situation has several consequences. On the one hand, developers struggle to maintain, understand, and test variability spaces since they can hardly analyze or execute all variants in every possible settings. According to several studies [275, 135], the flexibility brought by variability is expensive as configuration failures represent one of the most common types of software failures. On the other hand, end-users fear software variability and stick to default configurations [323] that may be sub-optimal (e.g., the software system will run very slowly) or simply inadequate (*e.g.*, the quality of the output will be awful).

```
\ifVal{ACK}{
\ifVal{BOLD_ACK}{\textbf{Acknowledgements.}}
\ifVal{PARAGRAPH_ACK}{\paragraph{Acknowledgements}}
\ifVal{LONG_ACK}{We thank Pierre Laperdrix for the
% project fundings also
}
%
\scriptsize
%\vspace*{-2mm}
\vspace*{-\getVal{vspace_bib}mm}
\bibliographystyle{abbrv}
\bibliography{DEModularity15}
```

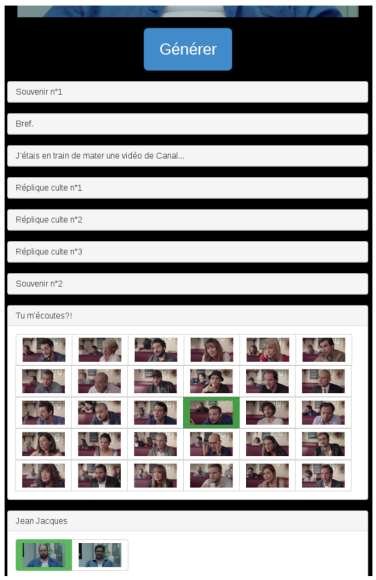
(a) VaryLaTeX [26]

algorithmics

Q: The algorithmic package does not produce line numbers when used with babel and Hebrew!

A: use this patched version of algorithmic.sty instead.

(b) Feature interaction https://www.cs.bgu.ac.il/~yoavg//tech-notes/heblatex/

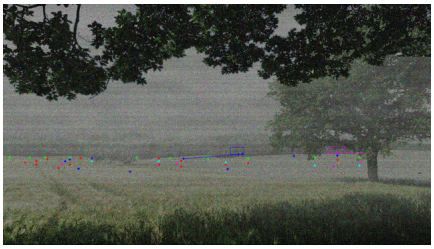


(c) Web generator [48] (entertainment)

```
# generating a puzzle variant
ncols=nlines=8 # we can parameterize the size of the board ncols*nlines
N=5 # eg sounds impossible to place 5 rooks, 5 bishops, 9 knights on 6x6
pk=[c.QUEEN for i in range(0,N)]
pk[4]=c.KNIGHT
#pk[3]=c.ROOK
#pk[2]=c.KNIGHT

# resolving the puzzle variant
l=set(filter(lambda x:x>>3<ncols and x&7<nlines,range(0,64)))
while(True):
    s=gen_rand()
    b=c.Board(fen=None)
    for q in range(0,N):
        b.set_piece_at(s[q],c.Piece(pk[q],w))
    if not set().union(*[list(b.attacks(s[q]))for q in range(0,N)]) & set(s):
        print(b, "you find a solution!")
        break
```

(d) Chess puzzle variants http://blog.mathieuacher.com/ProgrammingChessPuzzles/



(e) Video variant [7]

(f) Video variant [7] (bis)

Figure 1.3: Software variability in the wild: A gallery (cont'n)

There are several research questions that can be addressed:

- Is there a common language to *express* variations in artefacts as different as videos, 3D model, Java source code, Maven file, or operating systems? Owing to the diversity of situations (see Figure 1.1, 1.2 and 1.3), finding an universal formalism might be hard. Hence a more reasonable objective is to find reusable constructs that can be applied for systematically expressing variability.
- How to *verify* and *validate* variability-intensive systems? There are subtle interactions among options (or features) that cause bugs (see Figure 1.3(b)). Looking at the Java code snippet of Figure 1.2(a), the reader can feel the underlying complexity: Will the Java code compile whatever options' values fed to the generator? Will this Java code be running consistently with the variations also expressed in the Maven file (see Figure 1.2(b))? The challenge is mostly to deal with the combinatorial explosion, which requires adapting verification techniques or considering a limited sample of configurations.
- How to assist users in *configuring* software systems? Among the numerous possible configurations and variants, the objective is to find a subset that fits *e.g.*, performance requirements. For instance, what parameters' values should be set for the video encoder x264 (see Figure 1.2(d)) in case a fast execution time is a top priority? A challenge is again the enormous variability space: it is practically impossible to execute and measure all configurations *a priori*.
- Where is software variability in the wild? Can we identify, extract, re-engineer, or improve such variability? Figure 1.1, 1.2 and 1.3 give some examples, but there are many other domains, systems, and engineering contexts worth considering.

The list of questions is of course non-exhaustive but I consider them as quite representative of the open problems investigated in the field. A requirement that is common to all these questions is the ability to synthesize the right abstractions (models) for expressing, verifying, validating, and configuring software variability. Without a proper representation of the variability, it is hard, not to say impossible, to explore, observe, and reason about the space of possible variants. In short, a key and central question addressed in this HDR manuscript is:

How to model software variability?

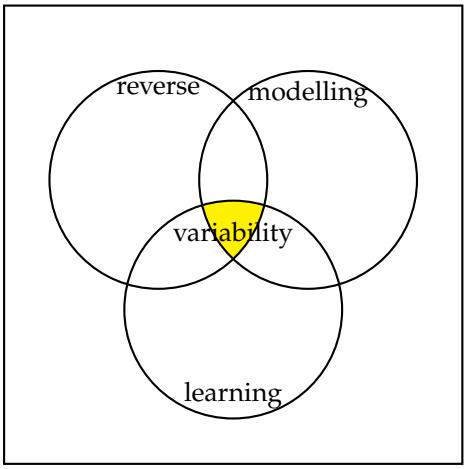


Figure 1.4: Modelling, reverse engineering, and learning software variability

1.3 Overview of Scientific Contributions

I am not formulating a complete answer to the question now and here, Chapter Conclusion will. I have followed three different paths to model variability (see Figure 1.4).

Firstly, I contribute to support the persons in charge of manually specifying feature models, the *de facto* standard for modeling variability. I develop an algebra together with a language for supporting the composition, decomposition, diff, refactoring, and reasoning of feature models [16, 20, 47]. A key idea is to rely on logics to provide guarantees about the configuration semantics. I further establish the syntactic and semantic relationships between feature models and product comparison matrices, a large class of tabular data [271, 52]. I then empirically investigate how these feature models can be used to test in the large configurable systems with different sampling strategies [135, 249]. Throughout this path, I continuously report on the attempts and lessons learned when defining the "right" variability language [27].

Secondly, I contribute to synthesize variability information into models and from various kinds of artefacts. I develop foundations and methods for reverse engineering variability models from satisfiability formulae [47, 51], product comparison matrices [49, 102], dependencies files and architectural information [54], and from Web configurators [179, 178]. The underlying objective of this research direction is to automate the task of modeling variability and exploit opportunities to mine variability information informally expressed here and there. I also report on the degree of automation and show that the involvement of developers and domain experts is beneficial to obtain high-quality models.

Thirdly, I contribute to learning constraints and non-functional properties (performance) of a variability-intensive system [298–300, 28, 209, 200, 25]. I describe a systematic process [29] "sampling, measuring, learning" that aims to enforce or augment a variability model, capturing variability knowledge that domain experts can find it difficult to express. I show that supervised, statistical machine learning can be used to synthesize rules or build prediction models in an accurate and interpretable way. This process can even be applied to a huge configuration space, such that of the Linux kernel one [25, 208].

Finally, I show that the three contributions "modeling", "reverse engineering", and "learning" (1) have pros and cons, (2) can be combined to produce an integrated variability model of a system under study (see coloured part of Figure 1.4).

1.4 Research Methods

As part of my research in software variability, software product line engineering, and configurable systems, I am trying to consider:

- as much as possible application domains (medical imaging and grid computing during my PhD, and then video processing, 3D printing, paper generation, Web applications, operating systems, *etc.*). A key lesson I learned during my PhD is the added value of collaborating with domain experts to better understand the specificites of their problems. The diversification of application domains is also a good way to question your proposal, being a theory, a method, or a tool. Section "In search of the right variability language and models" gives a good example of this questioning;
- widely-used, realistic open source projets (JHipster, Linux, x264, etc.) or industrial systems (collaborations with Thales, DGA, *etc.*). It is related to the first point: working on real-world projects challenges your contributions. There is also the ambition to have concrete impact on the practices and quality of software projects (see Section "Effectiveness of sampling strategies for testing", Section "Reverse engineering architectural variability models" and Section "Learning variability constraints" for examples);
- classes of general problems *e.g.*, reverse engineering and more recently "learning" software variability spaces. Decomposing the variability problem is definitely helpful. A challenge and actually a contribution is to identify such relevant sub-problems (see the three next chapters) and then connect them together (see the conclusion of the manuscipt);
- different techniques at the intersection of software engineering and artificial intelligence (SAT solving/CP programming, supervised machine learning, software testing/performance engineering, model-driven engineering, *etc.*). As part of the research, there are two aspects to consider: i) identifying what is suited for the targeted problem; ii) properly combining the techniques. There are many examples in this manuscript;
- both foundational and empirical approaches: On the one hand, I am convinced formally defined solutions with guaranteeing properties (*e.g.*, soundness) are sometimes possible. On the other hand, identifying what kinds of distributions are relevant to the “real world software” is a key issue. Overall, the two approaches are complementary and my ultimate goal is to understand what kinds of algorithms perform well on software variability data drawn from the kinds of distributions we care about. For instance, the foundations for reverse engineering Web configurators have been designed thanks to an in-depth, empirical study (see Section Reverse engineering Web configurators);
- variability techniques beyond the engineering of "pure" software product lines: see above the application domains, but also publications related to Wikipedia [271, 142] or testing (*e.g.*, multimorphic testing [296]). I consider the scope of software variability is still expanding and should not be restricted to classes of software systems (*e.g.*, software product lines).

Case studies. I have used different research methods (controlled experiments [249, 28], survey [29, 23], *etc.*). The tradeoff to find between internal and external validity is one of the most challenging issue I faced as a researcher and academic [286]. However I must admit I have been much more comfortable with case study research. More specifically, I like to focus on one specific subject system or domain at a time. As a researcher, you can deeply understand the specificities of the problem, report on qualitative insights, and possibly confront quantitative insights (*e.g.*, metrics) with domain experts. It is quite difficult to repeat the effort for as many systems or at least to comprehensively report on the findings into a paper. On the other hand, you cannot generalize from one case; it is simply not the goal. The hope is that other studies are performed to validate or refute your findings, methods, or theories. Besides, some works consider a large number of of systems. However, when digging into the individual systems, I have observed that the assumptions made about the systems and the problem do not hold – and so the solution. Stated differently, some papers are trying to mitigate external validity with the inclusion of many systems, but what if the individual systems are superficially treated? Do not get me wrong: there are plenty of excellent papers with multiple systems that do not have such limitations; we definitely need them. My point is that this phenomenon can be better controlled with the focus on specific cases.

The role of teaching software variability I like very much teaching, especially when it is related to advanced research topics and open problems (*e.g.*, see my report of a representative year http://blog.mathieuacher.com/Teaching1819/ in non-COVID era). In fact I consider that teaching can be beneficial for research. I have tried to follow the moto "Teach or perish!": What is the point of doing research in the software variability field if you are unable to disseminate your results and train the engineers of tomorrow? As other colleagues advocated [23], teaching software variability is challenging. I toke this challenge as an opportunity to question my own research and results of the state of the art. For instance, interactions with students strongly influenced the work on product comparison matrices, since the relationship with feature models was not crystal clear. I also use courses to explore some ideas (*e.g.*, metamorphic domain-specific languages [19], multimorphic testing [296]). Overall I see teaching as an interesting feedback-loop for testing and refining variability-related works.

From this perspective, I am co-leading a worldwide initiative http://teaching.variability.io/ for disseminating the constantly growing body of software product line knowledge. This repository aims to share and deliver teaching material related to variability, configurable systems or generative approaches. It is notably the result of three international workshops I have co-organized and various surveys [23].

Software development Together with students and colleagues, I have developed numerous software projects mainly to support reseach activities but also with the objective of having concrete impacts. FAMILIAR (see Section FAMILIAR, a language for combining feature model operators), OpenCompare (see Section Feature models and product comparison matrices), VM (see Section In search of the right variability language and models), TuxML (see Section Transfer learning across variants and versions: the case of Linux) are the most visible and have required substantial effort. Some of these projects are unfortunately no longer maintained. Finding a sustainable model is a difficult challenge that goes beyond academic concerns. Despite relative failures, I am still a strong supporter of developing software. First, you can find new research opportunities along the way, potentially outside your

Name	Rate	Period	Funding	Topic	Position
Sana Ben Nasr	60%	2013-2016 (defense: apr. 2016)	CONNEXION project (with EDF)	Mining and modeling variability	PhD student
Guillaume Bécan	70%	2013-2016 (defense: sep. 2016)	MESR	Reverse engineering and synthesis	PhD student
Paul Temple	50%	2015-2018 (defense sep. 2018)	MESR	Learning variability	PhD student
Quentin Plazar	50%	2015-2018 (not defended)	ANR SOPRANO	Automated reasoning	PhD student
Hugo Martin	50%	2018-2021 (defense: planned)	ANR VaryVary	Learning variability	PhD student
Luc Lesoil	50%	2020-2023	ANR VaryVary	Deep Variability	PhD student
Juliana Alves Pereira	100%	2018-2020	ANR VaryVary	Learning variability	Post-doc
Jin Hyun Kim	50%	2014-2016	SAD (Britany region)	Formal verification	Post-doc
Mauricio Alvares	50%	2012-2014	MOTIV project (DGA+Bertin+InPixal)	Modeling variability	Post-doc
Xhevahire Tërnava	100%	2020-2022	SLIMFAST	Debloating variability	Post-doc

original domain of expertise (see *e.g.*, [19, 45]). Second, you can understand in a fine-grained way some (software) engineering problems through practice. I am still developing with my students and hope to continue. Third, I consider reproducible software is mandatory to conduct research (see also Section Software Variability and Science). Fourth, I still have hope that one day we will be able to develop mainstream solutions.

Community and collaborations Finally, my research activities are strongly grounded in many collaborations. Most of the results are acknowledged to the various Master and PhD students, software engineers and post-doctoral researchers that I have been pleased to supervise (see Section 1.5). Also, the vision I developed has been motivated and evaluated on case studies provided by industrial partners which are essential to keep focus and problem-driven the research activities (see Section 1.6). Finally, most of the ideas result from various discussions and collaborations with colleagues around the world, either within collaborative projects or through informal discussions in scientific events (visits to universities, workshops, seminars and conferences).

1.5 Supervision

The work presented here results from collaborations I have had with many researchers all over the world, my colleagues in the DiverSE team, as well as students I supervised during their Masters and PhD thesis, and post-docs I supervised and worked with on specific projects. I have co-supervised 4 PhD theses, 3 of which were defended in April 2016, September 2016, and December 2018, and one thesis not defended (despite publications at IJCAI 2017 [248] and ICST 2019 [249]). I am currently co-supervising 2 PhD theses. I have also supervised 3 post-docs and I am currently supersing 1 post-doc as part of SLIMFAST.

In addition to the PhD students I officially supervised at University of Rennes 1, France, I also enjoyed to closely work with various other PhD students in their research projects [271, 106, 267]. I was in the PhD defense of Bosco Filho, Ebrahim K. Abbasi, and in the PhD committee of José Galindo.

I have also supervised Master thesis oriented towards research: Axel Halin and Alexandre Nuttinck (University of Namur) [135, 136], Benoit Amand (University of Namur) [31], Georges Aaron Randrianaina (ENS Rennes), Paul Le Gall, Bruno Merciol.

Finally, I have have been fortunate to supervise a number of students for developing software related to research activities (FAMILIAR, opencompare, TuxML, *etc.*).

1.6 Grants, Contracts, and Projects

The research work presented in this document has been supported by various research grants, bilateral contracts with industry, as well as international and national collaborative projects. They provided the necessary funding to realize the research work, including the research staff (internships, PhD students, post-doctoral researchers and software engineers) and scientific environment. These collaborations also provided great opportunities to motivate, challenge, experiment, and possibly validate our solutions in industrial settings.

Among others, the ITEA2 MERgE (http://www.merge-project.eu/) supported the development of the research work related to variability in systems engineering, especially thanks to collaborations with Melexis and Thales [313, 198]; the Inria-Thales bilateral contract VaryMDE 2011-2015 (Variability in Model Driven Engineering, http://varymde.gforge.inria.fr/) supported the research work related to the reuse and variability management of modeling languages; the CONNEXION project (collaboration with EDF) challenged us to mine variability in textual documents; the MOTIV project (collaboration with DGA, Bertin, and InPixal) investigated variability in the video domain.

More recently, I am fortunate to lead the VaryVary ANR JCJC (see https://anr.fr/Projet-ANR-17-CE25-0010 and https://varyvary.github.io/). I am also leading the SLIMFAST project (funded by Britany region and DGA).

1.7 Organization of the manuscript

How to read this manuscript? I am using two kinds of boxes throughout the 12 sections of the 3 core chapters. First, I am mentioning the main publications related to each section. There are three reasons: (1) I am intensively reusing the material of such publications; (2) the interested reader can find more details; (3) the publications are representative of a research line I have investigated.

Second, I have made the effort to verify whether my work can be potentially reproduced. I am mentioning online resources that reference tools, data, script, documentation, models, *etc.* developed as part of the conducted studies. The interested reader can use these resources to continue the research. From a personal perspective, I found this exercise interesting, but it also raises some questions (see also Section Software Variability and Science).

replication

Terminology The term *"feature"* is used all over the manuscript. Features can refer to very different concepts, at different levels of abstractions (*e.g.*, from a portion of code in a C program to a high-level description of a product functionality). In a sense, the meaning of feature is specific to a context and I have tried to clarify it in the dedicated sections. A more problematic usage of feature is when features actually correspond to a feature in a video, in an image, or to a predictive variable in a statistical learning problem. For this reason, I am using the term *"option"* (or configuration option) to make the distinction between "features" as used in related domains. However, for some parts of the manuscript, it is simply too difficult and I stick to "features".

Another potential issue with the terminology is related to configuration *vs* product *vs* variant. Generally speaking, configuration is used at the problem space level and corresponds to an assignment of options' values. Variants (and products) are used at the solution space level and are typically built and executed. A configuration is fed to a build system, a generator, a software product line, or a configurable system to produce and execute a variant (or product) of the system. However, there are exceptions here and there in the manuscript. For instance, in a product comparison matrix (see *e.g.*, Section 2.2), the description of product is rather abstract and resides at the problem space level (*i.e.*, at the same level of abstraction than a configuration).

I have done my best to use a consistent and unified terminology throughout the manuscript. Yet, I would not be surprised that some inconsistencies remain. I would appreciate any feedback to pinpoint some errors.

Novelty As mentioned, I reuse lots of material from previous publications. Beyond a careful editing, what is new in this manuscript is:

- the contextualization of each chapter and section;
- the "Wrap up, applicability, and limitations" sections at the end of the 3 core chapters for summarizing a line of research in an original way;
- the "Conclusion" section that answers the question of Section 1.1 and connects the dots between the three chapters;
- the "Perspectives" section that describes new directions and ideas;
- some retrospective analysis and new concept/terminology (*e.g.*, approximate configuration oracle) in some sections.

Remainder In the rest of this document I first present contributions related to the modeling of variability (Chapter 2). Chapter 3 describes contributions for reverse engineering models of variability. Chapter 4 details how learning techniques can be used to refine or augment variability models. Chapter 5 concludes this document with a wrap-up of the research activities conducted during the last decade and the main outcomes. Finally, Chapter 6 introduces a broader vision and four main research directions.

Chapter 2

Modelling Software Variability

In this chapter I present a set techniques and formalisms to specify models of variability. I consider that a model is an abstraction of a system under study (some details are hidden or removed to simplify and focus attention) [187]. Models are created to serve particular purposes, for example, to present a human understandable description or to present information in a form that can be mechanically analyzed. This chapter mainly focuses on scenarios in which persons (domain experts, developers, testers, *etc.*) elaborate and write models, possibly with automated facilities to support their (meta-)modelling activities. It is in contrast with the two next chapters in which, as I will detail, models are automatically obtained, refined or augmented. This chapter also questions the language suited to specify models of variability. I first focus on the formalism of feature models, then I explain the design of a metamodel for modelling product comparison matrices, and finally the development of the VM language to meet the requirements of an industrial video generator.

Section 2.1 describes a generic solution together with a language (FAMILIAR) for composing, decomposing, diffing, and refactoring feature models. Section 2.2 studies so-called product comparison matrices, their relationships with feature models, and details the data-driven design of specific metamodels. Section 2.3 investigates how feature models can be used to test in the large configurable systems with different sampling strategies. Section 2.4 reports on our experience in modelling variability in the video domain.

Contents

2.1 Automated feature model management

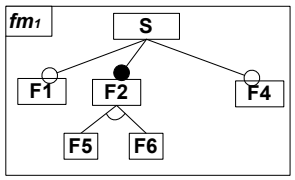
The content of this section is adapted from the following publications:
M. Acher, P. Collet, P. Lahire and R. France, 'FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models', *Science of Computer Programming (SCP) Special issue on programming languages*, vol. 78, no. 6, pp. 657–681, 2013. DOI: {http://dx.doi.org/10.1016/j.scico.2012.12.004}
M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire and R. B. France, 'Composing your Compositions of Variability Models', in *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, 2013
G. Bécan, M. Acher, B. Baudry and S. Ben Nasr, 'Breathing Ontological Knowledge Into Feature Model Synthesis: An Empirical Study', *Empirical Software Engineering (ESE)*, vol. 21, no. 4, pp. 1794–1841, 2016. DOI: 10.1007/s10664-014-9357-1. https://hal.inria.fr/hal-01096969

Designing, developing and maintaining software systems for one customer, one hardware device, one operating system, one user interface or one execution context is no longer an option. When properly managed, variability can lead to order-of-magnitude improvements in cost, time-to-market, and productivity of products. Models are traditionally employed to formally identify, organize and configure variability of a system, automate the generation of products as well as their verification. A variety of models may be used for different development activities and artifacts – ranging from requirements, source code, certifications and tests to user interfaces. In this line of work, our contributions provide automated support for composing, decomposing, diffing, and refactoring feature models (see Section 2.1.1). This support comes with the FAMILIAR language that can be used to address different scenarios when modelling variability (see Section 2.1.2).

Background about feature models

Feature Models are a widely used formalism for modelling and reasoning about commonality and variability of a system [94]. A recent survey of variability modelling showed that feature models are by far the most frequently reported notation in industry [59].

A feature model is a hierarchical organization of features that aims to represent the constraints under which features occur together in product configurations. When decomposing a feature into subfeatures, the subfeatures may be optional or mandatory or may form *Xor-* or *Or*-groups (see Figure 2.1(a) for a visual representation of a feature model). Not all combinations of features (*configurations*) are authorized by a feature model. Importantly, the hierarchy imposes some constraints: the presence of a *child* feature in a configuration logically implies the presence of its *parent* (*e.g.*, the selection of F5 implies the selection of F2). The hierarchy also helps to conceptually organize the features into different levels of increasing detail, thus defining an *ontological semantics*.



(a) fm_1

$[\![fm_1]\!] = \{\{S, F2, F6\},$
$\{S, F2, F5, F1\},$
$\{S, F2, F5, F4\},$
$\{S, F2, F5, F1, F4\},$
$\{S, F2, F6, F1\},$
$\{S, F2, F6, F4\}\}$

(b) Configurations

$\phi_{fm_1} = S$ // root
$\wedge \quad F2 \Leftrightarrow S$ // mandatory
$\wedge \quad F1 \Rightarrow S$ // optional
$\wedge \quad F4 \Rightarrow S$ // optional
// Xor-group
$\wedge \quad F5 \Rightarrow F2 \wedge F6 \Rightarrow F2$
$\wedge \quad F2 \Rightarrow F5 \vee F6$
$\wedge \quad \neg F5 \vee \neg F6$

(c) Boolean formula

Figure 2.1: Feature model, set of configurations and Boolean logic encoding

A *valid* (or legal) configuration is obtained by selecting features in a manner that respects the hierarchy and the following rules: *i)* If a parent is selected, the following features must also be selected - all the mandatory subfeatures, exactly one subfeature in each of its Xor-groups, and at least one of its subfeatures in each of its Or-groups; *ii)* propositional constraints must hold. A feature model defines a set of valid feature configurations (see Definition 1). Figure 2.1(b) displays the set of valid configurations characterized by the feature model of Figure 2.1(a).

Definition 1 (Configuration Semantics) *A configuration of an feature model fm_1 is defined as a set of selected features. $[\![fm_1]\!]$ denotes the set of valid configurations of fm_1 and is a set of sets of features.*

A feature model is usually encoded as a propositional formula, denoted ϕ, and defined over a set of Boolean variables, where each variable corresponds to a feature [99] (see Figure 2.1(c) for the propositional formula corresponding to the feature model of Figure 2.1(a)). The terms feature model and *feature diagram* are employed in the literature, usually to denote the same concept. In this manuscript, we make a distinction. We consider that a feature diagram (see Definition 2) includes a feature hierarchy (tree), a set of feature groups, as well as human readable constraints (implies, excludes). The syntactical constructs offered by such feature diagrams are not expressively complete w.r.t propositional logics. Similar to [284], we thus consider that a feature model is composed of a feature diagram *plus* a propositional formula ψ (see Definition 3).

Definition 2 (Feature Diagram) *A feature diagram $FD = \langle G, E_{MAND}, G_{XOR}, G_{OR}, I, EX \rangle$ is defined as follows: $G = (\mathcal{F}, E, r)$ is a rooted, labeled tree where $\mathcal{F}$ is a finite set of features, $E \subseteq \mathcal{F} \times \mathcal{F}$ is a finite set of edges and $r \in \mathcal{F}$ is the root feature ; $E_{MAND} \subseteq E$ is a set of edges that define mandatory features with their parents ; $G_{XOR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ and $G_{OR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ define feature groups and are sets of pairs of child features together with their common parent feature ; I a set of implies constraints whose form is $A \Rightarrow B$, EX is a set of excludes constraints whose form is $A \Rightarrow \neg B$ ($A \in \mathcal{F}$ and $B \in \mathcal{F}$).*

Definition 3 (Feature Model) *A feature model is a tuple $\langle FD, \psi \rangle$ where FD is a feature diagram and ψ is a propositional formula over the set of features $\mathcal{F}$.*

2.1.1 Composing and decomposing feature models

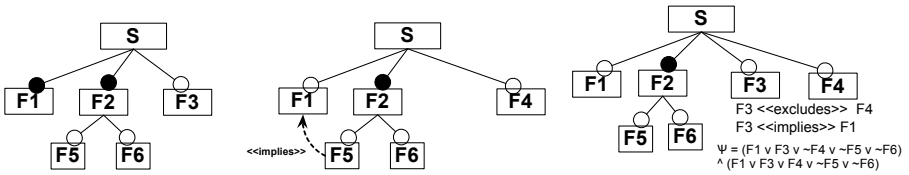
In an increasing number of scenarios, support for (de-)composing models and their variability is becoming more and more crucial [16, 68, 72, 261, 139, 140, 69, 163, 11, 265, 83, 155, 278].

Multiple systems When a multitude of subsystems (modular systems such as software services) or artifacts must be combined, several variability descriptions are to be related, organized and finally composed to form a consistent result. This context of use is broad, with first needs on organizing several *software product lines (SPLs)* with shared variabilities [72], evolving to compositional SPLs [68], in which a complex domain is captured and organized [155] into multiple SPLs [140, 265, 307] with relations between input product lines' variability models. Handling these relations lead to both reasoning about the represented configuration sets and maintaining a understandable organization (*i.e.*, a feature hierarchy). However these various usages necessitate different interpretations of the feature model composition operation to reflect the captured variable assets.

Multiple stakeholders Together with multiple product lines comes the need to handle different stakeholders on one or several SPLs. Researchers have developed techniques for feature models that reflect organisational structures and tasks. For example, Reiser *et al.* [261] address the problem of representing and managing feature models in SPLs that are developed by several companies in the automotive domain. Several feature models are used and structured hierarchically, so that they can be managed separately by suppliers. The feature model composition is then concerned with the propagation of local changes through the hierarchy. In a similar situation, Hartmann *et al.* [139] used a feature model in the context of multiple SPLs supporting several dimensions. It requires the definition of a merging process for feature models during their pre-configuration.

Multiple perspectives The need for reasoning on feature model compositions while manipulating a consistent feature model hierarchy is also emphasized by the *separation of concerns* on variability models. With their increasing complexities and usages, practitioners may define different viewpoints according to different criteria or concerns. The most used viewpoints are the ones defining the user-oriented view (external variability) from the technical features (internal one) [251]. These views have many usages [163, 162, 279], *i.e.*, defining abstraction layers, reflecting organizational structure with specific stakeholders [207], supporting collaborative design [214] or multi-level staged configurations [96]. For instance, multiple feature models are used when modelling variability of the software and the context where the software is executed (see example in Section Learning contextual variability models or our contributions with dynamic SPLs [18, 12]). Separation of these views (or concerns) also means that some relations and compositions must be done at some point to reason over the whole SPL, with references, constraints [14], a reduced form of composite model, and even in a semi-automatic way to synthesize an integrated model [16, 265].

As a result, several modelling artifacts, each coming with their own variability and possibly developed by different stakeholders, should be combined together. Our contributions are: i) the identification of composition mechanisms and semantic properties for building more complex composition-based operators on feature models; ii) the development of four possible variant implementations of such composition-based operators; iii) a reading framework to help select the right composition according to qualities and representative scenarios.



(a) fm_2 (b) fm_3 (c) $fm_{m_{union}}$

Figure 2.2: $fm_{m_{union}}$ is a possible composition of fm_1 (of Figure 2.1(a), page 21), fm_2, and fm_3

Meanings of Composition-based Operators

We first show that several salient variants of composition operators can actually be defined, depending on the syntactic and semantic properties expected in the composed feature model.

A first illustrative example. Let us consider the composition of fm_1, fm_2 and fm_3 (see respectively Figure 2.1(a), Figure 2.2(a) and Figure 2.2(b)). We denote by $\circ$ a composition operator over feature model that computes a new feature model. In our specific example, we consider that the composed feature model, denoted $fm_{m_{union}}$, should represent the *union* of input sets of configurations of fm_1, fm_2 and fm_3, that is: $[\![fm_{m_{union}}]\!] = [\![fm_1]\!] \cup [\![fm_2]\!] \cup [\![fm_3]\!]$. Such a composition is typically used to build a new SPL offering all the possible configurations supported in at least one of the products or SPLs of an organization or a supplier. Two possible resulting feature models are depicted in Figure 2.2(c) and Figure 2.3. Intuitively, when features are selected in the composed feature model, it means that the selection of corresponding features (*i.e.*, with the same names) is valid in either fm_1 or fm_2 or fm_3. For instance, a partial configuration involving the selections of features F1, F2, and F3 is valid in $fm_{m_{union}}$ since the combination of features F1, F2, and F3 is also valid in fm_2. However it is not possible to both select features F3 and F4 in $fm_{m_{union}}$ since no valid configurations of fm_1, fm_2 and fm_3 support this combination.

Meanings. Obviously, the semantics of the previous composition can be in contradiction with the intentions, requirements or simply modelling objectives of a practitioner. First there are different ways of interpreting the way features *match* and are related to each other (*e.g.*, the mapping is not necessarily one-to-one). Second the *configuration semantics* expressed in the composed feature model may differ (stakeholders may want to compute the intersection, the reduced product, the difference, *etc.* of configuration sets instead of the union). Finally the conceptual organization of the features in the resulting feature model is another variation. Due to the variety of compositional scenarios exposed in the introduction, there is no one-size-fits-all interpretation when feature model have to be composed. In order to address the variations' meanings, we identify common mechanisms and present a generic framework to devise (new) composition-based operators.

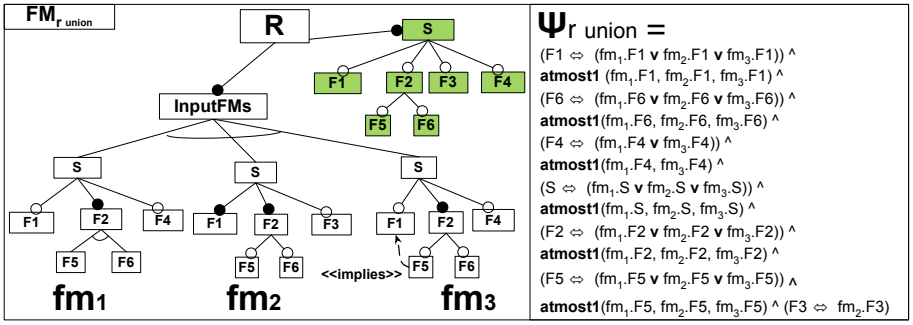


Figure 2.3: Composition of fm_1, fm_2, and fm_3, somehow equivalent to $fm_{m_{union}}$. The term atmost1 $(F_1, \ldots, F_n)$ is equivalent to $\wedge_{i<j}(\neg F_i \vee \neg F_j)$

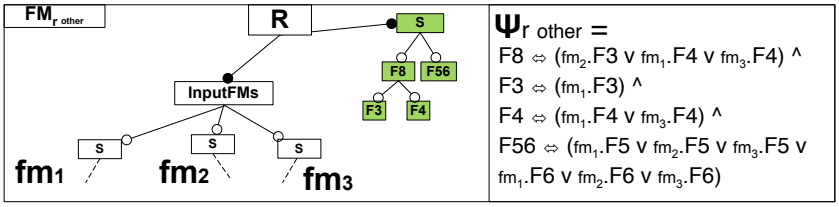


Figure 2.4: Another composition of fm_1, fm_2, and fm_3 with different matching/merging strategies and semantic properties

Different Strategies for Matching and Merging The previous strategy for matching/merging feature models is rather basic and straightforward: features match if they have the same names while the merging consists in simply creating new features with the same names S, $F1$, $\ldots$, $F6$. However more sophisticated matching and merging mechanisms are needed especially when input feature models are coming from different sources (*e.g.*, suppliers) or when the composed feature model should reflect a *view* of the system that does not necessarily include all the original details or feature names.

We give an example in Figure 2.4. Firstly, F56 is mapped to features F5 and F6 of input feature models. The intuition is that either selecting F5 or F6 is sufficient to realize the feature F56. In a sense, F56 *abstracts* features F5 and F6 since no distinction is made between F5 and F6 at the level of abstraction of the view (coloured features). Secondly, F1 is no longer present in the composed view. It is another form of abstraction: unnecessary details are removed. Thirdly another feature, named F8, is present in the view and aims to better structure the feature model, considering that features F3 and F4 are ontologically closed.

Different Semantic Properties The matching and merging mechanisms are the basics for devising a composition operator. However they do not state what are the properties of the composed feature model in terms of *configuration semantics* and *ontological semantics*. Let us consider once again the composition of fm_1, fm_2, fm_3 and assume that features F3 match in the three feature models and are merged as a new feature F3 in the composed feature models. There is still need to establish the meaning of the new feature F3 in terms of configuration, *i.e.*, what is the impact of a selection and deselection of F3 in the composed feature model?

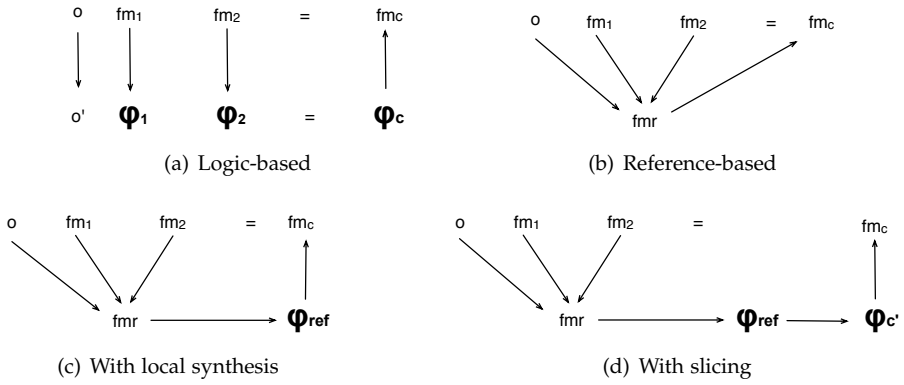
Configuration semantics A first interpretation is that the selection of F3 in the composed feature model involves the selection of F3 in *one and only one* input feature model. (It corresponds to the *union* of configuration sets as considered in the first illustrative example.) The direct impact of this specific semantics is that the selection of F3 induces in turn the selection of F1 (see Figure 2.2(c) and Figure 2.3), since there is no SPL that supports F3 without F1. Another more restrictive interpretation is that the selection of F3 in the composed feature model forces the selection of *all* features named F3 in input feature models. If this interpretation is applied on all features, the composition intuitively corresponds to the *intersection* of configuration sets. Yet another (less restrictive) interpretation is that the selection of F3 in the composed feature model forces the selection of *at least one* features named F3 in input feature models.

Ontological semantics Another important aspect of feature models is the way features are conceptually organized in the tree-based hierarchy. Given a set of configurations, there still exists different candidate feature models yet with different hierarchies [284]. Therefore what the most appropriate feature hierarchy is should be part of the composition. For instance, a practitioner may consider that the feature F3 is more appropriately located below the feature F1 than below the root S in Figure 2.3.

Variations in the Compositions of Feature Models

A composition operator $\circ$ takes as input a set of FMs and can be customized for supporting different matching/merging strategies and semantic properties (being related to configuration or ontological aspects) in the resulting FM. The following section addresses another important and related problem: How to implement these compositions? Different variants are indeed worth to consider, each having strengths and weaknesses.

Denotational-based Composition (Logic-based) The logic-based implementation consists in *i)* encoding the expected configuration set of the composed feature model as a Boolean formula ϕ_c *ii)* synthesizing the feature diagram from ϕ_c. Figure 2.5(a) summarizes the process. The first step is to compute ϕ_c. All input FMs (resp. fm_1 and fm_2) are encoded as Boolean formula (resp. ϕ_1 and ϕ_2). Then the composition operator is *denoted* (or translated) in the Boolean logic. If we consider the case of *union* (see the first illustrative example), the denotational operator roughly corresponds to a *disjunction* of formulae (details have been given in [14]). Similar denotations can be applied for computing the intersection, diff, reduced product, *etc.* of configurations sets. The second step determines an appropriate hierarchy and synthesizes variability information. First we compute the binary implication graph of ϕ_c. It is a directed graph $BIG_c = (V, E)$, V being the set of nodes corresponding to variables of the formula, while the set of edges is formally defined as $E = \{(f_i, f_j) \mid \phi \wedge f_i \Rightarrow f_j\}$. BIG_c is a representation of all logical implications between two variables in ϕ_c and corresponds intuitively to all possible hierarchies of fm_c. Second we compute a directed *minimum spanning tree (MST)* of BIG_c that maximises the parent-child relationships of input hierarchies. Finally, other components of the feature diagrams can be synthesized [32, 21]. In Figure 2.2(c), the resulting synthesized feature model corresponds to the first illustrative composition of fm_1, fm_2 and fm_3 (union mode, name-based matching strategy).



(a) Logic-based (b) Reference-based

(c) With local synthesis (d) With slicing

Figure 2.5: Variants of composition-based operator implementation

Operational-based Composition (Reference-based) Another radically different implementation is to *reference* input feature models. The key idea is to build a separated feature model (*i.e.*, a *view*) that typically contains features with the same names of the inputs. The features of the view are then related to input features through a set of logical constraints. The result is a feature model that both aggregates the input models, the view, and the constraints. Figure 2.3 depicts the resulting feature model on the same kind of composition (union) as previously considered. Other kinds of configuration semantics (*e.g.*, intersection) can be realized by defining another view and logical mapping.

The main difference is that features of input FMs are still present (*i.e.*, the merging strategy differs compared to the denotation-based implementation). Yet it is worth to observe that the configuration semantics expressed in $fm_{r_{union}}$ (see Figure 2.3) is *equivalent* to $fm_{m_{union}}$ (see Figure 2.2(c)). The equivalence is defined as follows:

$$[\![fm_{m_{union}}]\!] = [\![fm_{r_{union}}]\!]_{|\mathcal{F}_{r_{view}}}$$

where $\mathcal{F}_{r_{view}}$ is the set of features in the view (coloured features in Figure 2.3) and $A_{|B}$ denotes the projection of two given sets A and B such that: $A_{|B} \triangleq \{a' \mid a \in A \wedge a' = a \cap B\} = \{a \cap B \mid a \in A\}$. Intuitively it means that the exact same combinations of $S, F1, \ldots, F6$ are authorized in $fm_{r_{union}}$ and $fm_{m_{union}}$. This is due to $\psi_{r_{union}}$ that constrains the way features $S, F1, \ldots, F6$ of $fm_{r_{union}}$ can be combined. For instance, $\psi_{r_{union}}$ states that the selection of F2 should correspond to at least and at most one of the following features: $fm1$.F2, $fm2$.F2, or $fm3$.F2. Therefore F2 is actually mandatory in $\psi_{r_{union}}$ (as in $\psi_{m_{union}}$). The semantic operator that produces a projection of a feature model (a slice) with respect to a set of selected features (slicing criterion) is called *slicing*. (A formal definition can be found in Section Reverse engineering architectural variability models, Definition 9.)

Hybrid. The semantic equivalence of the denotational and operational-based implementations and the last remarks give the idea of going further by *correcting* the view of the reference-based FM. Two equivalent solutions are considered. In both cases, the principle is to *i)* denote the reference-based feature model as a formula ϕ_{ref} and then *ii)* synthesize a new feature diagram (see Figure 2.5(c) and Figure 2.5(d)).

Reference-based and Local Synthesis. Ideally the composed feature model only contains features of $\mathcal{F}_{r_{view}}$. However ϕ_{ref} contains many Boolean variables that may disturb reasoning procedures. For instance the computation of the implication graph is likely to contain nodes and edges that are actually not relevant. Furthermore considering all variables of ϕ_{ref} can increase the computation time when reasoning. We thus adapt the synthesis procedure so that operations are only applied *over relevant variables* ("locally"). For instance, the computation of the implication graph is achieved by the checking of implications between features of interest only. On the previous example, the synthesis of the variability information leads to the same exact feature diagram as depicted in Figure 2.2(c).

Reference-based and Slicing. Another variant is to eliminate disturbing variables in ϕ_{ref} and obtain a new formula $\phi_{c'}$. Intuitively, non relevant variables are removed by existential quantification in ϕ_{ref}.

Definition 4 (Existential Quantification) *Let v be a Boolean variable occurring in ϕ. $\phi_{|_v}$ (resp. $\phi_{|_{\bar{v}}}$) is ϕ where variable v is assigned the value True (resp. False). Existential quantification is then defined as $\exists v\ \phi =_{def} \phi_{|_v} \vee \phi_{|_{\bar{v}}}$.*

In case of union, intersection, *etc.* $\phi_{c'}$ is equal to ϕ_c (the formula obtained with a denotational-based approach), *i.e.*, the formula logically represents the exact same valid configurations and the set of variables is exactly the same. Therefore $\phi_{c'}$ can be used afterwards to synthesize an FM: the feature diagram obtained is the same as Figure 2.2(c).

A Framework for Composing your Compositions

Users of composition operators for FMs have to define a specific semantics (or reuse an existing one, *e.g.*, union, intersection) and then select an appropriate implementation (*e.g.*, reference-based and slicing). In this section, we provide a *reading grid* and *practical illustrations* in order to assist users in customizing a composition adapted to their requirements.

We now discuss and compare the pros and cons of each implementation variant.

Quality of the Feature Diagram The feature diagram (see Definition 2) can be seen as a syntactical view of the configuration set s that practitioners or automated tools usually exploit in a forward engineering phase. The following qualities are expected from a feature diagram: (1) *maximality* [99]: as much variability information as possible should be represented in the resulting feature diagram to approximate or fully represent s. For instance, the operational-based composition does not produce a maximal feature diagram: the feature F2 is optional in the feature diagram whereas it is always included in every configuration; (2) *soundness* and *completeness w.r.t.* configuration semantics: in the reference-based FM, it may happen that the retained hierarchy is not a spanning tree of the implication graph, with the consequence of either precluding some valid configurations (incomplete) or all possible configuration (unsound) – see an example in Figure 2.6 (the feature model is incomplete).

Reasoning A composition-based operator computes a feature model that can be exploited afterwards for *reasoning*, for example, when performing assisted configurations (decision verification and propagation, auto-completion, scheduling of configuration tasks, *etc.*), when automating analysis over the FMs (*e.g.*, debugging of FMs, comparison of two FMs) [55, 303]. The question we address here is: how to reason about the configurations *once* the resulting

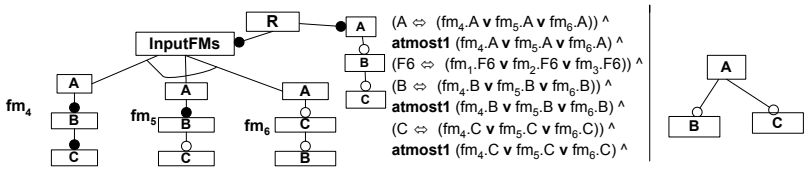


Figure 2.6: Composition of fm_4, fm_5, and fm_6 (union): in left-part, the hierarchy leads to an incomplete feature model ; in the right-part, a complete and sound FM.

	Denotational-based (Logic-based)	Operational-based (Reference-based)	Reference-based + Local Synthesis	Reference-based + Slicing
Diagram quality	A	C	A	A
Reasoning	A	C	C	A
Customizability	C	B	A	A
Traceability	C	A	A	A
Composability	A	C	B	A
Performance	OI	OI	OI	OI

Table 2.1: Comparison of approaches (A: best ; C: worst; OI: open issue)

feature model has been synthesized? The drawback of a reference-based approach is that the reasoning should be performed over (a large amount of) features that are sometimes not relevant. For instance, if we want to perform a configuration over the features F1, F2, ..., F6, it necessarily involves considering the referenced features fm_1.F1, fm_1.F2, ..., fm_3.F6.

Traceability Features are usually mapped to development artefacts, such as components, models and user documentation. The preservation of the *traceability* between the feature model and the artefacts is essential for automatic derivation of products from the configuration of the composed FM. In the case of a denotational-based technique, the mapping between the input FMs is not kept intact because they are replaced by a merged FM. As a result, the selections of features in the composed model may correspond to as many corresponding features in the input FMs.

Customizability In the previous section, we have shown that there are different mechanisms that can be customized to specify the meaning of a composition. The denotational-based strategy is the most rigid since the matching strategy is assumed to be one-to-one and based on feature names while the merging process creates a new feature with the same name. The reference-based techniques are more general since any kinds of logical mappings between *i)* the features planned to be present in the composed feature model and *ii)* the features in the input FMs can be defined.

Composability Let us consider the composition in union mode and a matching strategy based on feature names (as the example explained in Figure 2.3). The reference-based technique is neither associative nor commutative, *e.g.*, $\circ(\circ(fm_1, fm_2), fm_3) \neq \circ(\circ(fm_1, fm_3), fm_2) \neq \circ(fm_1, fm_2, fm_3)$. Though the configuration set represented is the same, the feature diagrams are different. On the contrary the denotational-based and hybrid techniques are associative and commutative (in the case of union) since the Boolean formulas obtained are the same as previously and the logical operations do have the properties.

Table 2.1 summarizes the discussions and results by classifying the *best* and the *worst* solution in a given dimension. Some implementation variants are equivalent for some criteria (*e.g.*, denotational and hybrid techniques compute the same feature diagram). The slicing-based technique fulfils all the criterion and, as such, can be considered as the most suited in the general case. Yet, its *performance* has to be confronted to other composition variants in

practical settings (with different kinds of input FMs, matching and semantic properties, *etc.*). The performance of compositional approaches is marked OI (for open issue) in the table since a quantitative evaluation has not been done. We leave it as future work since it is a *knowledge compilation problem* [101] that deserves a focused and careful attention.

Overall, depending on the variability modelling scenarios and needs (*e.g.*, modular model checking of SPLs, modelling variability of independent suppliers, engineering of configurators, hierarchical SPLs), there are tradeoffs to consider when choosing the composition approach. Existing works [261, 140, 14, 15, 326] can benefit from the new proposed techniques when reasoning, aligning feature models or simply devising new composition-based operators.

Iterative generalization

A few comments on this line of work. Firstly, I consider this contribution nicely generalizes our past attempts [14, 15, 13, 17] on composing and decomposing feature models. Retrospectively, prior efforts seek to:

- define the right meanings *i.e.*, what does it mean to compose feature models? The answer was not straightforward since, as our research has shown, many semantics can actually be defined, with different properties of the resulting models. Hence it is not surprising we did not anticipate all possible semantics in the first place (*e.g.*, in our seminal work about composing feature models [15]);

- motivate the use of multiple feature models: the need to support composition and decomposition (slicing) of large feature models was not crystal clear when we started our research. The concepts of multiple SPLs or dynamic SPLs were either not defined or quite novel at that time. As a result, there are now more and more papers that report on similar composition or decomposition needs, in many engineering contexts;

- implement operators for feature models: we first tried with rule-based approaches, but quickly found limitations. The use of logics has been eye-opening and here I would like to highlight the strong influence of seminal papers [99, 43] about logics and feature models.[1]

- apply feature modelling in realistic settings: we have tried to use feature models in the medical imaging, grid computing, and video processing domains. This experience was key for the points previously mentioned and helped us to refine the semantics and implementations of the support.

Secondly, we have developed other operators for feature models, like the diff (for difference) or the refactoring of feature models [21, 22]. Such operators can be combined with composition and decomposition operators. In a sense, we have developed a practical algebra for feature models. Also, we further generalize the definition of operators with an ontology-aware solution [47]. This solution synthesizes sound and complete feature models

[1] Little anecdote here: I met Krzysztof Czarnecki at the SLE'09 conference, a bit randomly (at lunch), after the presentation of M. Acher, P. Collet, P. Lahire and R. France, 'Composing Feature Models', in *2nd International Conference on Software Language Engineering (SLE'09)*, ser. LNCS, LNCS, Oct. 2009, p. 20. Krzysztof suggested that I look at the synthesis work described in his paper K. Czarnecki and A. Wasowski, 'Feature diagrams and logics: There and back again', in *SPLC'07*, 2007. This nice hint did not give a full solution, but definitely pushed me in the right direction. Would that happen with a virtual conference?

while taking feature hierarchies into account. Thanks to ontological heuristics, it is applicable without prior knowledge or artefacts, and can be used either to fully synthesize a feature model or guide the user during an interactive process. I do not detail this contribution [47] here, but will get back to it in reverse engineering scenarios (*i.e.*, in Chapter Reverse Engineering Software Variability).

Thirdly, the explanations made in this section may seem abstract and not useful in practice. I use on purpose abstract feature names (*e.g.*, F1, F56) in order to focus on salient properties of resulting feature models. However, I want to highlight the fact that these operators have been used in different application domains and kinds of systems. Some examples are given in the next sections or in related contributions.

replication

The tutorial https://github.com/FAMILIAR-project/familiar-documentation/blob/master/manual/composition.md contains executable scripts, written in FAMILIAR (see below), to replay the different (de-)composition scenarios. The ontologic-aware synthesis procedure as well as experiments are also available https://github.com/gbecan/FOReverSE-KSynthesis.

2.1.2 FAMILIAR, a language for combining feature model operators

Our previous research addressed the management of feature models mainly from a theoretical perspective. Yet, a practical solution is still missing for importing, exporting, composing, decomposing, manipulating, editing and reasoning about FMs. It is especially important for modelers since numerous feature models and management operations have to be combined in practice.

We have developed FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning) [16]. FAMILIAR is a domain-specific language with a textual syntax. The language includes facilities for performing operations over multiple feature models and their configurations. Two reasoning back-ends (SAT solvers using SAT4J and BDDs using JavaBDD) are internally used over propositional formulae. FAMILIAR is used in different teaching contexts, for conducting research, or for collaborating within projects or with industry. FAMILIAR can be used: i) with an executable JAR and an interactive REPL; ii) with the source code and a Java API; iii) online through a Web application; iv) inside Eclipse with a REPL; v) within a Docker.

The code snippet below illustrates how to use the four implementation variants on the illustrative example of the previous subsection. It is a simplified excerpt of the companion, online page [89] that provides a comprehensive tutorial and numerous examples.

The script first loads three feature models $fm1$, $fm2$ and $fm3$ (the rest of the script will be explained in detail hereafter). It should be noted that $fm1$, $fm2$ and $fm3$ actually correspond to feature models used in the paper H. Hartmann, T. Trew and A. Matsinger, 'Supplier independent feature modelling', in *SPLC'09*, IEEE, 2009, pp. 191–200 and in the previous subsection (see Figure 2.1(a), Figure 2.2(a), and Figure 2.2(b)). We use a specific, textual notation for specifying feature models, inspired by guidsl [43] and grammar notation. Other formats are supported for importing feature models. FAMILIAR is a DSL for managing feature models – the scope is beyond the sole specification of variability models.

```
1  /**** input feature models ****/
2  // F1, F4 optional, F5 and F6 mutually exclusive
3  fm1 = FM (S : F1 F2 F4 ; F2 : (F5|F6) ; )
4  // F5, F6 still subfeatures of F2 but optional, F1 mandatory
5  fm2 = FM (S : F1 F2 F3 ; F2 : F5 F6 ; )
6  // F5 implies F1
7  fm3 = FM (S : F1 F2 F4 ; F2 : F5 F6 ; F5 -> F1 ; )
8
9  // logic-based
10 fmMUnion = merge union { fm1 fm2 fm3 }
11 // reference-based
12 fmRUnion = aggregateMerge union { fm1 fm2 fm3 }
13
14 // basic, syntactic extraction (features are all optionals)
15 fm6 = extract fmRUnion.S
16
17 // slicing (same FD + formula than fmMUnion)
18 fm7 = slice fmRUnion including fmRUnion.S*
19 assert ((compare fm7 fmMUnion) eq eq REFACTORING)
20
21 // local synthesis (same FD but formula differs)
22 fm8 = ksynthesis fmRUnion over fm5.S*
23 assert ((compare fm8 fmMUnion) eq eq GENERALIZATION)
```

Back to our script, we want to compose the three feature models. In particular, we are targeting the following scenario. fm1, fm2 and fm3 are representing three product lines maintained by three different suppliers. We want to build a new product line offering every possible product offered by suppliers (no more, no less). In terms of feature modelling, we want to compute a new feature model (a composed feature model) that represents the *union* of input sets of configurations.

Merge (logic-based) Line 10 implements the **merge** operation with a logic-based approach. The principle is to encode each feature model as a formula and then compute a "composed" formula. This can be achieved by *denoting* into the Boolean logic the configuration semantics (*e.g.*, of "union"). A hierarchy is then automatically selected and variability information is synthesized [6]. The result $fmMUnion$ corresponds to the feature model depicted in Figure 2.2(c), page 23.

Reference-based approach. Another approach is to reference input feature models. A composed view, containing the features that are of interest for the composition, is specified. Then features of the view "reference" features of input feature models through (logical) constraints.

To implement a reference-based approach, the following steps are: (1) definition of the composed view; (2) definition of the set of constraints to establish correspondences with input features aggregate the composed view, (3) definition of the constraints as well as the input feature models together. However, the constraints to be specified can be huge, thus making the process time-consuming and error-prone. Another (more technical) problem is that the existing **aggregate** operator of FAMILIAR assumes that features' names are unique (in order to make a distinction between features).

Therefore we develop and provide another operator to automate this task, called **aggregateMerge** (see Line 12). It takes as input a set of feature models and a "mode" (*e.g.*, union, intersection). The operator works as follows. First, a new feature model, playing the role of the *view*, is created. The features of this view are the union of all features of input feature models, assuming that features are equals whenever they have the same name. Furthermore, all these features are syntactically optional. Second, the input feature models are copied and aggregated with the new view into a composed feature model. Features' names are automatically rewritten to guarantee unicity *e.g.*, to avoid having four features named *S*. Finally, constraints between features of the view and features of input feature models are added.

The interest of the aggregated feature model, here $fmRUnion$, is to obtain a semantically equivalent representation of the set of configurations. Indeed, the combinations of features authorized in the composed view of $fmRUnion$ are exactly the same as in $fmMUnion$, thus realizing the configuration semantics of union. It is straightforward to understand why: the constraints force the selection of at least one and at most one corresponding feature (if any) in the input feature models. However, there is another remaining problem: features are all optional in the view. In the code snippet, the extracted view $fm6$ is a syntactical copy of the subtree rooted at feature *S*. As a result, you cannot use the view ($fm6$) independently. Furthermore the view contains anomalies, since the syntactical constructs are not conformant to the actual meaning. In particular, F2 is modeled as optional but is actually mandatory ; F3 and F4 are mutually exclusive ; F3 implies the selection of F1. It is clearly not the case in $fm6$. It is a very rough over-approximation of the actual configuration set. Therefore it not acceptable to exploit $fm6$ afterwards.

Two techniques are conceivable for improving the quality of the "view".

With slicing. A first one is to rely on the **slice** operator. It takes as input a feature model and a set of features (slicing criterion). It computes a new feature model containing only the features of the slicing criterion. Importantly the resulting feature model characterizes the projected set of configurations onto the slicing criterion. In our case, we are only interested by features of the view (see green features of Figure 2.3, page 24). In line 18, we slice $fmRUnion$ (resulting from the **aggregateMerge** operator) with $fmRUnion.S*$ that returns feature *S* as well as all features below *S* (*i.e.*, all green features of Figure 2.3). Intuitively, the projected set of configurations is the same set of configurations of the input feature model without the features not in the slicing criterion. Obviously, the **slice** operator is not internally implemented like this. Enumerating all configurations and then computing the projected configurations with set operations is not efficient and scalable even for small feature models. Instead a Boolean formula, free for non relevant variables, is computed. Then satisfiability techniques are used to synthesize a comprehensive feature model [6]. In the example, a key benefit is that the computed feature model $fm7$ is exactly the same as $fmMUnion$. That is, the formulas are the same as well as the synthesized feature diagram.

With local synthesis. Another strategy for correcting the "view" of $fmRUnion$ is possible. Instead of computing a new Boolean formula (like with the slice, see above), we can directly exploit the original formula of $fmRUnion$ and perform reasoning over the relevant Boolean variables. We use the operator **ksynthesis** that can perform over the set of features specified (see Line 22). The local synthesis procedure makes it best for producing a maximal *feature diagram*. But it is well-known that feature diagrams offer syntactical constructs that are not expressively complete *w.r.t.* Boolean logics (see Definition 3). It is the case of $fm8$ that authorizes two configurations not valid in $fmMUnion$ (*i.e.*, $fm8$ is a "generalization" of $fmMUnion$).

replication

The repository https://github.com/FAMILIAR-project/familiar-language contains the source code of the language including the grammar, Java API and interpreters. There are many scripts and Java test cases to demonstrate the usage of FAMILIAR.

2.2 Feature models and product comparison matrices

The content of this section is adapted from the following publications:
N. Sannier, M. Acher and B. Baudry, 'From Comparison Matrix to Variability Model: The Wikipedia Case Study', in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, Palo Alto, USA, 2013. http://hal.inria.fr/hal-00858491
G. Bécan, N. Sannier, M. Acher, O. Barais, A. Blouin and B. Baudry, 'Automating the formalization of product comparison matrices', in *ASE*, 2014

On the one hand, a feature model is an abstraction to compactly represent a set of related products. On the other hand, tabular data – and more specifically what we call product comparison matrices – can be used to document a set of related products *w.r.t.* different criteria (*e.g.*, features supported or not). Are feature models and product comparison matrices two sides of the same coin? It is quite natural to question the relationship between the two formalisms from a syntactic and semantic point of view. In this contribution, we address the following question: Is feature modelling an adequate formalism to represent variability information encoded in product comparison matrices? We also seek to understand product comparison matrices themselves, independently from traditional variability models like feature models.

Product Comparison Matrices (PCMs)

An example of an PCM is depicted in Figure 2.7(a). PCMs are specific tabular data. Specifically, PCMs provide a simple and convenient way to express properties on products and *compare* them to several different others from the same *family*. Companies use PCMs to present and advertise different *facets* of their *product series*, typically through comparators [272]. PCMs provide a global view of several different competing products, showing the presence, absence, limitations of a facet, expressing *commonality and variability* between products under comparison.

We first start with an empirical study in order to understand what and how variability information is expressed within PCMs. We consider the Wikipedia case study. Wikipedia manages one of the most important source of PCMs from various domains and for different kinds of products.

We introduce an illustrative example mined from Wikipedia and propose a first set of observations on variability information found in these PCMs. Then we evaluate the observations against a systematic analysis of 50 PCMs and perform a second automatic analysis on a larger set of PCMs.

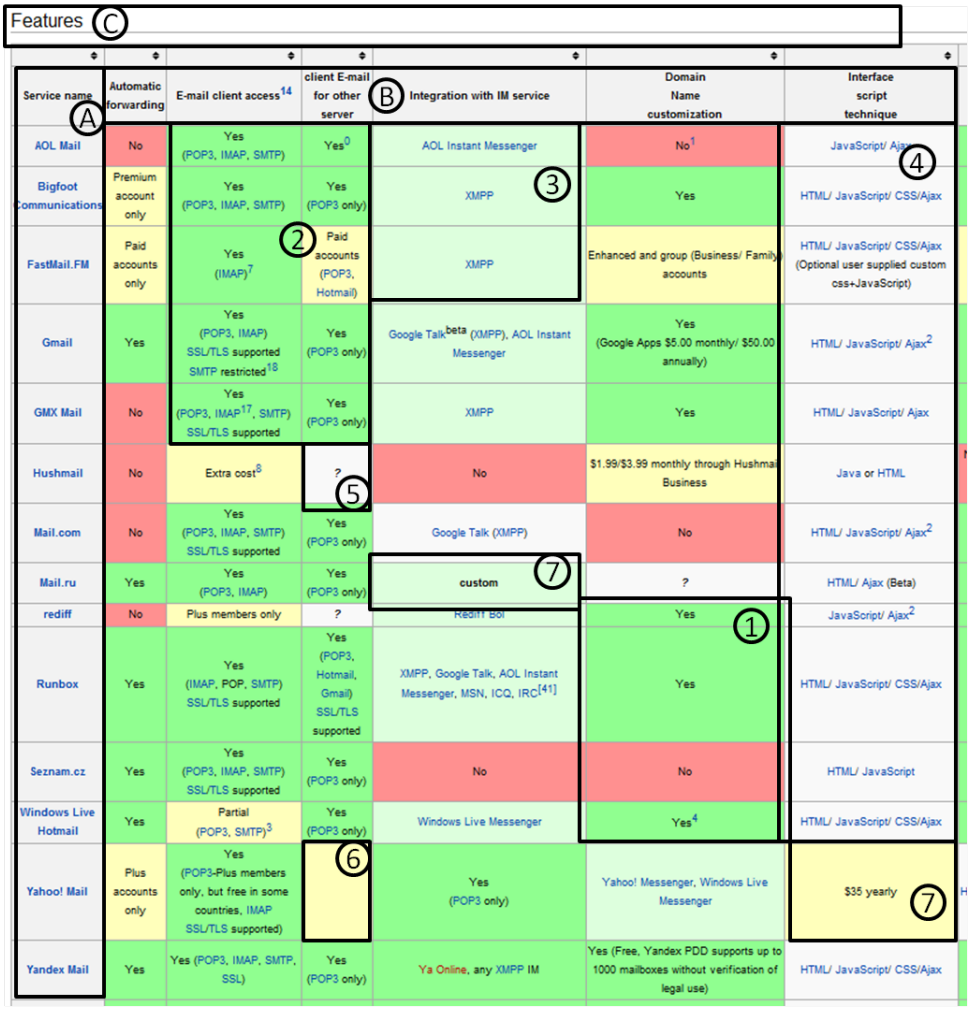
Anatomy of a Wikipedia PCM: a First Example

We analyze a PCM about webmail providers mined from Wikipedia[2] and present a sample of the PCM in Figure 2.7(a). This PCM compares 15 different products (Ⓐ in the figure) against 12 different criteria (Ⓑ in the figure). This Wikipedia page also proposes different comparison perspectives (Ⓒ) and, consequently, several PCMs related to these perspectives. However, our example focuses on the PCM of figure 2.7(a), which includes 180 different cells to analyze.

The first observation we make is related to the different comparison criteria, found as headers of the PCM. A PCM is composed of a list of heterogeneous criteria with different levels of precision and flexibility. Consequently, product values regarding these criteria can be of various kinds such as:

- ① **Boolean yes/no values**. This kind of variability deals with the straight, non ambiguous, presence or absence of the comparison criteria. We observe that pairs of tokens like "yes/no", "true/false", *etc.* are potential candidates for this kind of variability information.
- ② **Constrained/Partial/ambiguous yes/no values**. This kind of cell has to be interpreted as: "the criterion is satisfied under the condition of, with the following limitation, etc"."Only", "if", "through", can be candidate words to recognize this kind of value. The token "partial" is the most significant evidence of the presence of the value type. One can also see a "yes" with a footnote or followed by one or several elements that express a condition or limitation.
- ③ **Single-value**. This kind of information has to be interpreted as: "the criterion is satisfied using this element". It forms a unique way to satisfy the criteria. The purpose of this information is not to know whether or not the criterion is satisfied but how.
- ④ **Multi-values**. This kind of information has to be interpreted as: "the criterion is satisfied using these elements". It forms a set of elements that contributes to satisfy the criterion. It should be noted that there is no homogeneity, within the same matrix, in the way of expressing such enumerations. A same product can be delivered with all of these elements, or deliver different versions for each element.
- ⑤ **Unknown value**. One does not know if the criterion is satisfied. Cells are generally filled with "?", "unknown". This information is rather hard to manage. It cannot be fully interpreted as a boolean "no" answer, as it can prevent the product from being selected, despite the domain reality that is unknown.

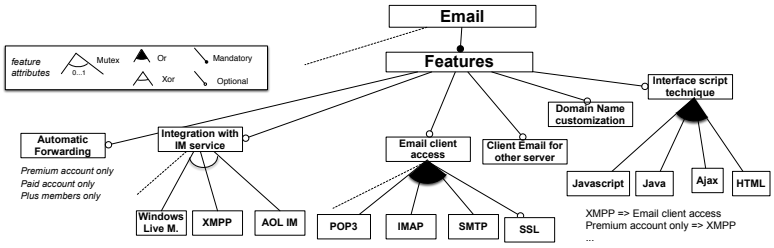
[2]Available online at http://en.wikipedia.org/wiki/Comparison_of_webmail_providers, last access 10th may 2013



Features (C)

Service name (A)	Automatic forwarding	E-mail client access[14]	client E-mail for other server (B)	Integration with IM service	Domain Name customization	Interface script technique
AOL Mail	No	Yes (POP3, IMAP, SMTP)	Yes[0]	AOL Instant Messenger	No[1]	JavaScript/ Ajax (4)
Bigfoot Communications	Premium account only	Yes (POP3, IMAP, SMTP)	Yes (POP3 only)	XMPP (3)	Yes	HTML/ JavaScript/ CSS/Ajax
FastMail.FM	Paid accounts only	Yes (IMAP)[7] (2)	Paid accounts (POP3, Hotmail)	XMPP	Enhanced and group (Business/ Family) accounts	HTML/ JavaScript/ CSS/Ajax (Optional user supplied custom css+JavaScript)
Gmail	Yes	Yes (POP3, IMAP) SSL/TLS supported SMTP restricted[18]	Yes (POP3 only)	Google Talk[beta] (XMPP), AOL Instant Messenger	Yes (Google Apps $5.00 monthly/ $50.00 annually)	HTML/ JavaScript/ Ajax[2]
GMX Mail	No	Yes (POP3, IMAP[17], SMTP) SSL/TLS supported	Yes (POP3 only)	XMPP	Yes	HTML/ JavaScript/ Ajax
Hushmail	No	Extra cost[8]	? (5)	No	$1.99/$3.99 monthly through Hushmail Business	Java or HTML
Mail.com	No	Yes (POP3, IMAP, SMTP) SSL/TLS supported	Yes (POP3 only)	Google Talk (XMPP)	No	HTML/ JavaScript/ Ajax[2]
Mail.ru	Yes	Yes (POP3, IMAP)	Yes (POP3 only)	custom (7)	?	HTML/ Ajax (Beta)
rediff	No	Plus members only	?	Rediff Bol	Yes (1)	JavaScript/ Ajax[2]
Runbox	Yes	Yes (IMAP, POP, SMTP) SSL/TLS supported	Yes (POP3, Hotmail, Gmail) SSL/TLS supported	XMPP, Google Talk, AOL Instant Messenger, MSN, ICQ, IRC[41]	Yes	HTML/ JavaScript/ CSS/Ajax
Seznam.cz	Yes	Yes (POP3, IMAP, SMTP) SSL/TLS supported	Yes (POP3 only)	No	No	HTML/ JavaScript
Windows Live Hotmail	Yes	Partial (POP3, SMTP)[3]	Yes (POP3 only)	Windows Live Messenger	Yes[4]	HTML/ JavaScript/ CSS/Ajax
Yahoo! Mail	Plus accounts only	Yes (POP3-Plus members only, but free in some countries, IMAP SSL/TLS supported)	(6)	Yes (POP3 only)	Yahoo! Messenger, Windows Live Messenger	$35 yearly (7)
Yandex Mail	Yes	Yes (POP3, IMAP, SMTP, SSL)	Yes (POP3 only)	Ya Online, any XMPP IM	Yes (Free, Yandex PDD supports up to 1000 mailboxes without verification of legal use)	HTML/ JavaScript/ CSS/Ajax

(a) Example of a Wikipedia Product Comparison Matrix



(b) A possible corresponding feature model of the PCM (excerpt)

Figure 2.7: A family of online emails products: PCMs and feature models

- ⑥ **Empty cell**. This information is hard to interpret, *i.e.*, whether it should be analyzed as a strong boolean "no" and accordingly assessed as the absence of the feature or should this be analyzed as an unknown answer ?
- ⑦ **Inconsistent value**. The provided value is partial, ambiguous or lightly related to the analyzed criterion. For instance, in Figure 2.7(a), it is mentioned that "Yahoo! Mail" has a "$35 yearly" interface, whereas all other products mention the underlying technology of their interface.
- ⑧ **Extra Information**. The provided cell value offers additional information such as latest dates, versions. Though not present in Fig 2.7(a), this pattern exists.

It should be noted that the eight information types defined above are not necessarily expressed in a regular way for a given criteria/header. Specifically, a same header can refer to a specific value for one product, be unknown for another one, or conditionally active in another case, *etc.* An example is given for the header Client access for email Server (see Figure 2.7(a)).

Further remarks. Beyond the eight information types defined above, we report some qualitative observations. *Colors* of the cell in the PCM have a specific meaning, sometimes undocumented or even non apparent from a user perspective. In our example, "yes" values seem to correspond to green color, "no" values to red colors and "partial" values to yellow colors. This meaning is not explicit neither is systematic over PCMs. Colors can mean more than expected. For instance, software licences of a product may be documented through a specific value (*e.g.*, LGPL or Apache license) complemented with a color. Here the color aims to characterize the kind of software licence (free or proprietary). This kind of information is usually available in the source code of a Wikipedia page.

footnotes are also worth considering. They may influence the meaning of a cell value, *e.g.*, restricting the validity of a cell value to particular conditions. This makes the PCM information a bit more scattered and ambiguous.

A Qualitative Analysis of 50 Wikipedia PCMs

We want to further confirm our intuition over PCM contents. For this purpose, we analyze a sample of 50 Wikipedia PCMs. We selected the sample according to the following steps:

- We extracted all the pages from Wikipedia using the following search criterion: the page title must contain the following words substrings "comparison" or "comparison of", "comparison between". We retrieved 381 Wikipedia pages;
- We then analyzed the retrieved pages and rejected the ones that did not contain any comparative table and that were not relevant to our study. We kept 300 "relevant" pages from various domains, including economy, linguistic, technology, defense, *etc.*;
- We classified the set of candidate pages according to the number of comparison criteria they have: [1-10],[11-20] ..., [91-100], and [100+] and obtained a PCM distribution. When a comparison page contains more than one table, we consider a page with merged tables with the addition of all criteria, and a maximum value when looking at the products;
- We sampled the candidate set and randomly picked 50 Wikipedia PCMs according to the distribution to have a representative state of practice of PCMs in Wikipedia;
- We manually assessed the 50 retained pages using our catalog of 8 value types.

Table 2.2: Value types frequencies for 300+ Wikipedia PCMs

	①	②	③	④	⑤	⑥
amount	111309	1788	45903	33922	16823	15279
%	49.4	0.8	20.4	15.1	7.5	6.8

We provide some information about the 381 pages we automatically retrieved. More than half of the pages have between 1 and 20 criteria (Families 1 and 2). There exist very large PCMs. 17 PCMs have more than 100 criteria. The largest comparison page is the "Comparison of Nvidia graphics processing units" with 55 different tables for a maximum of 64 products under comparison and a total amount of 1387 criteria. 11 analyzed pages contain over 1000 cells, which make these pages *understandability and usage* even harder.

More detailed information about the 50 Wikipedia pages is available online at https://github.com/FAMILIAR-project/familiar-documentation/tree/master/manual/Wikipedia. These 50 pages contain 165 tables and about 29500 different cells. The 50 pages mainly deal with computer systems, architectures, programming at various levels but also include topics like linguistic, mechanics, politics, defense, among others.

We observe a large variety of value type frequencies at the individual page level and family level (① varies from 21.02% to 73.13%, ⑤ varies from 2.54% to 27.66%). This is due to the large variety of comparison criteria and their level of precision. This heterogeneity also reflects Wikipedia's diversity in terms of domains, collaborative authors, *etc.*

Concerning "uncertainty", information that is not a straightforward variability information (②, ⑤, ⑥, ⑦, and ⑧), it represents a mean of 25.6% per page. It is a significant number of cells that cannot stand as-is in a feature model. On the other hand, around 75% of PCMs content is rather direct information and allow a direct mapping to feature models.

A Quantitative Analysis of 300+ Wikipedia PCMs

To gain further statistical evidence about the frequency of the eight patterns, we implemented an automated extraction process for operating over 300+ Wikipedia pages. We used the state of the art parser *Sweble* [111] to process the source of each Wikipedia page. In addition, we implemented automated techniques to recognize the pattern of a cell value, following the observations of the qualitative study. We do not seek to automatically detect patterns ⑦ and ⑧ since they are mainly based on human perception. In total, we analyzed 31097 products and 225024 cell values. The results are reported in Table 2.2.

We now compare the results with those previously obtained in the qualitative study. The frequency of Boolean values has slightly increased (49.4 *versus* 47.3) and still important, confirming the importance of the pattern ① . Similarly, the frequency of single values (pattern ③) remains important (slight decrease with 20.4 *versus* 22.75). The frequency of multi-values ④ has increased to a large proportion (15.1 *versus* 4.37). We can hypothesize that part of the values can actually belong to pattern ⑦ or ⑧ (two patterns we do not detect and that are usually constituted of multiple values). The frequency of pattern ② has decreased significantly (0.8 *versus* 3.71) but still constitutes a minor pattern.

The most important result is that we confirm patterns ① and ③ are by far the most widely used, constituting almost 75% of the content of PCMs.

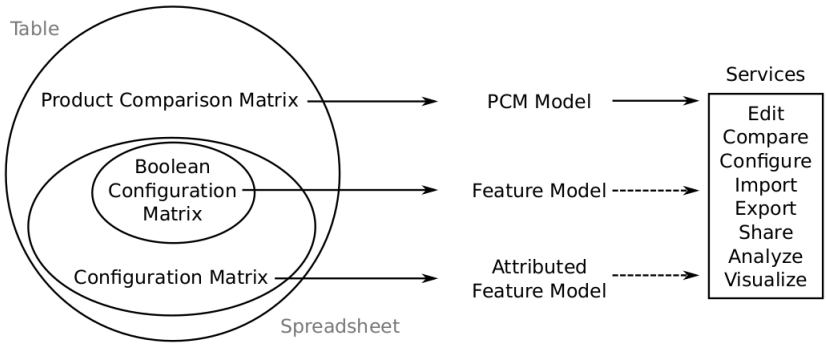


Figure 2.8: Modelling product comparison matrices

Summary and implications. We proposed a catalog of 8 kinds of variability information and report on qualitative as well as quantitative insights that help understanding the gap between PCMs and feature models. This study shows that PCMs potentially provide lots of rich and useful information but present many drawbacks such as lack of formalization, lack of tool support and understandability. One possibility to tackle these concerns is to translate PCMs into feature models, giving a clear semantics and enabling the automatic analysis of a family of products. Around 75% of PCM content can be directly translated to Boolean-based feature models but the handling of numeric attributes or uncertainty requires more effort to fit with the current state of practice of PCMs. Figure 2.8[3] summarizes the situation. Specific PCMs can be encoded with (attributed) feature models, but not all of them. As feature modelling does not cover all variability constructs needed to encode any PCM found in the wild, another modelling approach is needed. Based on these results, we have followed two research directions:

- automatic synthesis of (attributed) feature models out of PCMs: Section Synthesizing attributed feature models out of tabular data in the next Chapter Reverse Engineering Software Variability details our contributions;

- the development of a PCM metamodel capable of encoding all variability constructs of PCMs: the next subsection will detail this direction.

Formalizing product comparison matrices

Though apparently simple, synthetic, or easy to design, PCMs hide, in *practice*, an important complexity while expressing commonalities and variabilities among products. PCMs can be seen as a special form of spreadsheets and thus share some of their problems [148, 240, 4, 151, 153, 93, 75, 150, 152, 92, 91, 3, 80, 119, 113, 219]. Specifically, the underlying reasons of the complexity of PCMs are: (1) ambiguity: PCMs are mainly written in uncontrolled natural lan-

[3] All credits go to Guillaume Bécan who has realized this figure as part of the concluding chapter of his PhD thesis.

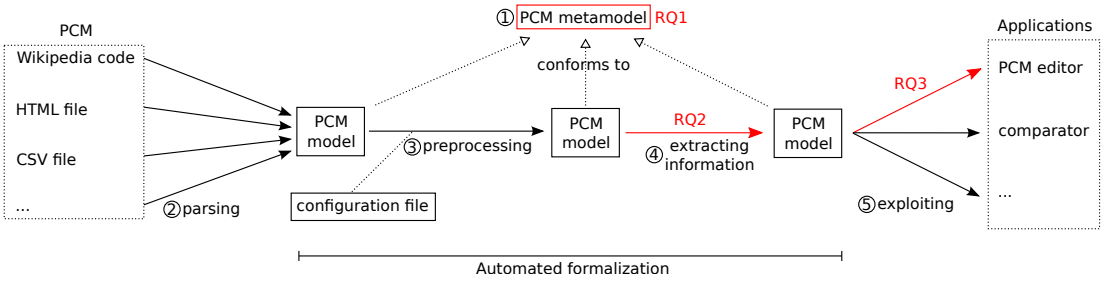


Figure 2.9: Overview of the automated formalization of PCMs

guage, mixing scopes, granularity, and heterogeneous styles; (2) scalability: as a PCM grows, its readability dramatically decreases; (3) equality: all the information is equally presented as textual assets; (4) lack of support and services: the previous points are emphasized by the limited number of advanced services tackling these limits.

The problem impacts three kinds of users: i) data writers (*e.g.*, Wikipedia contributors): how to add a new product entry when everything around is heterogeneous and there is no standard way to edit data? How to give a proper structure and semantics to the data? ii) developers (*e.g.*, data scientists or product analysts) in charge of processing, transforming, and analyzing PCMs; iii) end-users that want to understand or interact with PCMs.

A systematic engineering approach with dedicated tools is needed to improve the current practice of editing, maintaining, and exploiting PCMs. The general problem of transforming raw data to formal models has a long tradition [230]. There are two key challenges:

- **metamodelling** (1) to abstract from heterogeneity, reinforce structure and give a clear semantics to data; (2) to enable the specification of raw data transformation into models conformant to a metamodel;

- **model transformations** to automatically (1) parse and encode data into a suitable format, despite heterogeneity (2) normalize data to give a proper semantics (3) devise new editing tools.

The research problem we address in this contribution can be summarized as follows: *How to automate the encoding of heterogeneous, ambigous and large-scale data of PCMs into well-structured, well-typed, well-formalized models?*

An Iterative Process Driven by Data, Users and Services

Figure 2.9 provides an overview of our approach, from raw data of PCMs (*e.g.*, as expressed in Wikipedia) to models conforming to a metamodel.

We first propose a PCM metamodel (see ① of Figure 2.9). Metamodels provide a definition for the main concepts of a domain and their properties as well as the organization of these concepts by providing a set of associations. Using this metamodel, we can develop a transformation chain (see ②, ③, and ④ of Figure 2.9) for producing PCM models. To gather feedback from end-users, we use an editor dedicated to PCMs and built on top of our metamodel. The editor allows to present the concrete syntax of the PCM models to the evaluator instead of directly presenting the metamodel or a model in graph notation (*e.g.*,

the object diagram notation of UML). The main advantage of using the concrete syntax is to display a familiar view of a PCM. As such, there is no need for modelling knowledge to check that our formalization is correct. The feedback gathered can be used to refine our metamodel and transformation through an iterative process.

The PCM Metamodel. Figure 2.10 presents the PCM metamodel we defined as our unifying canvas. This metamodel was obtained while observing various PCMs either on the internet or real ones in magazines or shops and discussions all along the past year.

This metamodel describes both the structure and the semantics of the PCM domains. In this metamodel, PCMs are not individual matrices but a set of different matrices that contain cells. This happens when comparing a large set of products or features. In order to preserve readability, PCM writers can split the PCM content into several matrices. Cells can be of 3 types: *Header*, *ValuedCell*, and *Extra*. Header cells identify products or features.

On the semantic side, PCM express commonalities and differences between products. As a consequence, formalizing such domains necessarily requires introducing some concepts from the variability and product line engineering community but also introducing new ones.

The interpretation of a valued cell is given according to different variability patterns and information types defined as sub-concepts of *Constraint* in the metamodel.

- Boolean: states that the feature is either present or absent
- Integer: integer numbers
- Double: real numbers
- VariabilityConceptRef: reference to a product or a feature
- Partial: states that the feature is partially or conditionally present
- Multiple (And, Or, Xor): composition of values constrained by a cardinality
- Unknown: states that the presence or absence of the feature is uncertain
- Empty: the cell is simply empty
- Inconsistent: the cell is inconsistent with the other cells bound to the same feature

The domain of a feature is represented as a set of *Simple* elements (*Boolean, Integer, Double* or *VariabilityConceptRef*) which define the valid values for the cells that are related to this feature. The domain allow us to detect invalid values and reason on discrete values such as features but also use the properties of boolean, integers and real values.

The transformation toolsuite. Having a formalizing canvas with a metamodel is only one means to a larger end. Formalizing PCMs in their whole diversity and heterogeneity requires a set of transformation steps. These steps include:

- parsing: extracting the PCM from its original artefact (*e.g.*, MediaWiki code);
- preprocessing: normalizing the PCM;
- extracting information: interpreting cells and extracting variability concepts and feature' domains.

After the preprocessing phase, it remains to interpret the cells in order to extract the variability information that a PCM contains. In this phase, we progressively enrich the model with new information. This is by far the most difficult part: automated techniques can be inaccurate because of the set of transformation rules implemented. Several iterations are needed.

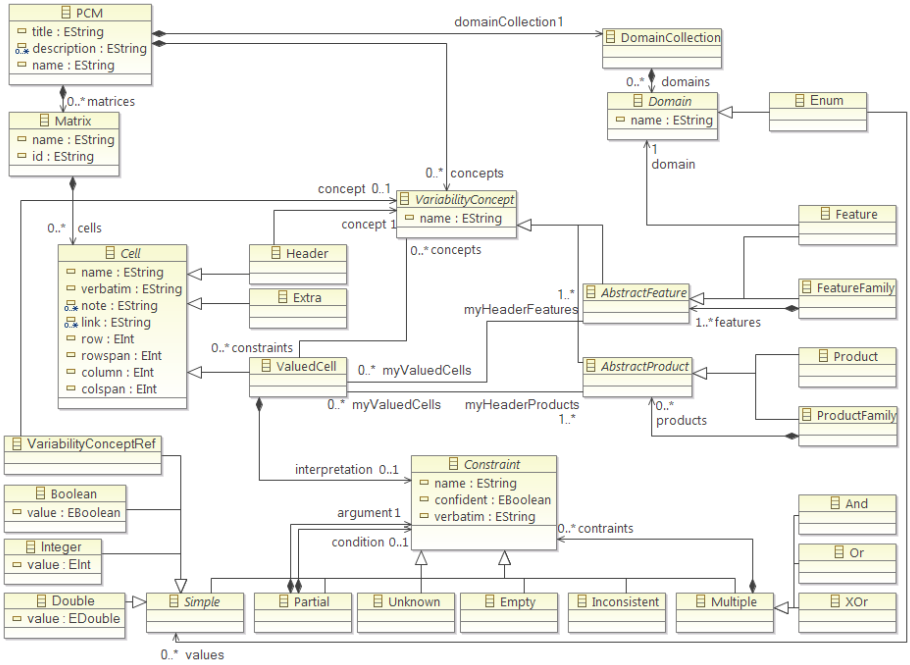


Figure 2.10: The PCM Metamodel

Evaluation

To assess our model-based formalization, we address the following research questions:

- **RQ1: Is our metamodel complete and relevant to the formalization of PCMs?** The metamodel should precisely represent the domain of PCMs and be able to formalize existing examples. The large diversity and number of PCMs make the formalization process error-prone and time consuming.

- **RQ2: What is the precision of our transformation of raw PCMs to PCM models?** The transformation is quite complex and the interpretation of variability information can be inaccurate.

Dataset. We selected 75 PCMs from Wikipedia. Though most of them are related to software (audio player software, browser synchronizers, application launchers, disc images, *etc.*) our sample also covers a very large spectrum of domains. This includes domains such as photography and technology (digital SLRs, embedded computer systems), sports and hobbies (ski resorts, video games), economy (tax preparation, enterprise bookmarking), electronics and hardware (fanless switches, chipsets, radio modules), history (World War II), healthcare (dental practice management, dosimeters, birth control) among others.

Formalization of PCMs. Following our automated process, depicted by Figure 2.9, the 75 selected PCMs were first parsed into models that conform to our PCM metamodel. Then, we manually configured the preprocessing step in order to specify the headers of each matrix. Finally, we executed the extracting information step without configuration and only with default interpretation rules.

These 75 Wikipedia PCMs are made of a total of 211 matrices from various sizes, going from 2 to 241 rows, and 3 to 51 columns for a total of 47267 cells. Among them, 6800 (14.39%) are *Headers*, describing either products or features. Another 992 (2.10%) are *Extra* cells that do not carry any valuable information. Finally 39475 (83.51%) cells are considered as *ValuedCells*.

Participants. To evaluate our research questions, each analyzed PCM was evaluated by at least one person among a panel of 20 persons (mainly researchers and engineers) that were not aware of our work. That is, they have never seen either our elaborated metamodel or our tooling before the experiment.

Evaluation Sessions. We organized two evaluation sessions where the evaluators were explained the goal of the experiment. We provided a tutorial describing the tool they will have to use, as well as the concepts they were about to evaluate and related illustrative examples. The checking process consists of looking at each cell of a PCM to identify cells for which computed formalism does not match the expected one. In such a case, the evaluators have to state whether the expected domain value exists in the metamodel, provide a proposition of a new concept, claim that there is no possible interpretation, or declare that he/she does not know at all how to analyze the cell. In addition to the validation task using the interface, evaluators were allowed to leave comments on an additional menu and to exchange with us.

Evaluation Scenario. The tool proposes a randomly chosen PCM in a way to assure the global coverage of the 75 PCMs. Consequently, during the group session, no two evaluators have the same PCM to evaluate. Right clicking on a cell displays options to validate or not its proposed formalization. To avoid painful individual validation, evaluators are allowed to make multiple selections for collective validation or corrections. Once evaluated, the cells are colored in green, but it is still possible to modify the evaluation. Once the evaluation of one matrix is finished, evaluators can check the interpretation of the other ones in order to complete the PCM evaluation. At the end, they submit their evaluation to a database and possibly start a new evaluation.

Evaluated cells. Among the 39475 cells of the 75 PCMs, 20.83% were ignored (the evaluator declared that he/she does not know how to analyze the cell) or omitted (no results) by our evaluators and 3.02% were subject to conflicts between evaluators (difference in the correction). As a consequence, 30061 cells are evaluated and present analyzable results. In the following, we will answer the research questions according to these evaluated cells.

Answer to RQ1. During the experiment, 95.72% of the evaluated cells were validated or corrected with concepts of the metamodel. The most used concepts are Boolean, VariabilityConceptRef and Unknown. Only 4.28% of cells were not validated and the evaluators proposed a new concept. Our metamodel thus proposes a set of relevant concepts. Three new concepts have emerged (dates, dimensions, and versions). Furthermore, while exchanging with our evaluators, one difficulty that emerged was related to the semantics of the Multiple cell that expresses a composition with an AND, OR or XOR operator. The semantics of Multiple looks simple at first glance, but is not intuitive and hard to determine at the PCM level.

Answer to RQ2. Regarding a fully automated approach (excepting the very first preprocessing steps described in the protocol), our automated approach has been able to qualify our data set with a precision of 93.11%. Overlapping concepts, missing concepts, missing interpretation rules, or implementation errors are the four reasons why our transformation is sometimes inaccurate. The development of specific rules or the manual correction can improve the quality of PCM models.

Large-scale experiments, Multi-Metamodel, and Opencompare

Based on these encouraging results, we have applied the approach on a larger dataset consisting of all tabular data contained in 1+ million Wikipedia pages (English version). Prior to the large-scale experiments, we have modified the metamodel as well as the transformations. The reason is that the original metamodel of Figure 2.10 suffered from drawbacks when developing specific services and tools (RQ3 in Figure 2.9). Specifically, although the metamodel correctly represents the core concepts, the metamodel was not satisfactory for addressing all requirements and tasks related to interoperability, edition, and manipulation. We came to this conclusion after several experiments with engineers and students that did use the metamodel with non-trivial programming tasks.

We addressed this problem by developing a domain metamodel and separating each concern in a task-specific metamodel – an approach called a multi-metamodel in Guillaume Bécan's thesis [45]. We also toke the opportunity to add new concepts as suggested by the participants of our previous experiment (see RQ1 and RQ2). We evaluated our revised solution on more than 1,500,000 PCMs of Wikipedia.

Results showed that the concepts in our domain metamodel are used in the context of modelling PCMs (RQ1). Compared to the original metamodel, our domain metamodel is simpler but still provides the necessary concepts for the domain of PCMs. Related to RQ2, our importing and exporting capabilities are able to handle a large variety of PCMs with a precision of 91%. The formalization errors mainly come from the presence of tables that are not PCMs in our dataset as well as the incorrect detection of the orientation of the PCMs (*i.e.*, when products are perceived as features, and vice-versa). Finally, the domain metamodel was easier to present and the different students and researchers did not have difficulty understanding it. In a few lines of code, they were able to manipulate a PCM and extract relevant information. For more complex development (*e.g.*, product charts), the participants benefited from the task-specific metamodels.

The new approach presented and the related services have been integrated in an initiative called OpenCompare. The objective of OpenCompare was to create a community of PCM contributors and users. For that purpose, OpenCompare provided innovative services for PCMs in the form of a website with an editor, a comparator, a configurator, importing and exporting facilities, and an API.

replication

Data and results of the preliminary Wikipedia analysis are available here: https://github.com/FAMILIAR-project/familiar-documentation/tree/master/manual/Wikipedia. Tools developed as part of the user study are here: https://github.com/gbecan/Tools4PCM. I also strongly recommend the PhD thesis of Guillaume Bécan: G. Bécan, 'Metamodels and feature models : complementary approaches to formalize product comparison matrices', Theses, Université Rennes 1, Sep. 2016. `https://tel.archives-ouvertes.fr/tel-01416129` that contains many details about the experiments.

2.3 Sampling feature models' configurations

The content of this section is adapted from the following publications:
Q. Plazar, M. Acher, G. Perrouin, X. Devroey and M. Cordy, 'Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?', in *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*, Xian, China, Apr. 2019, pp. 1–12. https://hal.inria.fr/hal-01991857
A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin and B. Baudry, 'Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack', *Empirical Software Engineering (ESE)*, vol. 24, no. 2, pp. 674–717, Jul. 2019, Empirical Software Engineering journal. DOI: 10.07980. https://doi.org/10.1007/s10664-018-9635-4

Out of a feature model, developers can sample configurations of interest *e.g.*, for testing their systems. The hope is that the subset of configurations is a good representative of the whole configurations' set. There are many ways of sampling configurations depending on *e.g.*, testing objectives. We have contributed to this rich area along two directions. First, we performed an empirical study to determine which sampling strategies are the most effective to find configurations' failures and bugs (Section 2.3.1). Second, we assessed the scalability and quality of existing sampling techniques *w.r.t.* uniform sampling (Section 2.3.2).

2.3.1 Effectiveness of sampling strategies for testing

A major challenge for developers of configurable systems is to ensure that all combinations of options (configurations) correctly allow the software to compile, build, and run. Configurations that fail can hurt potential users, miss opportunities, and degrade the success or reputation of a project. Though formal methods and program analysis can identify some classes of defects [302] – leading to variability-aware testing approaches (*e.g.*, [231, 181, 180]) – a common practice is still to execute and *test* a sample of (representative) configurations. Indeed, enumerating all configurations is perceived as impossible, impractical or both. Prior empirical investigations (*e.g.*, [211, 270, 269]) suggest that using a sample of configurations is effective to find configuration faults, at low cost. However, evaluations were carried out on a small subset of the total number of configurations or faults, constituting a threat to validity. They are usually based on a corpus of faults extracted from problem tracking systems and are therefore incomplete. Knowing *all* the failures of the whole configurable system provides a unique opportunity to accurately assess the error-detection capabilities of sampling techniques *with a ground truth*.

This contribution aims to grow the body of knowledge (*e.g.*, in the fields of combinatorial testing and software product line engineering [211, 270, 212, 154, 143, 87]) with a new research approach: the exhaustive testing of *all* configurations. While practitioners can hardly enumerate all configurations at each commit or even release, we believe researchers have a lot to learn by rigorously and exhaustively testing a configurable system. Specifically, gathering the ground truth of the whole configuration space is an unique opportunity for assessing the relative effectiveness of sampling strategies. We consider the case of JHipster, a popular code generator for web applications. Our goals are:

- to investigate the engineering effort and the computational resources needed for deriving and testing all configurations (**RQ1**), and
- to discover how many failures and faults can be found using exhaustive testing in order to provide a ground truth for comparison of diverse sampling strategies (**RQ2**).

We describe the effort required to distribute the testing scaffold for the 26,000+ configurations of JHipster, as well as the interaction bugs that we discovered. We cross this analysis with the qualitative assessment of JHipster's lead developers (**RQ3**). Overall, we collect multiple sources that are of interest for

- researchers interested in building evidence-based theories or tools for testing configurable systems;
- practitioners in charge of establishing a suitable strategy for testing their systems at each commit or release.

In short, we report on the first ever endeavour to test *all* possible configurations of an industry-strength open-source configurable software system: JHipster.

Case Study

JHipster is an open-source, industrially used generator for developing Web applications [135]. Started in 2013, the JHipster project has been increasingly popular (18000+ stars on GitHub) with a strong community of users and 600+ contributors in February 2021.

From a user-specified configuration, JHipster generates a complete technological stack constituted of Java and Spring Boot code (on the server side) and Angular and Bootstrap (on the front-end side). The generator supports several technologies ranging from the database used (*e.g.*, *MySQL* or *MongoDB*), the authentication mechanism (*e.g.*, *HTTP Session* or *Oauth2*), the support for social log-in (via existing social networks accounts), to the use of microservices. Technically, JHipster uses *npm* and *Bower* to manage dependencies and Yeoman[4] (aka *yo*) tool to scaffold the application [258]. JHipster relies on conditional compilation with EJS[5] as a variability realisation mechanism. Figure 1.2(a), page 8 presents an excerpt of class *DatabaseConfiguration.java*. The options reactive, elatsicsearch, h2Disk, h2Memory operate over Java annotations, fields, methods, *etc.* The options are also present in the Maven file (pom.xml), see Figure 1.2(b), page 8.

JHipster is a *complex* configurable system with the following characteristics: i) a variety of languages (JavaScript, CSS, SQL, *etc.*) and advanced technologies (Maven, Docker, *etc.*) are combined to generate variants; ii) there are 48 configuration options and a configurator guides the user throughout different questions. Not all combinations of options are possible and there are 15 constraints between options; iii) variability is scattered among numerous kinds of artefacts (*pom.xml*, Java classes, Docker files, *etc.*) and several options typically contribute to the activation or deactivation of portions of code, which is commonly observed in configurable software [171]. This complexity challenges core developers and contributors of JHipster. Unsurprisingly, numerous configuration faults have been reported on mailing lists and eventually fixed with commits.[6] Though formal methods and variability-aware program analysis can identify some defects [302, 231, 86], a significant effort would be needed to handle them in this technologically diverse stack. Thus, the current practice is rather to

[4] http://yeoman.io/
[5] http://www.embeddedjs.com/
[6] *e.g.*, https://tinyurl.com/bugjhipster15

execute and test some configurations and JHipster offers opportunities to assess the cost and effectiveness of sampling strategies [211, 270, 212, 154, 143, 87]. Due to the reasonable number of options and the presence of 15 constraints, we (as researchers) also have a unique opportunity to gather a ground truth through the testing of *all* configurations.

Methodology. We address these questions through quantitative and qualitative research. We initiated the work in September 2016 and selected JHipster 3.6.1[7] (release date: mid-August 2016). The 3.6.1 corrects a few bugs from 3.6.0; the choice of a "minor" release avoids finding bugs caused by an early and unstable release.

The two first authors of [135] worked full-time for four months to develop the infrastructure capable of testing all configurations of JHipster. They were graduate students, with strong skills in programming and computer science. They came to Rennes as part of their Master's thesis and I collaborated with them in Brittany. Through physical and virtual meetings, we gathered several qualitative insights throughout the development. Besides, we decided not to report faults whenever we found them. Indeed, we wanted to observe whether and how fast the JHipster community would discover and correct these faults. We monitored JHipster mailing lists to validate our testing infrastructure and characterize the configuration failures in a qualitative way.

RQ1: What is the feasibility of testing all JHipster configurations?

Insights about modelling JHipster variability. The first step towards a complete and thorough testing of JHipster variants is the modelling of its configuration space. JHipster comes with a command-line configurator. However, we quickly noticed that a brute force tries of every possible combinations has scalability issues. Some answers activate or deactivate some questions and options. As a result, we rather considered the source code from GitHub for identifying options and constraints. Though options are scattered amongst artefacts, there is a central place that manages the configurator and then calls different sub-generators to derive a variant.

We essentially consider *prompts.js*, which specifies questions prompted to the user during the configuration phase, possible answers (a.k.a. options), as well as constraints between the different options. Listing 2.1 gives an excerpt for the choice of a databaseType. Users can select no database, sql, mongodb, or cassandra options. There is a pre-condition stating that the prompt is presented only if the microservice option has been previously selected (in a previous question related to applicationType). In general, there are several conditions used for basically encoding constraints between options.

We specified JHispter's variability using a feature model (see Section 2.1). Though there is a gap with the configurator specification (see Listing 2.1), we can encode its *configuration semantics* and hierarchically organize options with a feature model. We decided to interpret the meaning of the configurator as follows:

1. each multiple-choice question is an (abstract) feature. In case of "yes" or "no" answer, questions are encoded as optional features (*e.g.*, databaseType is optional in Listing 2.1);
2. each answer is a concrete feature (*e.g.*, sql, mongodb, or cassandra in Listing 2.1). All answers to questions are exclusive and translated as alternative groups in the feature modelling jargon. A notable exception is the selection of testing frameworks in which several answers can be both selected; we translated them as an Or-group;

[7] https://github.com/jhipster/generator-jhipster/releases/tag/v3.6.1

```
(...)
when: function (response) {
    return applicationType === 'microservice';
  },
  type: 'list',
  name: 'databaseType',
  message: function (response) {
      return getNumberedQuestion('Which *type* of database would you like to use?',
    applicationType === 'microservice');},
  choices: [
      {value: 'no', name: 'No database'},
      {value: 'sql', name: 'SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle)'},
      {value: 'mongodb', name: 'MongoDB'},
      {value: 'cassandra',name: 'Cassandra'}
  ],
  default: 1
(...)
```

Listing 2.1: Configurator: server/prompt.js (excerpt)

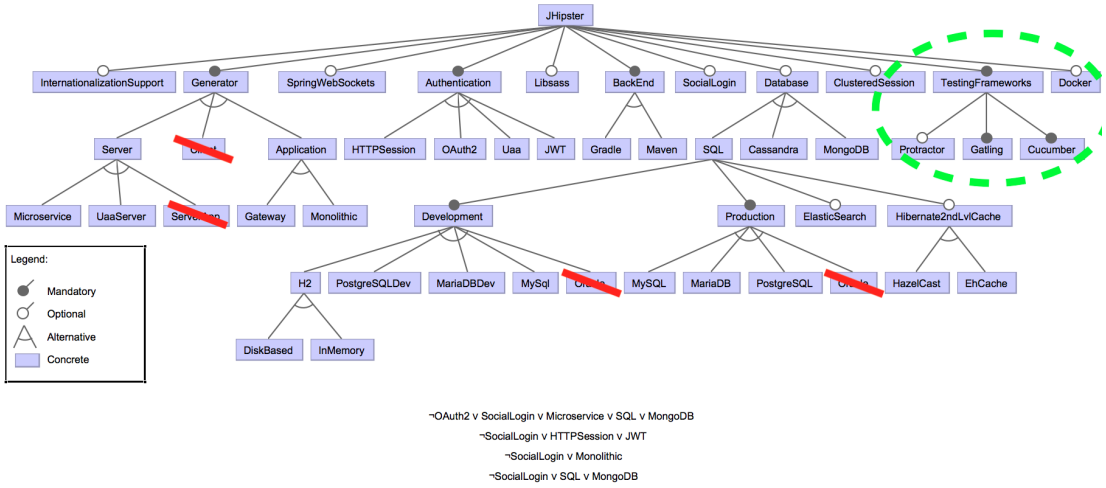


Figure 2.11: JHipster specialised feature model used to generate JHipster variants (only an excerpt of cross-tree constraints is given).

3. pre-conditions of questions are translated as constraints between features.

Based on an in-depth analysis of the source code and attempts with the configurator, we have manually elaborated[8] an initial feature model presented in Figure 2.11: 48 identified features and 15 constraints (we only present four of them in Figure 2.11 for the sake of clarity). The total number of valid configurations is 162,508.

Our goal was to derive and generate all JHipster variants corresponding to feature model configurations. However, we decided to adapt the initial model as follows:

1. we added Docker as a new optional feature (Docker) to denote the fact that the deployment may be performed using Docker or using Maven or Gradle.

[8] In the original publication [135], we use the terminology "reverse engineering" to refer to the fact we have made explicit the variability and semantics of the JHipster configurator. However, as the process was manual (*i.e.*, there was no automation), I consider in this manuscript that *modelling* is a better terminology.

2. we excluded client/server standalones since there is a limited interest for users to consider the server (respectively client) without a client (respectively server): stack and failures most likely occur when both sides are inter-related;

3. we included the three testing frameworks in all variants. The three frameworks do not augment the functionality of JHipster and are typically here to improve the testing process, allowing us to gather as much information as possible about the variants;

4. we excluded Oracle-based variants. Oracle is a proprietary technology with technical specificities that are quite hard to fully automate.

Strictly speaking, we test *all* configurations of a *specialized* JHipster, presented in Figure 2.11. This specialization can be thought of a test model, which focusses on the most relevant open source variants. Overall, we consider that our specialization of the feature model is conservative and still substantial with a total of possible **26,256** variants.

Human Cost (RQ 1.1)

The development of the complete derivation and testing infrastructure was achieved in about 4 months by 2 people (*i.e.*, **8 person * month** in total). For each activity, we report the duration of the effort realized in the first place. Some modifications were also made in parallel to improve different parts of the solution – we count this duration in subsequent activities.

Modelling configurations. The elaboration of the first major version of the feature model took us about **2 weeks** based on the analysis of the JHipster code and configurator.

Configuration-aware testing workflow. Based on the feature model, we initiated the development of the testing workflow. We added features and testing procedures in an incremental way. The effort spanned on a period of **8 weeks**.

All-inclusive environment. The building of the Debian image was done in parallel to the testing workflow. It also lasted a period of **8 weeks** for identifying all possible tools and settings needed.

Distributing the computation. We decided to deploy on Grid'5000 at the end of November and the implementation has lasted **6 weeks**. It includes a learning phase (1 week), the optimization for caching dependencies, and the gathering of results in a central place (a CSV-like table with logs).

Answering RQ1.1: *What is the cost of engineering an infrastructure capable of automatically deriving and testing all configurations?* The testing infrastructure is itself a configurable system and requires a substantial engineering effort (8 man-months) to cover all design, implementation and validation activities, the latter being the most difficult.

Computational Cost (RQ 1.2)

We used a network of machines that allowed us to test all 26,256 configurations in *less than a week*. Specifically, we performed a reservation of **80** machines for **4 periods** (4 nights) of **13 hours**. The analysis of 6 configurations took on average about 60 minutes. The total CPU time of the workflow on all the configurations is **4,376 hours**. Besides CPU time, the processing of all variants also required enough free disk space. Each scaffolded Web application occupies between 400MB and 450MB, thus forming a total of **5.2 terabytes**.

We replicated three times our exhaustive analysis (with minor modifications of our testing procedure each time); we found similar numbers for assessing the computational cost on Grid'5000. As part of our last experiment, we observed suspicious failures for 2,325 configurations with the same error message: "Communications link failure", denoting network communication error (between a node and the controller for instance) on the grid. Those failures have been ignored and configurations have been re-run again afterwards to have consistent results.

Answering RQ1.2: *What are the computational resources needed to test all configurations?* Testing all configurations requires a significant amount of computational resources (4,376 hours CPU time and 5.2 terabytes of disk space).

RQ2: To what extent can sampling help to discover defects in JHipster?

The execution of the testing workflow yielded a large file comprising numerous results for each configuration. This file allows one to identify failing configurations, *i.e.*, configurations that do not compile or build. In addition, we also exploited stack traces for grouping together some failures. We present here the ratios of failures and associated faults.

Bugs: A Quick Inventory. Out of the **26,256** configurations we tested, we found that **9,376** (**35.70**%) failed. This failure occurred either during the compilation of the variant or during its packaging as an executable Jar file (that includes execution of the different Java and JavaScript tests generated by JHipster). We also found that some features were more concerned by failures. Regarding the application type, for instance, *microservice gateways* and *microservice applications* are proportionally more impacted than *monolithic applications* or *UAA server* with, respectively, **58.37**% of failures (4,184 failing microservice gateways configurations) and **58.3**% of failures (532 failing microservice applications configurations). *UAA authentication* is involved in most of the failures: **91.66**% of *UAA-based microservices* applications (4,114 configurations) fail to deploy.

Statistical Analysis. Previous results do not show the root causes of the configuration failures – what features or interactions between features are involved in the failures? To investigate correlations between features and failures' results, we decided to use the Association Rule learning method [132]. It aims at extracting relations (called *rules*) between variables of large data-sets. The Association Rule method is well suited to find the (combinations of) features leading to a failure, out of tested configurations. Formally and adapting the terminology of association rules, the problem can be defined as follows.

- let $F = \{ft_1, ft_2, \ldots, ft_n, bs\}$ be a set of n features (ft_i) plus the status of the build (bs), *i.e.*, build failed or not;
- let $C = \{c_1, c_2, \ldots, c_m\}$ be a set of m configurations.

Each configuration in C has a unique identifier and contains a subset of the features in F and the status of its build. A rule is defined as an implication of the form: $X \Rightarrow Y$, where $X, Y \subseteq F$. The outputs of the method are a set of rules, each constituted by:

- X the *left-hand side (LHS)* or antecedent of the rule;
- Y the *right-hand side (RHS)* or consequent of the rule.

For our problem, we consider that Y is a single target: the status of the build. For example, we want to understand what combinations of features lead to a failure, either during the compilation or the build process. An example rule could be:

$$\{\text{mariadb}, \text{graddle}\} \Rightarrow \{\text{build failure}\}$$

Table 2.3: Association rules involving compilation and build failures

Id	Left-hand side	Right-hand side	Support	Conf.	GitHub Issue	Report/Correction date
MoSo	DatabaseType="mongodb", EnableSocialSignIn=true	Compile=KO	0.488 %	1	4037	27 Aug 2016 (report and fix for milestone 3.7.0)
MaGr	prodDatabaseType="mariadb", buildTool="gradle"	Build=KO	16.179 %	1	4222	27 Sep 2016 (report and fix for milestone 3.9.0)
UaDo	Docker=true, authenticationType="uaa"	Build=KO	6.825 %	1	UAA is in Beta	Not corrected
OASQL	authenticationType="uaa", hibernateCache="no"	Build=KO	2.438 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
UaEh	authenticationType="uaa", hibernateCache="ehcache"	Build=KO	2.194 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
MaDo	prodDatabaseType="mariadb", applicationType="monolith", searchEngine="false", Docker="true"	Build=KO	5.590%	1	4543	24 Nov 2016 (report and fix for milestone 3.12.0)

Meaning that if *mariadb* and *gradle* are activated, configurations will not build.

As there are many possible rules, some well-known measures are typically used to select the most interesting ones. In particular, we are interested in the *support*, the proportion of configurations where LHS holds and the *confidence*, the proportion of configurations where both LHS and RHS hold.

Table 2.3 gives some examples of the rules we have been able to extract. We parametrized the method as follows. First, we restrained ourselves to rules where the RHS was a failure: either *Build=KO* (build failed) or *Compile=KO* (compilation failed). Second, we fixed the confidence to 1: if a rule has a confidence below 1 then it is not asserted in all configurations where the LHS expression holds – the failure does not occur in all cases. Third, we lowered the support in order to catch all failures, even those afflicting smaller proportion of the configurations. For instance, only 224 configurations fail due to a compilation error; in spite of a low support, we can still extract rules for which the RHS is *Compile=KO*. We computed redundant rules using facilities of the R package arules.[9] As some association rules can contain already known constraints of the feature model, we ignored some of them.

We first considered association rules for which the size of the LHS is either 1, 2 or 3. We extracted 5 different rules involving two features (see Table 2.3). We found no rule involving 1 or 3 features. We decided to examine the 200 association rules for which the LHS is of size 4. We found out a sixth association rule that incidentally corresponds to one of the first failures we encountered in the early stages of this study. We conclude that six feature interaction faults explain 99.1% of the failures.

Qualitative Analysis. Table 2.3 gives the support, confidence for each association rule. We also confirm each fault by giving the GitHub issue and date of fix. There are 6 important faults, caused by the interactions of several features. Specifically, only two features are involved in five (out of six) faults, and four features are involved in the last fault.

[9] https://cran.r-project.org/web/packages/arules/

Testing infrastructure. We have not found a common fault for the remaining **242** configurations that fail. We came to this conclusion after a thorough and manual investigation of all logs.[10] We noticed that, despite our validation effort with the infrastructure (see **RQ1**), the observed failures are caused by the testing tools and environment. Specifically, the causes of the failures can be categorized in two groups: *(i)* several network access issues in the grid that can affect the testing workflow at any stage and *(ii)* several unidentified errors in the configuration of building tools (`gulp` in our case).

Answering RQ2.1: *How many and what kinds of failures/faults can be found in all configurations?* Exhaustive testing shows that almost 36% of the configurations fail. Our analysis identifies 6 interaction faults as the root cause for this high percentage.

Sampling Techniques Comparison (RQ2.2)

We first discuss the sampling strategy used by the JHipster team. We then use our dataset to make a ground truth comparison of six state-of-the-art sampling techniques.

JHipster Team Sampling Strategy. The JHipster team uses a sample of 12 representative configurations for the version 3.6.1, to test their generator (see [136] for further explanations on how these were sampled). During a period of several weeks, the testing configurations have been used at *each commit*. These configurations fail to reveal any problem, *i.e.*, the Web-applications corresponding to the configurations successfully compiled, build and run. We assessed these configurations with our own testing infrastructure and came to the same observation. We thus conclude that this sample was not effective to reveal any defect.

Comparison of Sampling Techniques. As testing all configurations is very costly (see **RQ1**), sampling techniques remain of interest. We would like to find as many failures and faults as possible with a minimum of configurations in the sampling. For each failure, we associate a fault through the automatic analysis of features involved in the failed configuration (see previous subsections).

Sampling techniques. We address **RQ2.2** with numerous sampling techniques considered in the literature [211, 246, 172, 1]. For each technique, we report on the number of failures and faults.

***t*-wise sampling.** We selected 4 variations of the *t*-wise criteria: **1-wise**, **2-wise**, **3-wise** and **4-wise**. We generate the samples with *SPLCAT* [172], which has the advantage of being deterministic: for one given feature model, it will always provide the same sample. The 4 variations yield samples of respectively **8**, **41**, **126** and **374** configurations. **1-wise** only finds **2** faults; **2-wise** discovers **5 out of 6 faults**; **3-wise** and **4-wise** find **all** of them. It has to be noted that the discovery of a 4-wise interaction fault with a 3-wise setting is a 'collateral' effect [247], since any sample covering completely t-way interactions also yields an incomplete coverage of higher-order interactions.

One-disabled sampling. Using **one-disabled** sampling algorithm, we extract configurations in which one feature is disabled at a time. To overcome any bias in selecting the first valid configuration, as suggested by Medeiros *et al.* [211], we applied a random selection instead. We therefore select a valid random configuration for each disabled feature (called

[10] Such configurations are tagged by "ISSUE:env" in the column "bug" of the JHipster results CSV file available online https://github.com/xdevroey/jhipster-dataset.

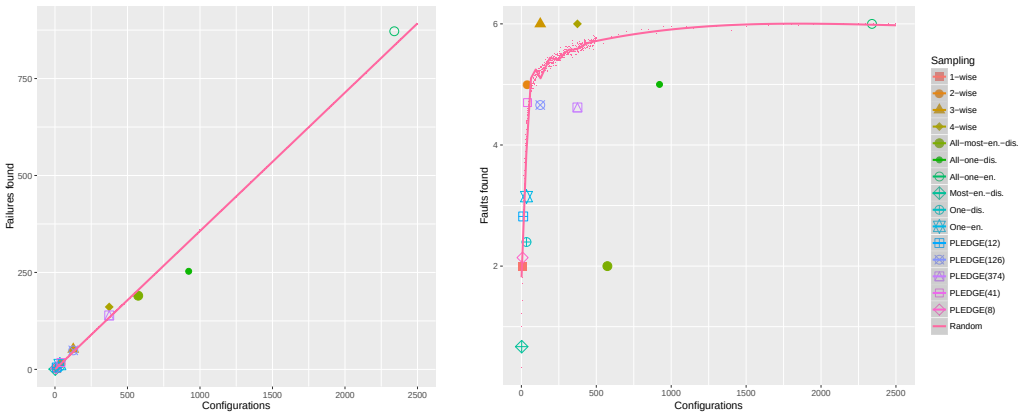
Table 2.4: Efficiency of different sampling techniques (bold values denote the highest efficiencies)

Sampling technique	Sample size	Failures (σ)	Failures eff.	Faults (σ)	Fault eff.
1-wise	8	2.000 (N.A.)	25.00%	2.000 (N.A.)	25.00%
Random(8)	8	2.857 (1.313)	35.71%	2.180 (0.978)	27.25%
PLEDGE(8)	8	3.160 (1.230)	39.50%	2.140 (0.825)	26.75%
Random(12)	12	4.285 (1.790)	35.71%	2.700 (1.040)	22.5%
PLEDGE(12)	12	4.920 (1.230)	41.00%	2.820 (0.909)	23.50%
2-wise	41	14.000 (N.A.)	34.15%	5.000 (N.A.)	12.20%
Random(41)	41	14.641 (3.182)	35.71%	4.490 (0.718)	10.95%
PLEDGE(41)	41	17.640 (2.500)	43.02%	4.700 (0.831)	11.46%
3-wise	126	52.000 (N.A.)	41.27%	6.000 (N.A.)	4.76%
Random(126)	126	44.995 (4.911)	35.71%	5.280 (0.533)	4.19%
PLEDGE(126)	126	49.080 (11.581)	38.95%	4.660 (0.698)	3.70%
4-wise	374	161.000 (N.A.)	**43.05%**	6.000 (N.A.)	1.60%
Random(374)	374	133.555 (8.406)	35.71%	5.580 (0.496)	1.49%
PLEDGE(374)	374	139.200 (31.797)	37.17%	4.620 (1.181)	1.24%
Most-enabled-disabled	2	0.683 (0.622)	34.15%	0.670 (0.614)	**33.50%**
All-most-enabled-disabled	574	190.000 (N.A.)	33.10%	2.000 (N.A.)	0.35%
One-disabled	34	7.699 (2.204)	0.23%	2.398 (0.878)	0.07%
All-one-disabled	922	253.000 (N.A.)	27.44%	5.000 (N.A.)	0.54%
One-enabled	34	12.508 (2.660)	0.37%	3.147 (0.698)	0.09%
All-one-enabled	2,340	872.000 (N.A.)	37.26%	6.000 (N.A.)	0.26%
ALL	26,256	9,376.000 (N.A.)	35.71%	6.000 (N.A.)	0.02%

one-disabled in our results) and repeat experiments 1,000 times to get significant results. This gives us a sample of **34 configurations** which detects on average **2.4 faults** out of 6. Additionally, we also retain **all-one-disabled** configurations (*i.e.*, all valid configurations where one feature is disabled and the other are enabled). The all-one-disabled sampling yields a total sample of **922 configurations** that identifies **all faults but one**.

One-enabled and most-enabled-disabled sampling. In the same way, we implemented sampling algorithms covering the **one-enabled** and **most-enabled-disabled** criteria [211, 1]. As for one-disabled, we choose to randomly select valid configurations instead of taking the first one returned by the solver. Repeating the experiment 1,000 times: one-enabled extracts a sample of **34 configurations** which detects **3.15 faults** on average; and most-enabled-disabled gives a sample of **2 configurations** that detects **0.67 faults** on average. Considering all valid configurations, **all-one-enabled** extracts a sample of **2,340 configurations** and identifies all the **6 faults**. **All-most-enabled-disabled** gives a sample of **574 configurations** that identifies **2 faults** out of 6.

Dissimilarity sampling. We also considered *dissimilarity* testing for software product lines [143, 133] using **PLEDGE** [146]. We retained this technique because it can afford any testing budget (sample size and generation time). For each sample size, we report the average failures and faults for 100 PLEDGE executions with the greedy method in 60 secs [146]. We selected (respectively) **8**, **12**, **41**, **126** and **374** configurations, finding (respectively) **2.14**, **2.82**, **4.70**, **4.66** and **4.60** faults out of 6.



(a) Failures found by sampling techniques

(b) Faults found by sampling techniques

Figure 2.12: Defects found by sampling techniques

Random sampling. Finally, we considered **random** samples from size 1 to 2,500. The random samples exhibit, by construction, 35.71% of failures on average (the same percentage that is in the whole dataset). To compute the number of unique faults, we simulated 100 random selections. We find, on average, respectively **2.18**, **2.7**, **4.49**, **5.28** and **5.58** faults for respectively **8**, **12**, **41**, **126** and **374** configurations.

Fault and failure efficiency. We consider two main metrics to compare the efficiency of sampling techniques to find faults and failures w.r.t the sample size. *Failure efficiency* is the ratio of *failures to sample size*. *Fault efficiency* is the ratio of *faults to sample size*. For both metrics, a high efficiency is desirable since it denotes a small sample with either a high failure or fault detection capability.

The results are summarized in Table 2.4. We present in Figure 2.12(a) (respectively, Figure 2.12(b)) the evolution of *failures* (respectively, *faults*) *w.r.t.* the size of random samples. To ease comparison, we place reference points corresponding to results of other sampling techniques. A first observation is that random is a strong baseline for both failures and faults. 2-wise or 3-wise sampling techniques are slightly more efficient to identify faults than random. On the contrary, all-one-enabled, one-enabled, all-one-disabled, one-disabled and all-most-enabled-disabled identify less faults than random samples of the same size. Most-enabled-disabled is efficient on average to detect faults (33.5% on average) but requires to be "lucky". In particular, the first configurations returned by the solver (as done in [211]) discovered 0 fault. This shows the sensitivity of the selection strategy amongst valid configurations matching the most-enabled-disabled criterion. Based on our experience, we recommend researchers a random strategy instead of picking the first configurations when assessing one-disabled, one-enabled, and most-enabled-disabled.

PLEDGE is superior to random for small sample sizes. The significant difference between 2-wise and 3-wise is explained by the sample size: although the latter finds all the bugs (one more than 2-wise) its sample size is triple (126 configurations against 41 for 2-wise). In general, a relatively small sample is sufficient to quickly identify the 5 or 6 most important faults – there is no need to cover the whole configuration space.

A second observation is that there is no correlation between failure efficiency and fault efficiency. For example, all-one-enabled has a failure efficiency of 37.26% (better than random and many techniques) but is one of the worst techniques in terms of fault rate due to its high sample size. In addition, some techniques, like all-most-enable-disabled, can find numerous failures that in fact correspond to the same fault.

Answering RQ2.2 *How effective are sampling techniques comparatively?* To summarise:

- random is a strong baseline for failures and faults;
- 2-wise and 3-wise sampling are slightly more efficient to find faults than random;
- most-enabled-disabled is efficient on average to detect faults but requires to be lucky;
- dissimilarity is superior to random for small sample sizes;
- a small sample is sufficient to identify most important faults, there is no need to cover the whole configuration space; and
- there is no correlation between failure and fault efficiencies.

(RQ2.3) *How do our findings compare to other studies* w.r.t. *sampling effectiveness?*

We aim to compare our findings with state-of-the-art results: Are sampling techniques as effective in other case studies? Do our results confirm or contradict findings in other settings? This question is important for i) practitioners in charge of establishing a suitable strategy for testing their systems; ii) researchers interested in building evidence-based theories or tools for testing configurable systems. We perform a literature review of case studies of configuration sampling approaches to test variability intensive systems. We select existing studies (specifically, see references [211, 269, 241, 294, 36]) based on different criteria *e.g.*, we discard evaluations that are solely based on feature models.

Our main findings are as follows:

- **From a practical point of view**: We concur with previous findings that show that most-enabled-disabled is an interesting candidate to initiate the testing of configurations. For identifying further faults (and possibly all), we confirm that 2-wise or 3-wise provides a good balance between sampling size and fault-detection capability.
- **From a researcher point of view**: Our results show that the assessment of sampling techniques may highly vary depending on the metrics used (failure or fault efficiency). Besides, a corpus of faults coming from an issue tracking system (GitHub) is a good approximation of the real, exhaustive corpus of faults. It is reassuring for research works based on a manually collected corpus.

Q3: Practitioners Viewpoint

We interviewed the JHipster lead developer, Julien Dubois, for one hour and a half, at the end of January 2017. We prepared a set of questions and performed a semi-structured interview on Skype for allowing new ideas during the meeting. We then exchanged emails with two core developers of JHipster, Deepu K Sasidharan and Pascal Grimaud. Based on an early draft of our article, they clarified some points and freely reacted to some of our recommendations. We wanted to get insights on how JHipster was developed, used, and tested. We also aimed to confront our empirical results with their current practice.

(RQ3.1) *What is the most cost-effective sampling strategy for JHipster?* Exhaustive testing sheds a new light on sampling techniques:

- the 12 configurations used by the JHipster team do not find any defect;
- yet, 41 configurations are sufficient to cover the 5 most important faults;

(RQ3.2) *What are the recommendations for the JHipster project?* Recommendations (and challenges) are:

- for a budget of 19 configurations, dissimilarity is the most appropriate sampling strategy;
- the trade-off between cost, popularity, and effectiveness suggests to further experiment with multi-objective techniques;
- crowdsourcing the testing effort would help to face the computational cost of testing JHipster;
- the development and maintenance of a configuration-aware testing infrastructure with a feature model to drive the sampling strategy is mandatory to automate JHipster testing.

replication

Data, scripts, and results are available at https://github.com/xdevroey/jhipster-dataset/

2.3.2 Scalability and quality of uniform samplers

As explained in previous study, there are many ways of sampling configurations (*e.g.*, t-wise, one-enabled, random). In the absence of constraints between the options, Arcuri *et al.* theoretically demonstrate that a uniform random sampling strategy may outperform t-wise sampling [37]. Random sampling thus typically serves as a baseline to evaluate and compare sampling strategies. Sampling is crucial for testing configurable systems, but also for learning functional or non-functional properties of configurations (see Chapter 4). In the context of configurable system, random sampling is widely used to either compare sampling strategies (*e.g.*, see Section Learning variability performance) or simply as a pragmatic strategy to effectively predict. In general, uniform or near-uniform generation of solutions for large satisfiability (SAT) formulas is a problem of theoretical and practical interest *e.g.*, for the testing community.

To sum up, the current body of knowledge emphasizes the importance of random uniform sampling and its specific potential for configurable systems. To assess the applicability of random sampling in practice, we aim to assess actual state-of-the-art implementations on feature models. To this end, we selected two approaches from the literature, UniGen [79, 78] and QuickSampler [116] because they exhibit interesting trade-offs with respect to uniformity and scalability (*e.g.*, QuickSampler sacrifices some uniformity for a substantial increase in performance). In the specific context of highly-configurable software systems and feature models, it is unclear whether UniGen and QuickSampler can scale and sample uniform software configurations. While there exist benchmarks [116] that evaluate and compare these tools, those do not consider feature models and their peculiarities. For example, the `uClinux-config` model, representing the configuration options of an embedded Linux for micro-controllers, has $7.7 * 10^{417}$ possible solutions. In contrast, the largest formula used by UniGen and QuickSampler has $\approx 10^{48}$ solutions [116]. Considering these differences, our

objective is to investigate: *(i)* whether UniGen and QuickSampler are efficient enough (in terms of computation time) to be ***applied on such feature models*** and *(ii)* whether they do so while ***guaranteeing to generate a reasonably-uniform sample*** of configurations. In this contribution, we perform a thorough experiment on 128 real-world feature models.

Study Design

Research Questions. Uniform sampling is an interesting approach to testing configurable systems. However, practitioners and researchers ignore whether state-of-the-art algorithms are applicable over feature models. Specifically, we aim to address three research questions:

- **RQ1** (scalability and execution time): *Are UniGen and QuickSampler able to generate samples out of feature models?* We aim to study the execution time needed for sampling over real-world feature models. It might be the case that samplers are unable to sample and do not scale.
- **RQ2** (uniformity): *Do UniGen and QuickSampler generate uniform configurations out of feature models?* We aim to assess the quality of the sample with respect to uniformity (prior work [116] suggests that QuickSampler is close to uniformity for some SAT instances (non-feature models).
- **RQ3** (relevance for testing): *How do QuickSampler's sacrifices on uniformity impact its bug-finding ability in JHipster?* By relating sampled frequencies of features with their associated bugs on the JHipster case, we perform an early exploration of how these techniques behave with respect to bug distribution [37].

UniGen and QuickSampler. Several SAT samplers exist in the literature [79, 78, 116, 318, 183, 120, 121] and achieve varying compromises between performance and theoretical properties of the sampling process (*e.g.*, uniformity, near-uniformity). We will focus our study on two samplers that achieve state-of-the-art results, UniGen [79, 78], and QuickSampler [116]. A recent ICSE'18 paper [116] compares the two algorithms on large real-world benchmarks (SAT instances), showing that QuickSampler is faster than UniGen, with a distribution reasonably closed to uniform.

On the one hand, UniGen uses a hashing-based algorithm to generate samples in a nearly uniform manner with strong theoretical guarantees: it either produces samples satisfying a nearly uniform distribution or it produces no sample at all. These strong theoretical properties come at a cost: the hashing based approach requires adding large clauses to formulas so they can be sampled. These clauses grow quadratically in size with the number of variables in the formula, which can raise scalability issues.

On the other hand, QuickSampler's algorithm is based on a strong set of heuristics, which are shown to produce samples quickly in practice on a large set of industrial benchmarks [116]. However, the tool offers no guarantee on the distribution of generated samples, or even on the termination of the sampling process and the validity of generated samples (they have to be checked with a SAT solver after the generation phase).

Input Feature Models. We use a large number of well-known and publicly available feature models in our study, which are of various difficulty. Specifically, we rely on the benchmarks used in [202, 184, 189]. Specifically, feature models were used to assess the SAT-hardness of feature models, to investigate the properties of real-world feature models [184] and to evaluate a configuration algorithm for propagating decisions [189].

Feature model benchmark properties In total, we use the feature models of 128 real-world configurable systems (Linux, eCos, toybox, JHipster, *etc.*) with varying sizes and complexity. We first rely on 117 feature models used in [184, 189]. The majority of feature models contain between 1,221 and 1,266 features. Of these 117 models, 107 comprise between 2,968 and 4,138 cross-tree constraints, while one has 14,295 and the other nine have between 49,770 and 50,606 cross-tree constraints [184, 189]. Second, we include 10 additional feature models used in [202] and not in [184, 189]; they also contain a large number of features (*e.g.*, more than 6,000). Third, we also add the JHipster feature model to the study (see Section 2.3.1), a realistic but relatively smaller feature model (45 variables, 26,000+ configurations). We later refer to these benchmarks as the feature model benchmarks.

Once put in conjunctive normal form, these instances typically contain between 1 and 15 thousand variables and up to 340 thousand clauses. The hardest of them, modelling the Linux kernel configuration, contains more than 6 thousand variables, 340 thousand clauses, and is generally seen as a milestone in configurable system analysis.

Replication of [116] In addition to these feature models, we have replicated the initial experiments on industrial SAT formulas as conducted in [116]. We use these results as a sanity check, to ensure that we are using the tools with the same configurations that were previously compared. Moreover, since these original formulae are much smaller than the feature models we use (typically a few thousand clauses), they will provide a basis of results for statistical analysis, in case a solver cannot produce enough samples on the harder formulas. We later refer to these benchmarks as the *non-feature model benchmarks*.

Experimental Setup. We are interested in several characteristics of the samplers under study, which include scalability (execution time) as well as the quality of generated samples (regarding their statistical distribution). For scalability, we evaluate execution time by running each sampler on every benchmark, until it generates 1 million samples or execution times out after 2 hours. For QuickSampler, the timeout duration is split equally between sample generation and validity check (one hour each). Experiments were run on an Intel(R) Core(TM) i7-5600U (2,6 GHz, 2 cores), 16GB RAM, running Linux Fedora 22.

Frequency of features The quality of a sampler's distribution is very hard to evaluate on large benchmarks, since the huge size of the solution space makes standard statistical testing inapplicable. For example, the `uClinux-config` feature model has $7.7 * 10^{417}$ possible solutions. Instead, we rely on an approximate measure of statistical indicators, namely the frequency of observation of certain features in the samples generated. These indicators are not sufficient to show if the distribution of samples is uniform. However, our indicators can pinpoint flaws in the sampling process that are critical for testing purposes, such as a feature never being selected in the produced samples (despite the ground truth states this feature should be selected 80% of the time). More details about the computation of this indicator are available in [249].

Independent support As a final note, both QuickSampler and UniGen can benefit from the knowledge of an *independent support* for the formula they sample. An independent support is a subset of the formula's variables which, when assigned a truth value, leaves at most one possible satisfying valuation for the remaining variables. We chose not to use independent support for feature models in our experiments (details and reasons why can be found in [249]).

Results

RQ1 (scalability and execution time). UniGen is not able to produce samples for any of the feature model benchmarks (except the smallest one, JHipster) and thus cannot be used in the context of large configurable systems. QuickSampler does scale and is able to produce one valid sample per millisecond on most feature models. In general, the heuristic of QuickSampler is effective to select valid configurations, but can also exhibit low valid ratios for some feature models [249].

RQ2 (uniformity). QuickSampler does not generate uniform samples out of real-world feature models; the difference with a uniform distribution is much more severe than with non-feature models (as in [116]). The majority of features exhibit frequencies that deviate above 50%. Some features have up to 800% frequencies differences or are never part of the sample (despite their theoretical presence). The JHipster feature model provides another compelling argument about the non-uniformity of QuickSampler (see below).

RQ3 (case study: JHipster). To derive further insights on the relevance of QuickSampler samples for testing, we consider the feature model of JHipster used in Section Effectiveness of sampling strategies for testing (45 features and 26,256 configurations). Importantly, features and feature interactions causing the bugs have been identified. With the JHipster case, we have an interesting opportunity to investigate the quality of the sampling *w.r.t.* bug-finding ability. This section thus addresses RQ3.

As the JHipster feature model is manageable (only 26,256 configurations), both UniGen and QuickSampler can sample a statistically significant number of samples. Therefore we can plot and exploit the histogram that counts how many times each configuration (SAT solution) has been sampled. Figure 2.13 shows that UniGen is indistinguishable from uniform, but QuickSampler is not close to uniform behavior. In the following, we consider UniGen as uniform (thus having same bug-finding ability as random uniform sampling study of JHipster) and therefore focus only on QuickSampler's ability to find bugs.

To better understand the difference with a uniform distribution, we apply our feature frequency methods (see Figure 2.14). Beyond the clear difference with UniGen, for QuickSampler, we can notice that 18 features have above 10% frequencies deviations and 5 features deviate above 50%:

- $dev(\texttt{MongoDB}) = 116\%$
- $dev(\texttt{Cassandra}) = 107\%$
- $dev(\texttt{UaaServer}) = 94\%$
- $dev(\texttt{Server}) = 87\%$
- $dev(\texttt{MicroserviceApplication}) = 84\%$

Table 2.5 lists all features involved in the 6 interaction faults that cause 99% of failures. We also report *rdev* for showing the positive or negative frequencies deviations for QuickSampler. For instance, the frequency of `Uaa` in QuickSampler samples is greater than the ground truth (+45%) while `MariaDB` is less frequent as it should be (-7%).

Specifically, for the different configuration bug reported by Halin *et al.* [135], we have: **MOSO**, the 2-interaction of `MongoDB` and `SocialLogin` (0.49% out of 35.70% of failures): It is the less important source of bugs. `MongoDB` is much more present than it should be (+116%), which has a positive incidence of the finding of this bug. It should be noted that the theoretical frequency of `MongoDB` is very low, since this feature appears in a very few configurations. **MAGR**, the 2-interaction of `MariaDB` and `Gradle` (16.179% out of 35.70%

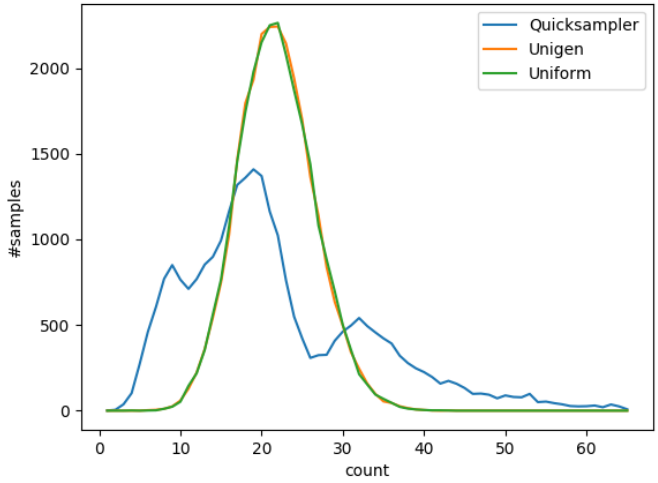
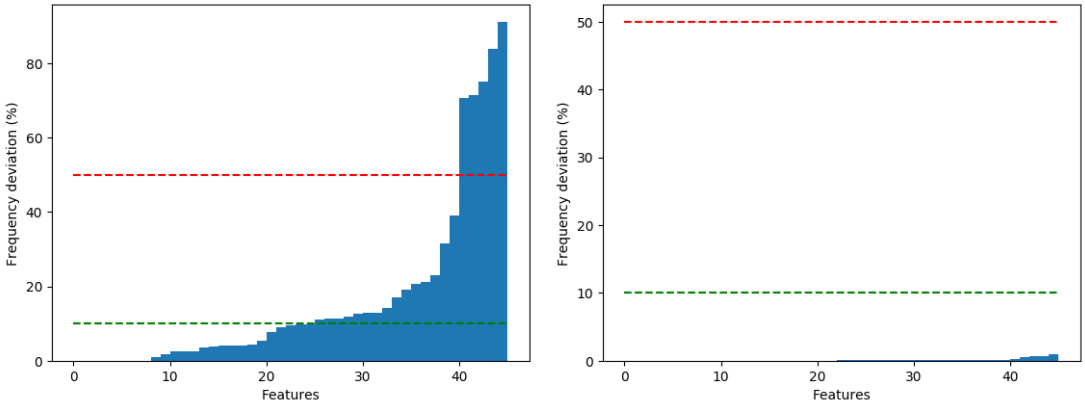


Figure 2.13: JHipster feature model: comparison of UniGen, QuickSampler and ground truth (uniform)



(a) QuickSampler (b) UniGen

Figure 2.14: Frequency deviations on the JHipster Feature Model

feature	f_{obs}	f_{th}	*dev*	*rdev*
MongoDB	0.039	0.018	116.0	116.0
Uaa	0.248	0.171	45.0	45.0
ElasticSearch	0.408	0.485	16.0	-16.0
Hibernate2ndLvlCache	0.573	0.647	11.0	-11.0
SocialLogin	0.237	0.268	11.0	-11.0
Docker	0.545	0.500	9.0	9.0
MariaDB	0.302	0.324	7.0	-7.0
Gradle	0.518	0.500	4.0	4.0
Monolithic	0.651	0.675	4.0	-4.0
EhCache	0.313	0.324	3.0	-3.0

Table 2.5: Features involved in JHipster bugs and their frequencies deviations in QuickSampler

of failures): It is the most severe source of bugs. Yet `MariaDB` is under-represented (-7%) in QuickSampler samples while being important and present in 32% of configurations. **UADO**, the 2-interaction of `Docker` and `Uaa` (6.825 % out of 35.70% of failures). Both features are over-represented (resp. +45% and +9%). As a result, there are more chances to find this fault. **OASQL**, the 2-interaction of `Uaa` and `Hibernate2ndLvlCache` (2.438 % out of 35.70% of failures). Unfortunately `Hibernate2ndLvlCache` is under-represented (-11%) despite a high presence in all configurations (65%). **UAEH**, the 2-interaction of `Uaa` and `EhCache` (2.194% out of 35.70% of failures). `EhCache` is sightly under-represented (-3%). **MADO**, the 4-interaction of `MariaDB`, `Monolithic`, `Docker` and `ElasticSearch` (5.59% out of 35.70% of failures). `ElasticSearch` is under-represented (-16%, only 40% of appearance versus 48% theoretically), as well as `MariaDB` and `Monolithic`.

Overall, QuickSampler is not uniform but is fortunate to over-represent `Uaa` (a feature involved in 3 interaction faults) while other large deviations have luckily no incidence on bug finding.

Practical implications for JHipster. In practice, executing and testing a JHipster configuration has a significant cost in resources and time (10 minutes on average per configuration). The exhaustive testing of all configurations at each commit or release is not an option. Developers and maintainers of JHipster rather have a limited testing budget at their disposal (*i.e.*, a dozen of configurations, see Section 2.3.1). As a result, we cannot take the whole sample of QuickSampler or UniGen and we rather need to take an excerpt of this sample. Various sub-sampling strategies can be considered [211, 315] either based on random, t-wise, *etc.* Without an uniform distribution, the sub-sampling process will operate over a non-representative configuration set, which may incidentally promote or underestimate some features. Overall, it is an open issue how to effectively sub-sample out of UniGen and QuickSampler solutions. Whatever the sub-sampling strategy would be, our study has shown that the sample of QuickSampler is not representative of the real features' distribution of JHipster while UniGen is uniform.

replication

We provide a Git repository with all feature models of the benchmarks, Python scripts to execute experiments, Python scripts to compute plots, figures, and statistics of this study as well as additional ones, and instructions to reuse our work: https://github.com/diverse-project/samplingfm

2.4 In search of the right variability language and models

The content of this section is adapted from the following publication:
M. Alférez, M. Acher, J. A. Galindo, B. Baudry and D. Benavides, 'Modeling Variability in the Video Domain: Language and Experience Report', *Software Quality Journal*, vol. 27, no. 1, pp. 307–347, 2019. DOI: 10.1007/s11219-017-9400-8. https://doi.org/10.1007/s11219-017-9400-8

Variability techniques have been successfully applied in many domains such as automotive, avionics, printers, mobile, or operating systems [251, 33]. However, different application domains pose specific challenges to variability engineering, both in terms of modelling language and implementation. Practitioners need empirically-tested techniques and languages for efficiently modelling and implementing variability in a systematic and scalable manner. Berger *et al.* [59] warn that the lack of experience reports on variability modelling techniques may impede the progress of variability research. This questioning is especially relevant *w.r.t.* to my own previous research: To what extent are prior works on feature modelling applicable?

Though I have always tried to work with concrete variability problems in different domains, research in software engineering seems an endless beginning. In 2013, I had the opportunity to confront existing variability techniques in the video domain and in an industrial setting (let us call it the MOTIV project). To overcome the specific issues of MOTIV, we developed a new variability modelling language, called VM. In this section, I want to report on our experience. Specifically, we address the following research questions:

- *RQ1: What are the practical considerations of applying our variability language VM?* We rely on different dimensions of the framework of Savolainen *et al.* [274] for reporting on our variability experiences carried out in an industrial setting.

- *RQ2: What are the practical benefits of a variability-based approach?* As we are in an *applied research* context, we aim to identify the improvements of our proposal with regards to existing industrial practice.

- *RQ3: What are the commonalities and differences between constructs of VM and state-of-the-art variability languages?* We discuss the literature and provide a comparison table.

In the remainder of this section, I briefly introduce the MOTIV project and the variability approach based on VM. I then address RQ1 – RQ3.

MOTIV: Industrial Problem and Overview of the Solution

Video analysis systems are ubiquitous and crucial in modern society [242, 252]. Their applications range from video protection, crisis monitoring, to crowd analysis. *Video sequences* (videos in short) are acquired, processed and analyzed to produce numerical or symbolic information. The corresponding information typically raises alerts to human observers in case of interesting situations or events.

Depending on the goal of video sequence recognition, signal processing algorithms are assembled in different ways. Each algorithm is a complex piece of software, specialized in a specific task (*e.g.*, segmentation, object recognition, tracking). Even for a specific task, a one-size-fits-all algorithm, capable of being efficient and accurate in all settings, is unlikely. The engineering of video sequence analysis systems, therefore, requires to choose and configure the right combination of algorithms [252].

The goal of the MOTIV project was to improve the evaluation of computer vision algorithms such as those used for surveillance or rescue operations. Two companies were part of the MOTIV project as well as the DGA (the French governmental organization for defense procurement). The two companies develop and provide algorithms for video analysis. A targeted scenario is usually as follows. First, airborne or land-based cameras capture on-the-fly videos. Then, a video processing chain analyzes videos to detect and track objects, for example, survivors in a natural disaster. Based on that information the military personnel triggers a rescue mission quickly based on the video analysis information. The DGA typically consumes video algorithms of the two companies for implementing the processing chains. The diversity of scenarios and signal qualities poses a difficult problem for all the partners of MOTIV: which algorithms are best suited given a specific application? From the consumer side (DGA), how to choose, select and combine the algorithms? From the provider side (the two companies), how to guarantee that the algorithms are appropriate for the targeted scenarios and robust to varying situations?

In practice, the engineering of such systems is an iterative process in which algorithms are combined and tested on various kinds of inputs (video sequences). Practitioners can eventually determine what algorithms are likely to fail or excel under certain conditions before the actual deployment in realistic settings such as using those algorithms in rescue operations. Admittedly, practitioners rely on empirical and statistical methods, based on numerous metrics. Also, the major barrier remains to find a *suitable, comprehensive input set of video sequences for testing analysis algorithms*.

Actual Practice and Early Attempts. The current testing practice is rather manual, very costly in time and resources needed, without any qualitative assurance (*e.g.*, test coverage) of the inputs. Specifically, our partners need to collect videos to test their video analysis solutions and detection algorithms. They estimated that an input data set of 153000 videos (of 3 minutes each) would correspond to 320 days of video footage and requires 64 years of filming outdoors (working 2 hours a day). These numbers were calculated at the starting point of the project, based on the previous experiences of the partners. Moreover videos themselves are not sufficient; video practitioners need also to annotate videos in order to specify the expected results (*i.e.*, ground truths) of video algorithms. This activity increases the amount of time and effort as well.

Another possible approach is to modify or transform existing videos. The first attempt was therefore to create a video generator for producing customized videos – based on user preferences that were hard-coded during the first versions. For deriving a variant, our partners had to manually comment lines or modify variable values directly in the video generator code to change the physical properties and objects that appear in each video.

When the video generator was more mature, the partners together with us decided to create configuration files to communicate input values instead of scattering the parameters in different source code files. In particular, they employed Lua configuration files that have a simple structure based the pattern *parameter* = *value*. Then, developers used Lua code and proprietary C++ libraries, developed by a MOTIV partner, to process those configuration files and execute algorithms to alter, add, remove or substitute elements in base videos. Lua is a widely used programming language (`http://www.lua.org/`). It should be noted that the generator not only computes a video variant but also some annotations, thus avoiding the manual specification of expected results. The Lua configuration files used helped to decouple implementation from input data.

However the effort still remains tedious, undisciplined, and manual. It was still hard to construct large datasets – our partners have to manually modify the configuration file, with the additional problem of setting non conflicting values. Also they ignore what kinds of situations are covered or not by the set of videos, *i.e.*, some kinds of videos may not be included in the dataset. Overall, more *automation* and *control* were needed to synthesize video variants in order to cover a large diversity of testing scenarios.

Variability Modelling Approach: An Overview

To overcome previous limitations, we introduced a variability-based approach (see Figure 2.15). The key idea is that practitioners now explicitly model variability using VM a variability language we have developed in the project. This approach was developed from scratch but clearly inspired with: First, FaMa [57] and FAMILIAR [16] languages as because they were developed by members of the project. Second, by other more recent variability modelling languages such as Clafer [40]. Using the VM language, variability is typically expressed in terms of mandatory, optional, mutually exclusive features, but also attributes for encoding non-Boolean values (integers, floats or strings). As detailed in [27], numerous other advanced constructs can be used for describing what can vary in a video. An excerpt of a variability model written in VM is depicted in Figure 2.15: constructs like **delta** or **@NT** are usually not available with feature modelling languages. **delta** is a way to reduce the number of acceptable numeric values. For instance, “real distractors.butterfly_level [0.0..1.0] delta 0.25" states that possible values are 0.0, 0.25, 0.5, 0.75, and 1.0. It allows to control the domain values and solvers can exploit this construct to operate over a reduced space of possible values. **@NT** (for non translatable) states that constraint solvers should not encode some features or attributes as variables. For instance, it has no interest to reason about “sequence.comment". The intent is that experts can reduce the complexity of the constraint problem and solvers can better scale.

A VM model is an abstraction of all possible Lua configuration files. It has the merit (1) of enforcing constraints over attributes and values (precluding invalid configurations); (2) reasoning techniques (*e.g.*, constraint programming or satisfiability techniques) can operate over the model to assign values to features and attributes in an efficient and sound way; (3) generative techniques can process the model to automatically produce configuration files.

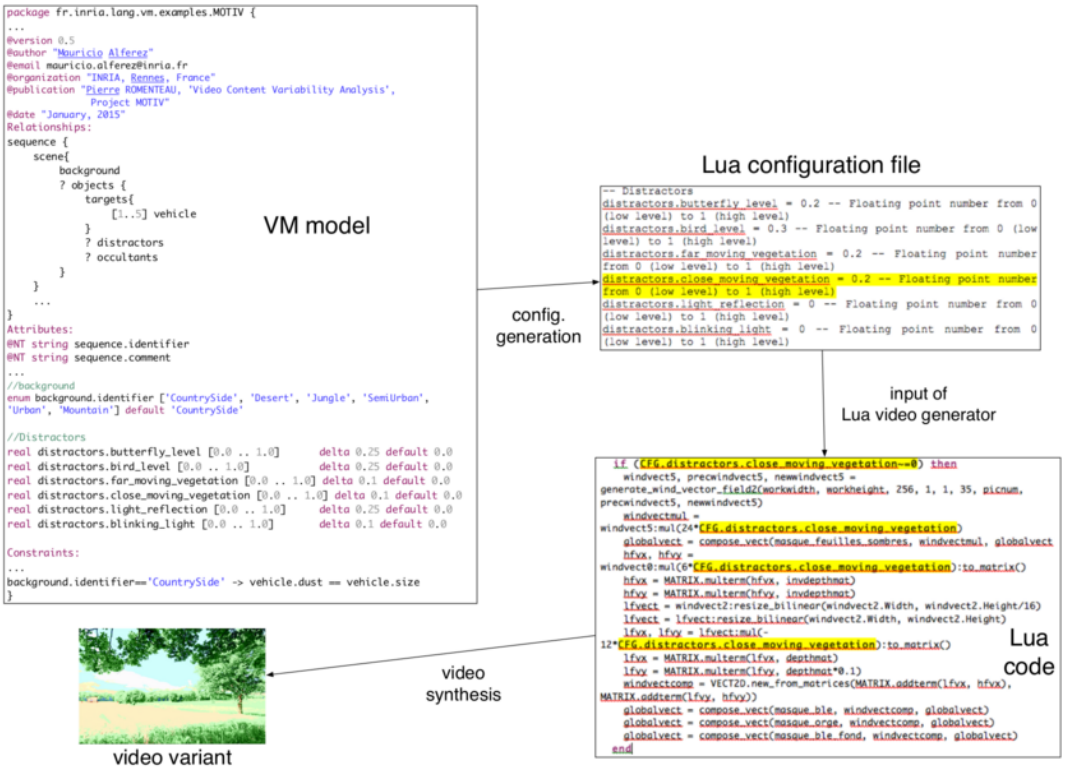


Figure 2.15: From a VM model, we generate configurations that are fed to a Lua generator for synthesizing video sequences. VM configurations are obtained through sampling and constraint solving (see previous sections). A VM configuration is translated into a Lua configuration file (each feature/attribute has a corresponding Lua parameter).

Specifically we developed reasoning operations for producing configuration sets that cover the $t-wise$ criteria (while handling constraints and some objective functions over attributes). Overall we can generate Lua configuration files (from a VM model) that the video generator can exploit to produce numerous video variants. The level of illumination, the distractors, or the blur levels have three different values and are examples of what can vary in a video.

Practical Considerations (RQ1)

I report an excerpt of key lessons learned; the interested reader can find more details in [27].

Model creation. This task was the one that took more time; it required to understand the domain, the requirements, and to discuss with video experts. We produced six different versions of the video variability model during a period of about nine months. These versions were made after four large meetings with all the project partners (these meetings focused on different topics apart from variability modelling, including administrative issues and technical issues in the video analysis domain), and two individual meetings with the main developer of the video generator.

Completeness. *"How complete is the feature model?"* [274]. In the video domain, the resources are limited and there are too many possible videos to generate. Therefore, we modeled extra variability including as much information as possible to restrict the number of combinations of features and attribute values to the minimum.

Stakeholders. *"Who puts effort into and who gains the benefits of the model? What knowledge about feature modelling methods in general and the product line in question do the stakeholders have?"* [274]. VM was developed mainly by a team composed of two people (a doctoral and postdoctoral researcher) and one lecturer at Inria, which knew about product lines, and some feature modelling methods. This team created the language infrastructure, implemented a translation from VM to a CSP (presented on a previous work [124]), and work to connect the VM tool with the video generator. The main video generator developer is also a video expert that provided feedback for improving the VM design. In addition, he wrote an initial and not exhaustive description of the important aspects that may be varied in a video that were important to test a predefined set of video algorithms. Based on the description, the development team wrote the first version of the VM model and used that version to communicate with the rest of the partners in the following meetings. Stakeholders from the DGA provided comments that were addressed in the following iterations and model versions. However, the role of the members of the DGA was mainly to review that the video sequences synthesized were realistic. Taking into account the variety of stakeholders, we took the decision of dividing the VM language by blocks, each one addressing a different concern. Video experts without too much technical expertise can focus on concerns described in the relationships, model information, definitions, and objectives blocks. Developers and video experts with a programming background can focus on adding annotations, constraints, deltas, or further defining the objectives and attributes blocks.

Domain. *"Does the model represent the problem or solution domain? Does the model represent a current or planned product line?"* [274]. VM can be used to model either the problem or solution domain; both sides influence the design of VM or to represent a current or future product line. In our experience, we first modeled the video domain from a problem domain perspective (during meetings with domain experts), and then realize variability through configuration files and Lua code.

Commonality. *"How much commonality is represented?"* [274]. Although VM mainly targets variability modelling, it also supports commonality modelling through the notion of common features. Attribute values can be fixed as well. However, common attributes are not described in details to not make the model unnecessarily too large. In our case, commonality rather emerged for specializing the variability model to specific testing scenarios (*e.g.*, when fixing a value to the luminance attribute).

Correspondence. *"What elements of the product line does the feature model correspond to?"* [274]. 1-to-1 mappings between features in the problem space and their realizations in the solution space ease their co-evolution. For example, many features in the video domain VM model have a 1-to-1 relationship with code modules that implemented the video generator. In a similar way, feature attributes tend to match input parameters of Lua functions. Using 1-to-1 mappings is not a strict rule. In fact, we also modeled features that are not mapped to any specific module to group other features or attributes. For example, the feature "objects" does not map directly to any module, but helped to group conceptually the "vehicles" and "man" features that have concrete mappings to the code. One important highlight regarding correspondence was that we decided not to use VM to model all possible variability in a video sequence. In particular, we decided not to model or provide constructs to determine the time

and order in which events happen in a video sequence or the path of moving vehicles and people in an scene. Our partners already had a way to orchestrate events and to create and manage paths in predefined backgrounds. However, we are considering the integration of those aspects in future versions of VM.

Constraints. *"What do the constraints represent?"* [274] VM addresses the challenge of managing and representing constraints through a set of functions and operations over features, attributes and sets of features (*e.g.*, "ClonesOf"). Constraints are also important for *specializing* the VM model to specific testing scenarios. For instance, experts want to synthesize only videos with a specific background (such as desert or urban) or luminance; some values are thus fixed, but the other features or attributes are still subject to variations.

Notation. *"What constructs and representation should different stakeholders use?"* [274] The VM language provides a *textual* notation for expressing variability. In contrast to diagrammatic languages, participants continue to use well-established efficient tools in the industry such as code editors. Furthermore, numerous attributes, meta-information, and cross-tree constraints have been specified; by construction they are textual information.

Benefits of Modelling Variability (RQ2)

In this section, we retrospectively compare the three approaches used throughout the project. In *applied research*, the objective is to create technology that is better in some manner than those already developed [164]. Here we show the improvements of a variability-based approach in terms of *automation* and *control* thus participating to the creation of *large-scale* datasets (videos).

Non generative approach (A_0) At the starting point of the project, the testing practice was either to collect existing videos or to film new videos. Then algorithms that perform over the videos and metrics are computed to determine the accuracy or the response time. Based on the results and statistical methods, practitioners can determine the strengths and weaknesses of their solutions.

We recall here two severe limitations. First, not only is the collection of videos a costly and time-consuming activity but also the annotations to specify what are the expected results and thus evaluate the algorithms. Another limitation is that the collected dataset is usually small in size and not representative of testing scenarios. Some benchmarks exist (for example, for event recognition, see, *e.g.*, [235]) but are specific to vision analysis tasks and cannot be seamlessly reused (*e.g.*, for military scenarios).

Summary: the practice we have observed at the beginning of the project suffers from a lack of automation – precluding the establishment of large datasets – and a lack of control over the testing videos.

With the generator (A_1) With the development of a video generator (see Figure 2.15), practitioners can envision to build a larger, more diverse, and representative dataset of videos for testing their algorithms. At that time, the elaboration of a dataset consists in setting some values to a configuration file and then executes the generator to produce a variant.

Compared to a non generative approach (A_0), the use of a video generator has the advantage of providing (1) more automation: there is neither the need to film nor to annotate videos; (2) more control: practitioners can tune the parameters to produce the video variant they want.

However some limitations remain. The approach still requires human intervention for specifying *each* configuration file. The setting of values is tedious and impractical when a large number of configuration files has to be set. This is evident in the current VM model that notably describes 84 attributes (out of which a large proportion are reals) and 2,161,711 individual[11] attribute values in total can be set.

With a manual elaboration of configuration files, practitioners eventually ignore what test cases (video variants) are covered. Moreover it is hard to augment the dataset because of the lack of automation and the lack of knowledge of what test cases are missing. This covering knowledge is very important since the most situations are covered, the more practitioners are confident in terms of robustness, performance and reliability of their algorithms. It is especially important for an institution like DGA to have a strong coverage guarantee. It is as important for the two industrial partners to cover a maximum kind of situations and determine if the algorithms behave accordingly.

Another limitation related to the previous observations is the difficulty of controlling the synthesis for the synthesis of *specific* datasets. For instance, the synthesis of video variants in which the global luminance only varies between, say, 0.6 and 0.8, is tedious and error-prone. In this case, practitioners have to manually set the value while ensuring it is not dependent of another parameter; the random modification of the luminance values and the whole process should be repeated for each configuration file. It is again impractical *w.r.t.* to the number of attributes and possible domain values.

Summary: the development of a video generator still suffers from a lack of automation – precluding the establishment of large and diverse datasets – and a lack of control over the testing videos.

Variability-based approach (A_2) The use of a VM model to pilot the generator allows practitioners to have more automation and more control. Instead of manually modifying each configuration file (see A_1), an automated operation processes a VM model and fully generates all configuration files. The effort of the practitioners is thus dramatically reduced. Another benefit is that constraints over or across parameters' values are valid by construction.

A variability-based approach also helps to *specialize* the synthesis. Different alternatives can be employed for this purpose:

- putting additional constraints and specializing the VM model for specific scenarios. For instance, a specific Background (*e.g.*, Urban) can be set up since the application is known to be deployed in a specific military ground. In turn the testing machinery will then consider only configurations with Urban. The benefit is that practitioners can focus on specific testing scenarios, specializing the test suite to realistic cases;

- optimizing different objective functions over attributes: practitioners can specify the relative importance or cost of a feature, fix some parameters, *etc.* Again it aims at customizing test suite to fit realistic needs;

- precluding some features or attributes, not relevant for testing, with the use of meta-information.

[11] Each attribute has a domain size, which corresponds to the number of individual values an attribute can take. We consider *deltas* [27] for the computation of domain size. It should be noted that the number of possible configurations is significantly greater than the sum of possible individual values – since a configuration is a combination of individual attribute values.

In terms of *covering*, a variability-based approach grants, by construction, the validity of the T-wise (*e.g.*, pair-wise) criterion. The covering criterion can be combined with other specialization mechanisms.

Finally, we were able to inject constraints into the variability model for avoiding the generation of irrelevant videos. This increase in quality was possible because of an explicit variability model. Without such an abstraction and without a variability approach in general, we simply could not realize our idea and thus enforce the video generator.

Summary: the introduction of variability techniques on top of the generator induces important benefits in terms of automation and control. Practitioners can now synthesize large, suitable, and diverse datasets – something practically impossible with previous practices A_0 and A_1.

Comparison with Existing Variability Languages (RQ3)

Numerous languages, being textual or graphical, have been designed to model variability. For instance, feature models have become more and more sophisticated since 1990 and their dialects have been detailed in comprehensive surveys, for example, by Schobbens *et al.* [277], Benavides *et al.* [55] and Eichelberger and Schmid [117]. Table 2.6 summarizes the comparison of VM with some representative languages in terms of the requirements that they address as a goal. We have relied on recent comprehensive surveys [59, 117] to select the languages. It is important to note that *while most of the characteristics are not that novel, they could not be found in a single language or are addressed but with restrictions*.

Most common characteristics. Boolean constructs of feature models (as supported by FODA [175], FDL [109], SXFM [216], VELVET [266] or FAMILIAR [16]) are useful in the video domain, but not sufficient. New dialects (*e.g.*, UTFM [316], CLAFER [40], SALOON [256], VSL [2], TVL [85] and FAMA [57]) have emerged to overcome the expressiveness limitations of feature models, for instance, to deal with *attributes* or *multi-features*.

Most of the languages do not allow to explicitly change a *default value* for features and attributes so, we considered that they do not address that characteristic as a goal. FDL is an example of a language that addresses this characteristic with restrictions. It includes the construct “default" however, it only uses it to declare a selected-by-default atomic feature in a group and not a default attribute value.

Another case of characteristics that are addressed only partially is the *constraints*. While constraints have been addressed by all the languages, in most of the cases they did not consider constraints including features, attributes values (*e.g.*, in FODA, FDL SXFM) and multifeatures (*e.g.*, VSL and FAMA).

Less common characteristics. The *main differences between VM and the other approaches* are mainly the use of meta-information associated to features or attributes. For example, VM users can include: i) *deltas*, ii) *elements definitions* –model, features and attributes information, iii) multi-ranges and multi-deltas, iv) meta-information annotations such as “not translatable", “not decidable", and “runtime", and v) objective functions. As reported in [27], our industrial experience strongly motivates the introduction of these new constructs. We also show the importance of the constructs in terms of reasoning scalability [27].

The comparison highlights two important aspects of variability languages. First, some *common* needs for modelling variability are emerging such as the support for attributes and multi-features. Second, *specific* constructs are also needed and were a prerequisite for successful adoption in the case of VM– similar observations have been reported in other domains.

Characteristic / Approach	FODA [175]	FDL [109]	SXFM [216]	FAMILIAR [16]	VELVET [266]	UTFM [316]	CLAFER [40]	Saloon [256]	VSL [2]	TVL [84]	FAMA [57]	VM
Multifeatures	○	○	○	○	○	●	●	●	●	●	○	●
Attributes	○	○	○	○	○	●	●	●	●	●	●	●
Default values	○	◑	○	○	○	○	●	○	●	●	●	●
Deltas	○	○	○	○	○	○	○	○	○	○	○	●
Elements definition	○	○	●	○	○	○	○	○	○	○	○	●
Constraints	◑	◑	◑	◑	◑	●	●	●	◑	●	◑	●
Multi-ranges, multi-deltas	○	○	○	○	○	○	○	○	○	○	◑	●
Run-time annotation	○	○	○	○	○	○	○	○	○	○	○	●
NT	○	○	○	○	○	○	○	○	○	○	○	●
ND	○	○	○	○	○	○	○	○	○	○	○	●
Objectives	○	○	○	○	○	○	●	○	○	◑	○	●

●addressed as goal, ◑addressed but with restrictions, ○not regarded as goal

Table 2.6: Summary of comparison between languages *w.r.t.* specific requirements of the MOTIV project and video domain

The variability languages do not address some of the requirements of the MOTIV project simply because they have not been design to. Similarly, some languages for variability offer specific constructs that VM does not (*e.g.*, CLAFER provides advanced specialization mechanisms [40]).

Variability languages and empirical insights: discussion of [283]. Sepulveda *et al.* performed a systematic literature review of requirements modelling languages for software product lines [283]. The study includes variability modelling languages developed from 2000 to 2013.

Interestingly our work confirms some findings of the IST article [283]. First Sepulveda *et al.* report that *"some constructs (feature, mandatory, optional, alternative, exclude and require) are present in all the languages, while others (cardinality, attribute, constraint and label) are less common"*. Second there is a concern for generating proposals with higher levels of expressiveness. It is in line with our previous comparison of variability languages. Meanwhile our work contributes to the lack of empirical validation and adoption in industry (as identified in [283]). For example, it is stated that *"57% of the languages have been proposed by the academia, while 43% have been the result of a joint effort between academia and industry"*. Our research is precisely a tight collaboration with industrial partners to capture the right level of expressiveness for VM and to fully develop a video generator.

In search of the right variability language

This experience showed how specific needs, encountered in the MOTIV project and in the video domain, have shaped the design of a textual variability language with advanced constructs and reasoning support. We learned the important lessons from our industrial experience. First, basic variability mechanisms à la FODA – Boolean (optional) features, hierarchy, group and cross-tree constraints – are useful but not enough and attributes and multi-features are of prior importance. Second, meta-information is relevant for (1) performing efficient computer-aided analysis of VM models, and (2) controlling the generation of testable configurations (*e.g.*,, to focus on specific attributes of features). Third, different iterations were needed to identify and implement additional specific constructs (*e.g.*, deltas and binding mode) when connecting VM to the video generator developed by the industrial partners [7].

The point of this section is not to present yet another variability language. We rather want to highlight the specific requirements we faced throughout the project, in the video domain, leading to the design and use of existing (or novel) variability constructs. Our experience, as others [59, 115, 60], question the existence of a one-size-fits-all variability solution applicable in any industry. Yet some common needs for modelling variability are becoming apparent (*e.g.*, support for attributes and multi-features [40, 90, 327]).

To conclude this part, I want to highlight two intiatives I have been involved in. First, the Common Variability Language (CVL) a domain-independent language for specifying and resolving variability. CVL, originally supported by the OMG, brought together several academic and industrial partners. CVL pursued a larger goal than providing a feature modelling language (like VM), with a strong emphasis on expressing variability withing base models. Though CVL is no longer active, ideas developed there are worth revisiting. Second, the MODEVAR workshop `https://modevar.github.io/` brought together researchers, tools or developers and calls to find a possible consensus on a simple feature modelling language. Overall, the definition of a "common" variability language is still an open question in the community – the VM language is one experience among others.

In search of the right variability model

It is one thing to have a language, it is another to develop models with this language. Another lesson learnt with the MOTIV project is the difficulty of elaborating sound and complete variability models, especially the writing of constraints among features and domain values. We elaborate more on this aspect in Section Learning variability constraints with a more radical approach.

replication

Interested readers can find the complete implementation code and grammar of VM online as well as the variability models we elaborated in the industrial project https://github.com/ViViD-DiverSE/VM-Source

2.5 Wrap-up, applicability, and limitations

Modelling variability in the sense of manually developing a model of variability is a complex activity. It is easy to forget a constraint and unintentionally alter the configuration semantics. I have first described a comprehensive support for the widely used formalism of feature model. The FAMILIAR language can be used to manipulate an algebra of feature models with well-defined properties and on top of satisfiability solvers. This support proved to be crucial for various compositional scenarios of software product line engineering. I then studied how feature models relate to product comparison matrices aka PCMs. I showed that there is a gap between the two formalisms and reported on the design of a specific metamodel to encode PCMs. Our journey with variability formalisms also included the design of the VM language with advanced constructs in such a way video experts can comprehensively express variability information.

As shown in this chapter, modelling variability can be used in a variety of scenarios: for testing a system, for developing and controlling generators, for abstracting multiple concerns of multiple system, or for analysing a domain. A variability model is mandatory to set up a sampling strategy for effectively finding bugs or controlling the generation of a diverse set of video variants. Throughout this line of research, I have considered different application domains (medical imaging, video processing, Web applications, computer vision, *etc.*). modelling has the advantage of integrating knowledge about a domain or a system. Developers, domain experts, and testers can express their intention, define a scope, and control in a fine-grained way the variabilty information. According to Collins dictionary, "-ing" as a suffix forming nouns refers to "the action of, process of, result of, or something connected with the verb". Model*ing* is one kind of process, with possibly different iterations to find the good abstraction or the good language.

This process, however, has some instrinsic limitations. First, modelling variability is time-consuming. The JHipster and MOTIV cases show that several months were needed to elaborate the variability models. Second, the process is error-prone. Features can be forgotten as well as complex relationships among multiple models, features, or numerical attributes. Third, systems frequently evolve: new versions are released, together with a new code and possibly new features and constraints. The manual effort can hardly be repeated. It is partly why, for instance, we did not continue the testing effort for subsequent versions of JHipster. Finally, the knowledge can be imprecise or simply difficult to formally express.

All these limitations have pushed us to look at two research directions: reverse engineering and learning software variability.

Chapter 3

Reverse Engineering Software Variability

In this chapter I present a set of methods and techniques to reverse engineer models of variability. Chikosfky and Cross define reverse engineering as *"the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction"* [82]. Though one section of this chapter is about architecture and components (plugins), the goal I am pursuing is both more specific and general: It is to create variability representations of artefacts in another form or at a higher level of abstraction. This chapter mainly focuses on scenarios in which persons (domain experts, Web developers, architects, *etc.*) automatically obtain models of variability out of existing descriptions of a system or a domain.

Section 3.1 describes the foundations to synthesize attributed feature models out of a class of product comparison of matrices. Section 3.2 develops methods and techniques to synthesize product comparison matrices out of textual descriptions of individual products. Section 3.3 develops methods and techniques to reverse engineer Web configurators (client-side). Section 3.4 develops methods and techniques to supervise and control the reverse engineering of architectural feature models throughout evolutions.

Contents

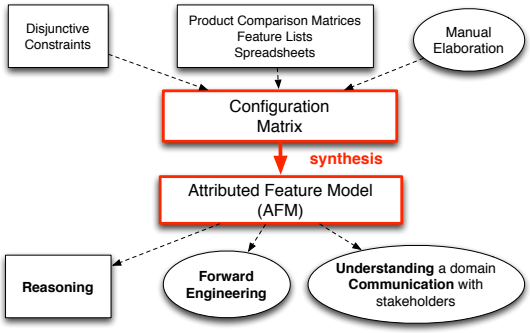


Figure 3.1: Core problem: synthesis of attributed feature model from configuration matrix

3.1 Synthesizing attributed feature models out of tabular data

The content of this section is adapted from the following publication:
G. Bécan, R. Behjati, A. Gotlieb and M. Acher, 'Synthesis of Attributed Feature Models From Product Descriptions', in *19th International Software Product Line Conference (SPLC'15)*, (research track, long paper), Nashville, TN, USA, Jul. 2015

Section Feature models and product comparison matrices has established syntactic and semantic relationships between product comparison matrices and feature models. From now on, we will use the term *configuration matrix* to refer to a subset of product comparison matrices (see Figure 2.8, page 38) that can be automatically translated to feature models. This subset contains non-Boolean information and the challenge is to synthesize *Attributed Feature Models (AFMs)*, a formalism that is more expressive than basic feature models considered in *e.g.*, Section 2.1 and 2.3.1. Overall we are addressing a reverse engineering scenario in which we want to create representations of an artefact (configuration matrix) in another form (AFM).

Figure 3.1 summarizes our motivation for synthesizing an AFM from a configuration matrix. As shown in the upper part of Figure 3.1, the input to the synthesis algorithm is a configuration matrix (see Definition 5).

Definition 5 (Configuration matrix) *Let* $\mathbf{c}_1, ..., \mathbf{c}_M$ *be a given set of configurations. Each configuration* $\mathbf{c}_i$ *is an N-tuple* $(c_{i,1}, ..., c_{i,N})$*, where each element* $c_{i,j}$ *is the value of a* variable V_j*. A variable represents either a feature or an attribute. Using these configurations, we create an* $M \times N$ *matrix* $\mathbf{C}$ *such that* $\mathbf{C} = [\mathbf{c}_1, ..., \mathbf{c}_M]^t$*, and call it a* configuration matrix*.*

Configuration matrices act as a formal, intermediate representation that can be obtained from various sources, such as (1) spreadsheets and product comparison matrices (*e.g.*, see [52]), (2) disjunction of constraints, or (3) simply through a manual elaboration (*e.g.*, practitioners explicitly enumerate and maintain a list of configurations [58]).

For instance, let us consider the domain of Wiki engines. The list of features supported by a set of Wiki engines can be documented using a configuration matrix. Figure 3.2 is a very simplified configuration matrix, which provides information about eight different Wiki engines.

Id	License Type	License Price	Language Support	Lang uage	WYSI WYG
W1	Commercial	10	Yes	Java	Yes
W2	NoLimit	20	No	–	Yes
W3	NoLimit	10	No	–	Yes
W4	GPL	0	Yes	Python	Yes
W5	GPL	0	Yes	Perl	Yes
W6	GPL	10	Yes	Perl	Yes
W7	GPL	0	Yes	PHP	No
W8	GPL	10	Yes	PHP	Yes

Figure 3.2: A configuration matrix for Wiki engines.

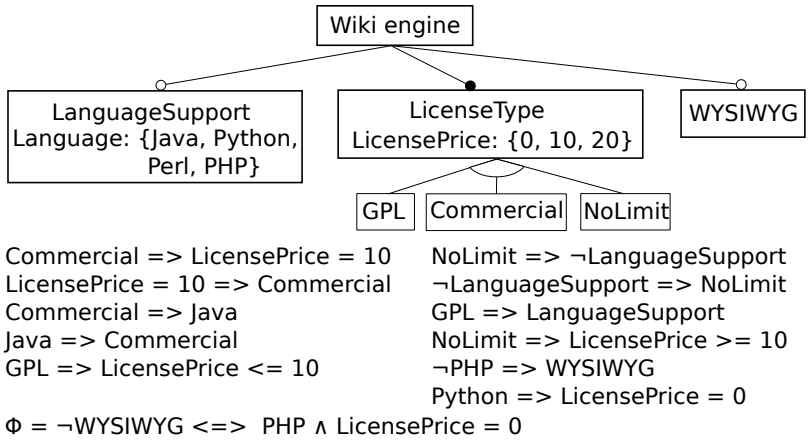


Figure 3.3: One possible attributed feature model for representing the configuration matrix in Figure 3.2

The resulting AFM (see lower part of Figure 3.1) can as well be used to document a set of configurations and open new perspectives. First, state-of-the-art reasoning techniques for AFM can be reused (*e.g.*, [302, 40, 90, 236]). Second, the hierarchy helps to structure the information and a potentially large number of features into multiple levels of increasing detail [97]; it helps to understand a domain or communicate with other stakeholders [55, 58, 97]. Finally, an AFM is central to many product line approaches and can serve as a basis for forward engineering [33] (*e.g.*, through a mapping with source code or design models).

Overall, configuration matrices and feature models are semantically related and aim to characterize a set of configurations. The two formalisms are complementary as they propose different views on the same product line; we aim to better understand the gap and switch from one representation to the other. For instance, Figure 3.3 depicts an attributed feature diagram as well as constraints that together provide one *possible* representation of the configuration matrix of Figure 3.2.

RC	::=	*bool_factor* '⇒' *bool_factor*
bool_factor	::=	**feature_name** | '¬' **feature_name** | *rel_expr*
rel_expr	::=	**attribute_name** *rel_op* **num_literal**
rel_op	::=	'>' | '<' | '≥' | '≤' | '**=**'

Figure 3.4: The grammar of readable constraints.

Attributed Feature Models

Several formalisms supporting attributes exist [40, 85, 56, 118]. In this chapter, we consider an extension of FODA-like feature models including attributes and inspired from the FAMA framework [55, 56]. An AFM is composed of an attributed feature diagram (see Definition 6 and Figure 3.4) and an arbitrary constraint (see Definition 7).

Definition 6 (Attributed Feature Diagram) *An attributed feature diagram (AFD) is a tuple* $\langle F, H, E_M, G_{MTX}, G_{XOR}, G_{OR}, A, D, \delta, \alpha, RC\rangle$ *such that:*

- F *is a finite set of boolean features.*
- $H = (F, E)$ *is a rooted tree of features where* $E \subseteq F \times F$ *is a set of directed child-parent edges.*
- $E_M \subseteq E$ *is a set of edges defining mandatory features.*
- $G_{MTX}, G_{XOR}, G_{OR} \subseteq P(E \backslash E_m)$ *are sets of feature groups. The feature groups of* G_{MTX}, G_{XOR} *and* G_{OR} *are non-overlapping and each feature group is a set of edges that share the same parent.*
- A *is a finite set of attributes.*
- D *is a set of possible domains for the attributes in* A.
- $\delta \in A \rightarrow D$ *is a total function that assigns a domain to an attribute.*
- $\alpha \in A \rightarrow F$ *is a total function that assigns an attribute to a feature.*
- RC *is a set of constraints over* F *and* A *that are considered as human readable and may appear in the feature diagram in a graphical or textual representation* (e.g., *binary implication constraints can be represented as an arrow between two features).*

A domain $d \in D$ *is a tuple* $\langle V_d, 0_d, <_d\rangle$ *with* V_d *a finite set of values,* $0_d \in V_d$ *the null value of the domain and* $<_d$ *a partial order on* V_d. *When a feature is not selected, all its attributes bound by* α *take their null value,* i.e., $\forall(a, f) \in \alpha$ *with* $\delta(a) = \langle V_a, 0_a, <_a\rangle$, *we have* $\neg f \Rightarrow (a = 0_a)$.

For the set of constraints in *RC*, formally defining what is human readable is essential for designing automated techniques that can synthesize *RC*. In this paper, we define *RC* as the constraints that are consistent with the grammar in Figure 3.4. Some examples of such constraints can be found in the bottom of Figure 3.3. We consider that these constraints are small enough and simple enough to be human readable. In this grammar, each constraint is a binary implication, which specifies a relation between the values of two attributes or features. Feature names and relational expressions over attributes are the boolean factors that can appear in an implication. Further, we only allow natural numbers as numerical literals (num_literal).

The grammar of Figure 3.4 and the formalism of attributed feature *diagrams* (see Definition 6) are not expressively complete regarding propositional logics. Therefore the formalism of AFD cannot represent any set of configurations (more details are given in [49]). To enable accurate representation of any possible configuration matrix, an AFM is composed of an AFD and a propositional formula:

Definition 7 (Attributed Feature Model) *An attributed feature model is a pair $\langle AFD, \Phi \rangle$ where AFD is an attributed feature diagram and Φ is an arbitrary constraint over F and A that represents the constraints that cannot be expressed by RC.*

Example. Figure 3.3 shows an example of an AFM describing a product line of Wiki engines. The feature WikiMatrix is the root of the hierarchy. It is decomposed in 3 features: LicenseType which is mandatory and WYSIWYG and LanguageSupport which are optional. The xor-group composed of GPL, Commercial and NoLimit defines that the wiki engine has exactly 1 license and it must be selected among these 3 features. The attribute LicensePrice is attached to the feature LicenseType. The attribute's domain states that it can take a value in the following set: {0, 10, 20}. The readable constraints and Φ for this AFM are listed below its hierarchy (see Figure 3.3). The first one restricts the price of the license to 10 when the feature Commercial is selected.

The main objective of an AFM is to define the valid configurations of a product line. A configuration of an AFM is defined as a set of selected features and a value for every attribute. A configuration is valid if it respects the constraints defined by the AFM (*e.g.*, the root feature of an AFM is always selected). The set of valid configurations corresponds to the *configuration semantics* of the AFM (see Definition 8).

Definition 8 (Configuration semantics) *The configuration semantics $[\![m]\!]$ of an AFM m is the set of valid configurations represented by m.*

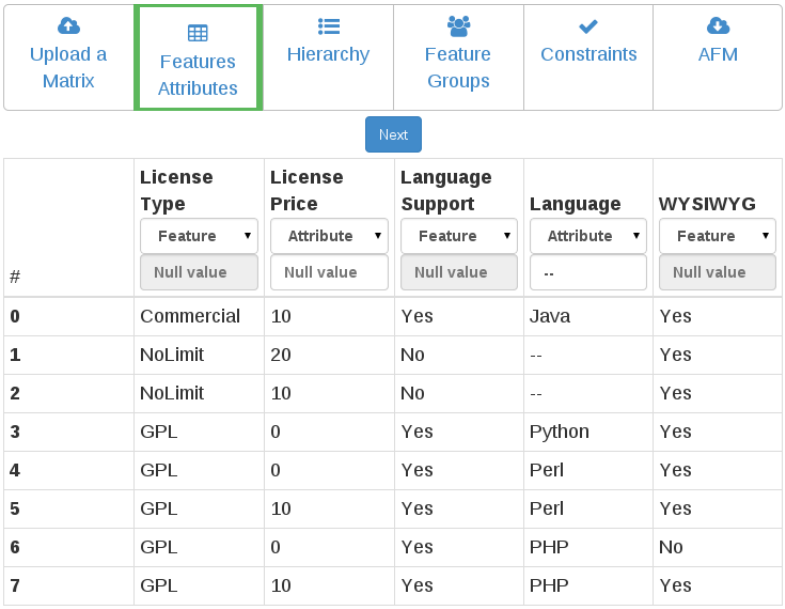
Two main challenges of synthesizing an AFM from a configuration matrix are (1) preserving the configuration semantics of the input matrix; and (2) producing a maximal and readable diagram for a further exploitation by practitioners (see Figure 3.1).

To avoid the synthesis of a trivial AFM (*e.g.*, an AFM with the input matrix encoded in the constraint Φ and no hierarchy, *i.e.*, $E = \emptyset$), we define the property of maximality for attributed feature models (see details in [49]). Given a set of configurations sc, the problem is to synthesize an AFM m such that $[\![sc]\!] = [\![m]\!]$ (*i.e.*, the synthesis is sound and complete) and m is maximal.

Synthesis parametrization

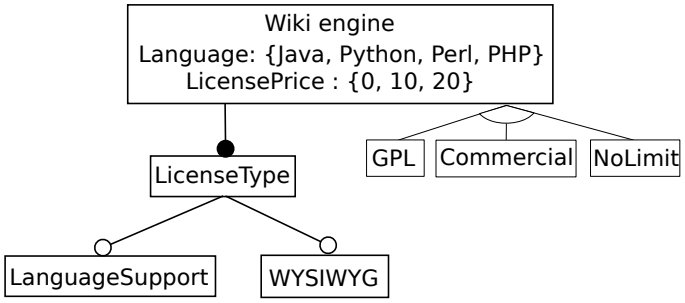
Despite the maximality property, the solution to the problem may not be unique. Given a set of configurations (*i.e.*, a configuration matrix), multiple maximal AFMs can be synthesized.

It has already been observed for the synthesis of Boolean FMs [47, 284, 285]. Extending boolean feature models with attributes exacerbates the situation. In some cases, the place of the attributes and the constraints over them can be modified without affecting the configuration semantics of the synthesized AFM.



#	License Type Feature Null value	License Price Attribute Null value	Language Support Feature Null value	Language Attribute --	WYSIWYG Feature Null value
0	Commercial	10	Yes	Java	Yes
1	NoLimit	20	No	--	Yes
2	NoLimit	10	No	--	Yes
3	GPL	0	Yes	Python	Yes
4	GPL	0	Yes	Perl	Yes
5	GPL	10	Yes	Perl	Yes
6	GPL	0	Yes	PHP	No
7	GPL	10	Yes	PHP	Yes

Figure 3.6: Web-based tool for gathering domain knowledge during the synthesis



Commercial => LicensePrice = 10
LicensePrice = 10 => Commercial
Commercial => Java
Java => Commercial
GPL => LicensePrice <= 10
NoLimit => ¬LanguageSupport
¬LanguageSupport => NoLimit
GPL => LanguageSupport
NoLimit => LicensePrice >= 10
¬PHP => WYSIWYG
Python => LicensePrice = 0
¬LanguageSupport => Language = "--"

Φ = ¬WYSIWYG <=> PHP ∧ LicensePrice = 0

Figure 3.5: Another attributed feature model representing the configuration matrix in Figure 3.2

Example. Figures 3.3 and 3.5 depict two AFMs representing the same configuration matrix of Figure 3.2. They have the same configuration semantics but their attributed feature diagrams are different. In Figure 3.3, the feature WYSIWYG is placed under Wiki engine whereas in Figure 3.5, it is placed under the feature LicenseType. Besides the attribute LicensePrice is placed in feature LicenseType in Figure 3.3, whereas it is placed in feature Wiki engine in Figure 3.5.

To synthesize a unique AFM, our algorithm uses *domain knowledge*, which is extra information that can come from heuristics, ontologies or a user of our algorithm. This domain knowledge can be provided interactively during the synthesis or as input before the synthesis. Our synthesis tool (Figure 3.6 shows the workflow) provides an interface for collecting the domain knowledge so that users can:

- decide if a column of the configuration matrix should be represented as a feature or an attribute;
- give the interpretation of the cells (type of the data, partial order);
- select a possible hierarchy, including the placement of each attribute among their legal possible positions;
- select a feature group among the overlapping ones;
- provide specific bounds for each attribute in order to compute meaningful and relevant constraints for RC.

All steps are optional; in case the domain knowledge is missing, the synthesis algorithm takes arbitrary yet sound decisions (*e.g.*, random hierarchy).

Example. The domain knowledge that leads to the synthesis of the AFM of Figure 3.3 can be collected with our synthesis tool. Users specify what constitutes an attribute or a feature. For instance, the column Language represents an attribute (for which the null value is "- -"). In further step, users can specify hierarchy and also precise that, *e.g.*, "10" is an interesting value for LicensePrice when synthesizing constraints.

Synthesis algorithm

The two inputs of the algorithm are a configuration matrix and some domain knowledge for parametrizing the synthesis. The output is a maximal AFD. In complement to the AFD, we compute the constraint Φ [49]. The addition of Φ and the AFD forms the AFM. The first step of the synthesis algorithm is to extract the features (F), the attributes (A) and their domains (D, δ). A follow-up and important step of the synthesis is to extract binary implications between features and attributes. It is used to select hierarchy and synthesize *requires* constraints.

An original and costly step concerns the synthesis of relational constraints *i.e.*, all the constraints following the grammar described in Figure 3.4 and involving at least one attribute. Admittedly there is a huge number of possible constraints that respect the grammar of RC. Our synthesis algorithm relies on some domain knowledge (see Figure 3.6) to restrict the domain values of attributes considered for the computation of RC. In case the knowledge is incomplete (*e.g.*, users do not specify a bound for an attribute), we randomly choose a value in the domain of the attribute. *Example.* From the configuration matrix of Figure 3.2, we can extract the following binary implication: $\mathsf{GPL} \Rightarrow \mathsf{LicensePrice} \in \{0, 10\}$. We also note that the domain of LicensePrice is $\{0, 10, 20\}$. Therefore, the right side of the binary implication can be rewriten as $\mathsf{LicensePrice} <= 10$. As this constraint can be expressed by the grammar of RC, we add $\mathsf{GPL} \Rightarrow \mathsf{LicensePrice} <= 10$ to RC (see Figure 3.3). In general, more details about the synthesis algorithm can be found in references [49, 50].

Evaluation

We developed a tool that implements the synthesis algorithm. The tool is mainly implemented in Scala programming language with appropriate data structures (*e.g.*, HashMap and HashSet) for efficient computation of implications. For the computation of or-groups, we rely on the SAT4J solver [197].

Evaluation on random matrices. To provide an insight into the scalability of our procedure, we experimentally evaluate the runtime complexity of our AFD synthesis procedure. For this purpose, we have developed a random matrix generator, which takes as input three parameters:

- number of variables (features and attributes)
- number of configurations
- maximum domain size (*i.e.*, maximum number of distinct values in a column)

We first perform some initial experiments on random matrices. We quickly notice that computation of the or-groups poses a scalability problem. It is not surprising since this part of the synthesis algorithm is NP-hard, leading to some timeouts even for Boolean feature models (*e.g.*, see [285]). Our experiments confirm that the computation of or-groups quickly becomes time-consuming. The 30 minutes timeout is reached with matrices containing only 30 variables. With at least 60 variables, the timeout is always reached. Therefore, we deactivated the computation of or-groups in the following experiments.

Scalability results. Our main observations are as follows: The square root of the time grows linearly with the number of variables. The time grows linearly with the number of configurations. The synthesis time grows quadratically with the maximum domain size. Besides, results show that the major part of the algorithm is spent on the computation of binary implications and relational constraints for *RC*. The rest of the synthesis represents less than 10% of the total duration. According to our *theoretical* analysis (see [49] for details), the two hard parts of the synthesis algorithm are the computation of mutex-groups (exponential complexity) and or-groups (NP-complete). We note that 93.8% of the configuration matrices in our dataset produce mutex graphs that contain absolutely no edges. In such cases, computing mutex-groups is trivial. The rest of the algorithm has a polynomial complexity, which is confirmed by the experiments.

Evaluation on real-world matrices. To provide an insight into the scalability of our approach on realistic configuration matrices, we executed our algorithm on configuration matrices extracted from *Best Buy* is a well known American company that sells consumer electronics. On their website, the description of each product is completed with a matrix that describes technical information. Each matrix is formed of two columns in order to associate a feature or an attribute of a product to its value. The website offers a way to compare products that consists in merging the matrices to form a single configuration matrix which is similar to the one in Figure 3.2.

Experimental Settings We developed an automated technique to extract configuration matrices from *Best Buy* website. The procedure is composed of 3 steps. First, it selects a set of products whose matrices have at least 75% of feature and attributes in common. Then, it merges the corresponding matrices of the selected product to obtain a configuration matrix. Unfortunately, the resulting configuration matrix may contain empty cells. Such cells have no clear

Table 3.1: Statistics on the *Best Buy* dataset.

	Min	Median	Mean	Max
Variables	23	50.0	48.6	91
Configurations	11	27.0	47.1	203
Max domain size	11	27.0	47.1	203
Empty cells before interpretation	2.5%	16.1%	14.4%	25.0%

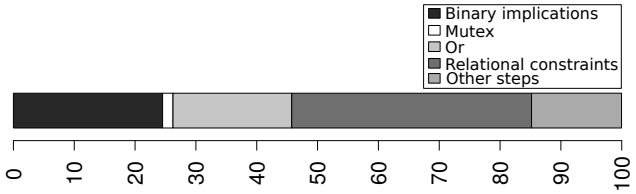


Figure 3.7: Time complexity distribution of the synthesis algorithm on the *Best Buy* dataset

semantics from a variability point of view. The last step of the procedure consists in giving an interpretation to these cells. If a feature or an attribute contain only integers, the empty cells are interpreted as "0". Otherwise, the empty cells are interpreted as "No" which means that the feature or attribute is absent.

With this procedure, we extracted 242 configuration matrices from the website that forms our dataset for the experiment. Table 3.1 reports statistics on the dataset about the number of variables, configurations, the maximum domain size and the number of empty cells before interpretation.

In the following experiments, we measure the execution time of the algorithm on the *Best Buy* dataset. We execute the algorithm on the same cluster of computers in order to have comparable results with previous experiments on random matrices. We also execute the synthesis 100 times for each configuration matrix of the dataset in order to reduce fluctuations caused by other programs running on the cluster and the random decisions taken during the synthesis.

Scalability results. We measure the execution time of the synthesis algorithm with the computation of or-groups activated. On the *Best Buy* dataset, the execution time is 0.8s in average with a median of 0.5s. The most challenging configuration matrix has 73 variables, 203 configurations and a maximum domain size of 203. The synthesis of an AFM from this matrix takes at most 274.7s. Figure 3.7 reports the average distribution for the dataset. It shows that the computation of binary implications, or-groups and the relational constraints are the most time-consuming tasks. It confirms the results of the experiments on random matrices. However, we note that on the *Best Buy* dataset, the computation of or-groups can be executed in a reasonable time. Our results also indicate that the execution time on the most challenging matrix of the *Best Buy* dataset has the same order of magnitude as the execution time on a similar random matrix.

replication

The complete source code of the synthesis algorithm can be found in https://github.com/gbecan/FOReverSE-AFMSynthesis. The repository https://github.com/gbecan/FOReverSE-AFMSynthesis-Evaluation gives details about the evaluation (data, R scripts, *etc.*).

3.2 Mining variability out of textual descriptions

The content of this section is adapted from the following publications:
S. Ben Nasr, G. Bécan, M. Acher, J. B. Ferreira Filho, N. Sannier, B. Baudry and J.-M. Davril, 'Automated Extraction of Product Comparison Matrices From Informal Product Descriptions', *Journal of Systems and Software (JSS)*, vol. 124, pp. 82–103, 2017. DOI: 10.1016/j.jss.2016.11.018. https://hal.inria.fr/hal-01427218
J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang and P. Heymans, 'Feature model extraction from large collections of informal product descriptions', in *ESEC/FSE*, 2013

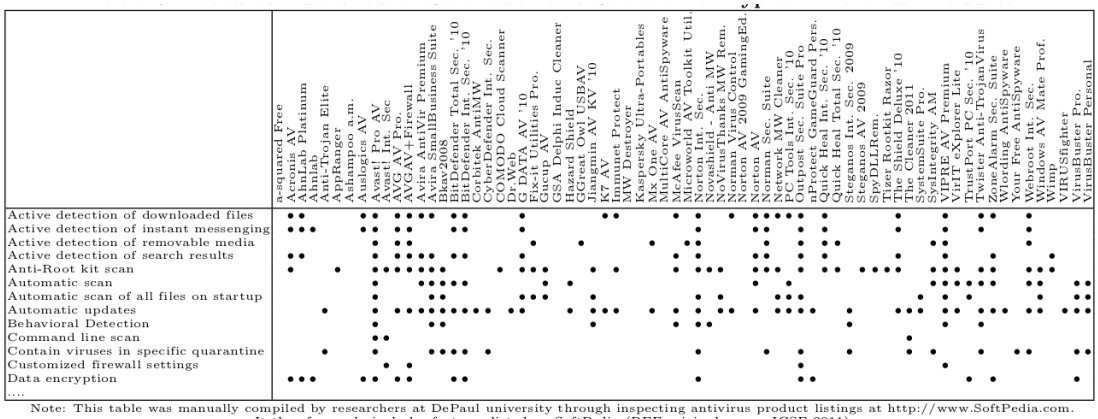
Domain analysis is a crucial activity that aims to identify and organize features that are common or vary within a domain [251, 33, 259]. At their respective level, domain experts, product managers, or even customers on their daily life activities need to capture and understand the important features and differences among a set of related products [166]. For instance, the motivation for a customer is to choose the product that will exhibit adequate characteristics and support features of interest; when several product candidates are identified, she or he will compare and eventually select the "best" product. In an organization, the identification of important features may help to determine business competitive advantage of some products as they hold specific features.

Manually analyzing and modelling a set of related products is notoriously hard [75, 103, 259, 138, 73]. The information is scattered all along textual descriptions, written in informal natural language, and represents a significant amount of data to collect, review, compare and formalize. A case-by-case review of each product description is labour-intensive, time-consuming, and quickly becomes impractical as the number of considered products grows.

Given a set of textual product descriptions, we propose an approach to automatically synthesize *product comparison matrices (PCMs)*. Our approach extracts and organizes information despite the lack of consistent and systematic structure for product descriptions and the absence of constraints in the writing of these descriptions, expressed in natural language.

With the extraction of PCMs, organizations or individuals can obtain a synthetic, structured, and reusable model for the understanding and the comparison of products. Instead of reading and confronting the information product by product, PCMs offer a *product line view* to practitioners. It is then immediate to identify recurrent features of a domain, to understand the specific characteristics of a given product, or to locate the features supported and unsupported by some products.

Techniques have been developed to mine variability [32, 176, 226] and support domain analysis [103, 259, 138, 73, 74, 81, 232, 30, 39, 165, 260, 123, 223], but none of them address the problem of structuring the information in a PCM. Many papers including our own work aim to elaborate a feature model as an outcome of the domain analysis. It is not our goal here: the outcome of our proposal is the synthesis of a PCM. As elaborated in Section 2.2, PCMs have a value per se, can be used with specialized tools or for deriving other services (*e.g.*, configurators and comparators). Furthermore, the synthesis of a feature model out of PCM can be envisioned if need be (as in [103] or as in Section 3.1). In fact, it is worth noticing that many works that synthesize feature models actually assume the presence of tabular data and PCM. An example is given in Figure 3.8, extracted from [103]. This "product–feature matrix", as named in [103], is what we call a product comparison matrix in this manuscript. It has been



Note: This table was manually compiled by researchers at DePaul university through inspecting antivirus product listings at http://www.SoftPedia.com. It therefore only includes features listed on SoftPedia (REF original source ICSE 2011)

Figure 3.8: Features published on SoftPedia for a selection of typical antivirus products. The table has been manually elaborated in [103]. Our goal in this contribution is precisely to reverse engineer a product comparison matrix out of informal textual descriptions.

used to build a feature model. The matrix has been obtained through manual inspection of informal descriptions and feature lists of dozens products. It is time-consuming, error-prone, and hard to replicate when information is updated. Our goal in this contribution is precisely to reverse engineer a product comparison matrix out of informal textual descriptions.

Motivation

Organizations describe the products they sell on their website using different categories of text forms. It goes from plain text in a single paragraph, formatted text with bullets, to matrices with product specifications. There is a spectrum of product descriptions ranging from structured data (matrices) to informal descriptions written in natural languages. Both have strengths, weaknesses, and have the potential to comprehensively describe a set of products. For instance, BestBuy provides descriptions for hundreds of thousands of products, including: (1) *product overviews*, text describing features of products using natural language (see Figure 3.9); (2) *technical specifications*, which describe the technical characteristics of products through feature lists.

Figure 3.9 illustrates the common scenario in which a customer needs to buy a laptop on BestBuy website and has to decide among a diversity of products. He/she has to go through many textual descriptions (product overviews) and reason over the different features of the product. A typical question is to figure out if a particular feature is supported by existing products (if any) and what are the alternatives. In domain analysis, the biggest challenge is related to the number of products and the number of features an analyst has to gather and organize. The more assets and products, the harder the analysis. Our goal is to automate the manual task of analyzing each product with respect to its textual description and clustering information over several products, and provide a reader with an accurate and synthetic *product comparison matrix (PCM)*, as shown in Figure 3.9.

Synthesis

Features	HP - ENVY	Lenovo - Yoga	Apple® - MacBook	Dell - Inspiron
High-quality images	YES	YES	YES	YES
Voice recognition software	NO	YES	NO	YES
Touch Screen	YES	YES	NO	YES
Media Reader	Digital	Multiformat	Digital	5-in-1
Webcam	Front-facing TrueVision high-definition	N/A	Built-in high-definition	Built-in high-definition
Hard Drive	1TB	500 GB	750 GB	1TB
Resolution	1920 x 1080	1366 x 768	2560 x 1600	1920 x 1080
Processor Speed	2.4 Ghz	1.5 Ghz	2.8 Ghz	3.0 Ghz
Memory	12GB DDR3L SDRAM	4GB DDR3L	16 GB	16GB DDR3L
Processor	5th Gen Intel Core i7-5500U	4th Gen Intel Core i3-4012Y	Fourth-generation quad-core Intel Core i7	5th Gen Intel Core i7-5500U
Operating System	Microsoft Windows 8.1 64-bit	Microsoft Windows 8.1 64-bit	Mac OS X 10.9 Mavericks	Microsoft Windows 8.1 64-bit

Figure 3.9: Automatic synthesis of a PCM from 4 textual product descriptions. Portions of texts in yellow have been identified and exploited to synthesize features' names and cell values. High-quality images and voice recognition (in red) are features only described in the text *i.e.,* these features are not described in the technical specifications (feature list) of the same 4 products.

Automated extraction

Automating the extraction comes with a set of challenges, mostly due to the informal and unstructured nature of textual overviews.

Our automated approach relies on Natural Language Processing (NLP) and mining techniques to extract PCMs from text. The proposed method takes the descriptions of the different products as input, and identifies the linguistic expressions in the documents that can be considered as terms. In this context, a term is defined as a conceptually independent expression. Then, the method automatically identifies which terms are actually domain-specific. We also rely on information extraction to detect *numerical information*, defined as domain relevant multi-word phrases containing numerical values. The task of building the PCM involves computing terms (resp., information) similarity, terms (resp., information) clustering, and finally features and cell values extraction.

The approach has been implemented in a tool, *MatrixMiner*: It is a web environment with an interactive support for automatically synthesizing PCMs from informal product descriptions [53]. *MatrixMiner* also maintains traceability with the original descriptions and the technical specifications for further refinement or maintenance by users.

MatrixMiner targets domain analysts, software practitioners, customers, or organisations that want to build and maintain PCMs. Afterwards users can, from PCMs, (1) generate other domain models, such as feature models [32, 103, 205]; (2) recommend features [138] (3) perform automatic reasoning (*e.g.*, [236, 223]); (4) devise configurators or comparators; or (5) simply understand a set of related products.

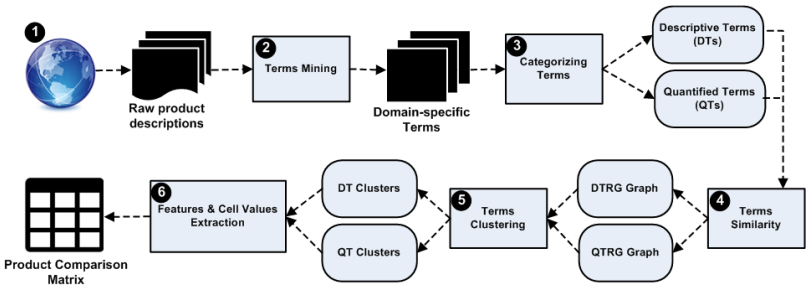


Figure 3.10: Extraction process; the resulting PCM can be visualized/edited in a Web environment (see Figure 3.11)

The extraction process of *MatrixMiner* is summarized in Figure 3.10 and consists of two primary phases. In the first phase, domain specific terms are extracted from a set of informal product descriptions (steps ❶ and ❷), while in the second phase the PCM is constructed (steps ❸ to ❻). For step ❶, the raw product descriptions are extracted. Currently, we have implemented a mining procedure on top of BestBuy API [156] for retrieving numerous product pages along different categories. We also provide means to either (1) manually select the products to be included in the comparison; or (2) group together closest product within a category. The gathering of data can be generalized to other sources of Web information (beyond BestBuy). We outline in the following the rest of the procedure.

Terms Mining. Step ❷ is based on a novel natural language processing approach, named *contrastive analysis* [67], for the extraction of domain specific terms from natural language documents. In this context, a term is a conceptually independent linguistic unit, which can be composed by a single word or by multiple words. A multi-word is conceptually independent if it occurs in different context (*i.e.*, it is normally accompanied with different words). For instance, "Multiformat Media Reader" is a term, while "Reader" is not a term, since in the textual product descriptions considered in our study it often appears coupled with the same word (*i.e.*, "Media"). Combining single and compound words is essential to detect features and their values.

Our multi–word term extraction method is based on a combination of "termhood" measures, assessing the likelihood of being a valid technical term, and contrastive methods. In particular, multi–word term extraction is carried out by identifying multi–word terms candidates in an automatically Part–Of–Speech (POS) tagged and lemmatized text, making use of different kinds of linguistic features. POS-tagging aims at representing a text as an abstract syntax tree with textual tokens annotated a nouns, verbs, noun phrases, *etc.* These candidates are then weighted with the C–NC value, currently considered as the state–of–the–art method for terminology extraction [67]. This metric establishes how much a multi-word is likely to be conceptually independent from the context in which it appears. The ranking of identified multi–words terms is then revised on the basis of a contrastive score calculated for the same domain-specific terms.

Building the PCM. Once the top list of terms is identified for each product, we start the construction of the PCM. This process requires creating some intermediate structures. In step ❸ we divide the set of terms in two categories: quantified terms containing measures (*e.g.*, "1920 x 1080 Resolution") including intervals (*e.g.*, "Turbo Boost up to 3.1 GHz"); and descriptive terms containing noun phrases and adjectival phrases (*e.g.*, "Multiformat Media Reader"). The key idea is to perform separately descriptive terms (DTs) clustering from quantified terms (QTs) clustering. A DTs cluster gives the possible descriptor values (*e.g.*, "Multiformat") while a QTs cluster provides the potential quantifier values (*e.g.*, "1920 x 1080") for the retrieved feature. In step ❹ we compute terms similarity to generate a terms relationship graph for each category. The goal of this step is to determine a weighted similarity relationship graph among terms within the same category. To identify coherent clusters, we first determine the similarity of each pair of terms by using syntactical heuristics. In step ❺ we apply term clustering in each graph to identify descriptive and quantified term clusters. The underlying idea is that a cluster of tight-related terms with different granularities can be generated by changing the clustering threshold value [81]. Finally, step ❻ extracts features and cell values to build the PCM. To extract the feature name from a cluster, we developed a process that involves selecting the most frequently occurring phrase from among all of the terms in the cluster. This approach is similar to the method presented in [161] for summarizing customer reviews. For example, "1920 x 1080 Resolution" and "1366 x 768" Resolution" represent QTs clusters that gives "Resolution" as a features name and two potential values: "1920 x 1080" and "1366 x 768". Terms which are not clustered will be considered as boolean features. Finally we distinguish different types of features (see Figure 3.11): *boolean* which have Yes/No values, *quantified* when their values contain measures (*e.g.*, "Resolution", "Hard Drive", *etc.*), *descriptive* if their values contain only noun and adjectival phrases (*e.g.*, "Media Reader"), and *empty* values. The resulting PCM can be visualized and refined afterwards (see next section).

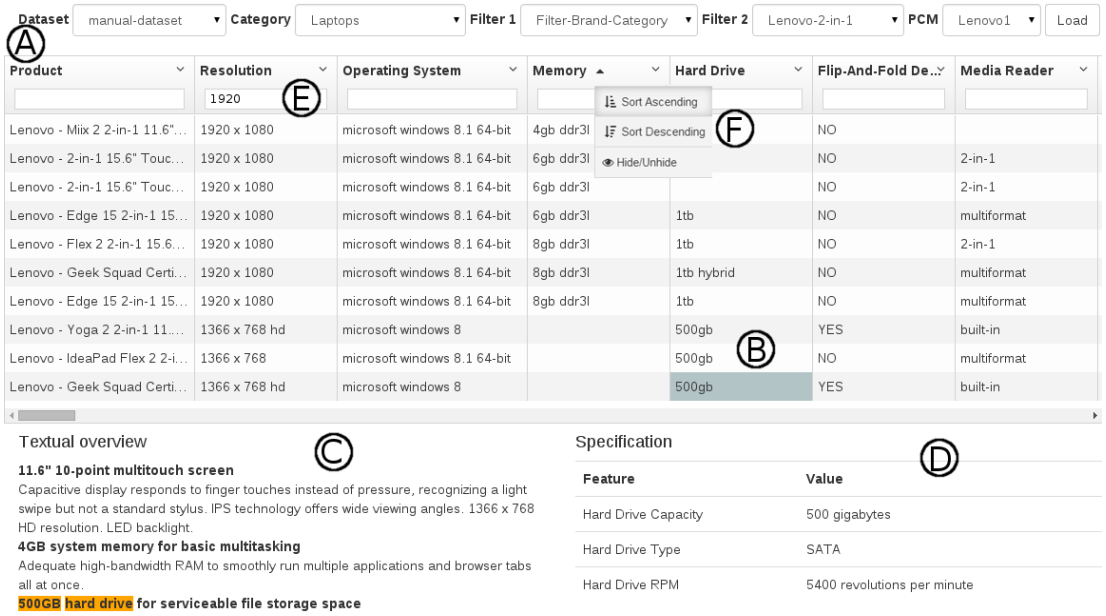


Figure 3.11: The editor of *MatrixMiner* in action

MatrixMiner offers an interactive mode where the user can import a set of product descriptions, synthesize a complete PCM, and exploit the result [53].

Our early empirical insights suggested that human intervention is beneficial to (1) refine/correct some values (2) re-organize the matrix for improving readability of the PCM. As a result we developed an environment for supporting users in these activities. *MatrixMiner* provides the capability for *tracing* products and features of the extracted PCM to the original product overviews and the technical specifications. Hence the PCM can be interactively controlled, complemented or refined by a user. Moreover users can restructure the matrix through the grouping or ordering of features. Overall, the functionalities available are the following:

- select a set of comparable products. Users can rely on a number of filters (*e.g.*, category, brand, sub categories, *etc.*. See Figure 3.11, Ⓐ);
- ways to visualize the PCM with a traceability with original product descriptions. For each cell value, the corresponding product description is depicted with the highlight of the feature name and value in the text. For instance, "500GB Hard Drive" is highlighted in the text when a user clicks on "500GB" (see Figure 3.11, Ⓑ and Ⓒ);
- ways to visualize the PCM with a traceability with the technical specification (see Figure 3.11, Ⓓ). For each cell value, the corresponding specification is displayed including the feature name, the feature value and even other related features. Regarding our running example, "Hard Drive Capacity" and two related features ("Hard Drive Type" and "Hard Drive RPM") are depicted together with their corresponding values;
- basic features of a PCM editor. Users can remove the insignificant features, complete missing values, refine incomplete values or revise suspect values if any – typically based on information contained in the textual description and the technical specification;

- advanced features of a PCM editor: means to filter and sort values (see Figure 3.11, Ⓔ and Ⓕ); ways to distinguish Yes, No and empty cells using different colors to improve the readability of the PCM; prioritise features by changing the columns order, *etc.*

Evaluation

Our evaluation is made of two major studies.

Empirical Study. We aim to measure some properties of the extracted PCMs. Is our extraction procedure able to synthesize comparable information and compact PCMs? Is there an overlap between synthesized PCMs and technical specifications?

User Study. We aim to evaluate the quality of the information in the synthesized PCMs. How correct are features' names and values in the synthesized PCMs? Can synthesized PCMs refine technical specifications? Such a study necessitates a human assessment. We have involved users to review information of our synthesized PCMs using *MatrixMiner* traceabilities.

We evaluate our tool against numerous categories of products mined from BestBuy [156], a popular American company that sells hundreds of consumer electronics. Specifically, we selected 9 products categories that cover a very large spectrum of domains (Printers, Cell phones, Digital SLR Cameras, Dishwashers, Laptops, Ranges, Refrigerators, TVs, Washing Machines) from BestBuy. Currently, we have implemented a mining procedure on top of BestBuy API [156] for retrieving numerous product pages along different categories. We mined 2692 raw product overviews using the BestBuy API. Another important property of the dataset is that product descriptions across and within different categories do not share the same template. The absence of template challenges extractive techniques. By design, our approach does not assume any regular structure of product descriptions.

Our empirical results show that the synthesized PCMs are compact and exhibit numerous quantitative, comparable information. Specifically:

- Our approach is capable of extracting numerous quantitative and comparable information (12.5% of quantified features and 15.6% of descriptive features).
- A supervised scoping of the input products reduces the complexity (in average 107.9 of features and 1079.7 of cells) and increases the homogeneity and the compactness of the synthesized PCMs (only 13% of empty cells).
- An open issue is that the size of PCMs can be important while PCMs, being from overviews or technical specifications, can be incomplete. It motivates the study of the complementarity of the two kinds of PCMs.
  - From the complementary perspective of product variability sources, a significant portion of features (49.7%) and cell values (26.2%) is recovered in the technical specifications.
  - The proportion of overlap of overview PCMs regarding specification PCMs is significantly greater than the overlap of the latter regarding overview matrices. This is explained by the fact that the natural language is richer, more refined and more descriptive compared to a list of technical specifications.
  - Overall, users can benefit from an interesting overlap. They can reduce the complexity of the PCMs by only focusing on overlapping features' names and values. They can also complete missing cell values or even refine some information of PCMs. It motivates the next "user study".

Our previous study does not evaluate the *quality* of the information in the synthesized PCMs. For example, we do not know how correct are features' names and values in the synthesized PCMs coming from informal and textual overviews.

User study. We considered the same set of supervised overview PCMs used earlier in the empirical study. These PCMs cover a very large spectrum of domains (Printers, Cell phones, Digital SLR Cameras, Dishwashers, Laptops, Ranges, Refrigerators, TVs, Washing Machines, *etc.*). These PCMs are made from various sizes, going from 47 to 214 columns (features), and 10 rows (products).

The PCMs were evaluated separately by 20 persons, each using their own computers. Participants were computer science researchers and engineers at Inria (France).

The evaluators have to *validate* features and cell values in the PCM against the information contained in the original text. Specifically, the evaluators had to specify for each column whether the PCM contains more/less refined information (features and cell values) than in the specification: We displayed one column at a time. The evaluators have to validate the feature and cell values. To this end, the tool provides ways to visualize the PCM with a traceability with original product descriptions. For each cell value, the corresponding product overview is depicted with the highlight of the feature name and the value in the text. Once the evaluation of one column is finished, the evaluator submits his/her evaluation and starts again a new evaluation for a new column. We obtained 118 evaluated features and 1203 evaluated cell values during an evaluation session of one hour. Overall, 50% of evaluated features belong to ranges, 24.57% come from laptops, 16.10% are related to printers, and 9.32% correspond to features of refrigerators, TV and washing machines. On the other hand, 45.95% of evaluated cell values are about ranges, 22.61% are contained in laptops PCMs, 16.90% of values belong to printers and 14.52% are related to refrigerators, TV and washing machines.

The user study shows that we can retrieve a significant portion of correct information. Specifically:

- Our automatic approach retrieves 43% of correct features and 68% of correct cell values. Users can rely on MatrixMiner's traceability to control, edit or remove some features' names and values without having to review the entire textual descriptions.

- Results show that we have as much or more information in the synthesized PCMs than in the technical specifications for a significant portion of features (56%) and cell values (71%). Again, users can rely on MatrixMiner to refine or expand the information in both sources.

Overall, we provide empirical evidence that there is a potential to complement or even refine technical information of products thanks to our extraction. The evaluation insights drive the design of the *MatrixMiner*, a web environment with an interactive support for synthesizing, visualising and editing PCMs. The presented work has the potential to crawl scattered and informal product descriptions that abound on the web. Other inputs such as online reviews of products can be considered as well. The identification of relationships between features (*e.g.*, conflict) is also an interesting perspective. Although http://matrix-miner.variability.io is no longer available at the time of writing, there is a demonstration of *MatrixMiner*: https://www.youtube.com/watch?v=ezKx-S0LiNQ

replication

The specific source code of the extraction procedure is available online: https://github.com/sbennasr/matrix-miner-engine. Our Web environment reuses the editor of OpenCompare https://github.com/gbecan/OpenCompare

.

3.3 Reverse engineering Web configurators

The content of this section is adapted from the following publications:
E. Khalil Abbasi, M. Acher, P. Heymans and A. Cleve, 'Reverse Engineering Web Configurators', in *17th European Conference on Software Maintenance and Reengineering (CSMR'14)*, IEEE, Ed., Antwerp, Belgium, Feb. 2014
E. Khalil Abbasi, A. Hubaux, M. Acher, Q. Boucher and P. Heymans, 'The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases', Anglais, in *25th International Conference on Advanced Information Systems Engineering (CAiSE'13)*, M. Norrie and C. Salinesi, Eds., Valencia, Espagne, Jun. 2013. http://hal.inria.fr/hal-00796555

A *Web Configurator* is an online product configuration environment for choosing or customizing products that match individual needs. Customers gradually select the configuration options to be included in the final product. A configurator provides an interactive graphical user interface (GUI) that guides the users throughout the configuration process (see Fig. 3.12 for an example). Web configurators are complex systems [179, 264, 306]: numerous kinds of constraints govern the options, the configuration process can be multi-step and non linear, and advanced capabilities are provided to check consistency, automatically complete undecided options, etc.

Our previous empirical study of 111 Web configurators [179] revealed the absence of specific, adapted, and rigorous methods in their engineering. Some of the Web configurators are developed like any other typical Web applications: proceeding this way leads to reliability, runtime efficiency, and maintainability issues. Specifically, we identified a large number of bad practices (incomplete reasoning, counter-intuitive representation of options, losing of all decisions when navigating backward, etc.) in the 111 configurators. Some of our industrial partners face similar problems and are now trying to migrate their legacy configurators to more reliable, efficient, and maintainable solutions [70].

To decrease the cost of migration, we propose to systematically *re-engineer* these applications. This encompasses two main activities: *(1)* reverse engineering a legacy configurator and encoding the extracted data into dedicated formalisms, and then *(2)* forward engineering new improved, customized configurator based on models [70]. The use of variability models to formally capture configuration options and constraints, and state-of-the-art solvers (*e.g.*, SAT, CSP, or SMT) to reason about these models, would provide more effective bases (see Chapter Modelling Software Variability).

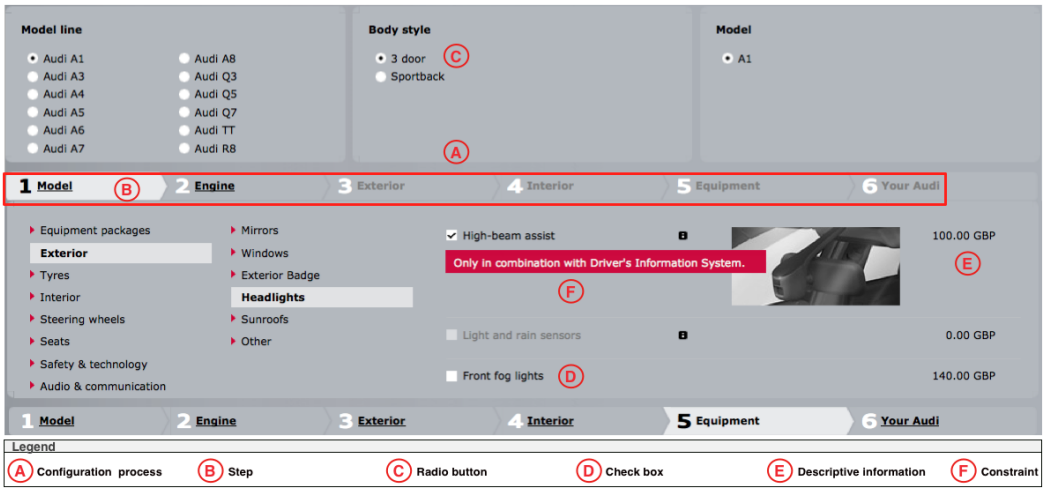


Figure 3.12: Audi Web configurator (`http://configurator.audi.co.uk/`)

In this contribution, we focus on the *reverse-engineering* process. It consists of extracting configuration-specific data such as options, their associated descriptive information, and constraints, altogether called *variability data*, from the Web pages of the configurator, and then constructing a variability model, for instance, a *feature model*. Building a complete feature model requires, ideally, analysing both the client and the server sides of a configurator. We investigate here the visible parts of configurators, *i.e.*, the GUI of the *Web client* because it is the entry point for customer orders and most of the variability data is somehow represented in Web pages.

The major difficulty is that Web configurators, despite having a common goal and similar features, vary significantly: variation in implementation and presentation of configuration-specific objects as well as the way constraints govern the selection of options. For example, some options are all located in the same Web page; in other configurators, some options only appear in a new Web page once a certain selection has been performed. To the best of our knowledge, the problem of extracting feature models from Web configurators has not been studied. Existing techniques for reverse engineering feature models assume a formal representation of the constraints (*e.g.*, through a formula [47, 49, 284, 32]) or exploit specific artefacts (*e.g.*, product descriptions (see Section Mining variability out of textual descriptions), dependency files (see Section Reverse engineering architectural variability models), source code [284, 330]). Methods for reverse engineering Web GUIs (*e.g.*, see [218]) do not propose *dedicated* techniques for (1) locating options in a Web page or for (2) analyzing the dynamics and the specificity of a configuration process.

Method

This contribution presents a novel tool-supported and supervised approach to reverse engineer Web configurators. The reverse-engineering tool consists mainly of two collaborative components: *Web Wrapper* and *Web Crawler*. A Web Wrapper extracts variability data from a Web page and transforms it into structured data in a semi-automatic way. A Web Crawler focuses on the runtime behaviour of configurators. It explores the *configuration space* (*i.e.*, all

objects representing configuration-specific data) and simulates (some of) users' configuration actions. The Crawler systematically generates dynamic variability data which is then extracted by the Wrapper. The Wrapper and the Crawler operate over the notion of *variability data extraction pattern* (*vde* pattern in short).

Due to the high diversity of presentations and implementations found in Web configurators [179], a fully automated approach is neither realistic nor desirable. We consider that the data extraction process should be supervised. A user manually marks and names data to be extracted by giving it a meaningful label in a *vde* pattern specification. Consequently, (1) the user distinguishes configuration-specific data from the other irrelevant and noisy data, (2) she explicitly and accurately organizes data items in the extracted data records by assigning them different labels, and (3) representing the extracted data in a predefined data model becomes feasible, because the types and logical relationships of data to be extracted from Web pages of a configurator are rather known. Users can specify a *vde* pattern, expressed in an HTML-like language, to extract the variability data from Web pages. The Web Wrapper, given a *vde* pattern (*i.e.*, the specification of the structure of objects of interest), locates in a Web page code fragments (implementing objects of interest) that structurally conform to that pattern and then extracts their data .

Fig. 3.13 depicts our proposed *supervised* and *semi-automatic* reverse-engineering process. Interactive (I) and automatic (A) activities are distinguished. The process starts with the specification of *vde* patterns for a given Web page (❶). The user inspects the source code of the page, identifies templates from which the page is generated, specifies the appropriate *vde* pattern defining the structure of those templates, and marks the required data in the pattern. The specified *vde* pattern is given to the Web Wrapper. The Web Wrapper is a program that takes as input specification of a *vde* pattern and a Web page, seeks and finds code fragments in the page that *structurally* match the given pattern, and extracts as output data items from those code fragments corresponding to the marked data in the pattern (❷). The extracted data is hierarchically organized and serialized using an XML format. Most likely, the analysed Web page does not contain all configuration-specific data objects. New configuration content may be added to the page based on some selections of options. The Crawler simulates some of users' configuration actions in order to automatically generate dynamic content (❸). The newly added data is extracted by the Wrapper (❷). The content extracted in steps ❷ and ❸ can be edited (❹). The clean XML file is then given to a module which transforms it into a feature model (❺). We rely on the *Text-based Variability Language* (*TVL*) to represent feature models [85]. At the end of the reverse-engineering process there are typically several generated TVL models (*e.g.*, each corresponding to a specific configuration step). To produce a fully-fledged TVL model, all these models are fed to FAMILIAR (❻) that provides operators to merge incomplete feature models into a single feature model (see Section Automated feature model management).

Variability Data Extraction Pattern. Client-side source code is usually developed or generated from a number of Web templates. Web Configurators are no exceptions. Each Web page consists of a number of *template instances* which are syntactically identical fragments except for variations in values for data slots (text elements and tag attribute values) as well as minor changes to their structures. We take advantage of templates used in Web pages to extract the required data.

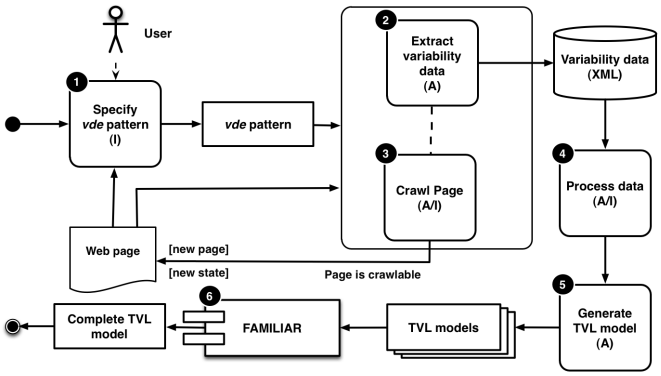


Figure 3.13: The reverse engineering process

All *vde* patterns required to extract data are specified in a *configuration file*. A configuration file contains the specification of at least one *data* pattern and one *region* pattern. A data pattern marks text elements and attributes carrying the content of interest, denoting code fragments (*i.e.*, template instances) that match certain properties and thus contain the relevant data. A region pattern highlights a portion of the source code where the Wrapper will operate. The syntax of vde patterns as well as the pattern matching algorithm can be found in [178]. Previous extensive empirical observations on 100+ Web configurations [179] drive the design of this specific pattern language and algorithm.

Crawling the configuration space. The configuration space may be distributed over multiple pages each having a unique URL (*multi-page* user interface paradigm) or all the configuration-specific objects are contained in a page (*single-page* user interface paradigm). For configurators following the single-page paradigm, one common scenario is that when a Web page is loaded, the configuration space contains some configuration-specific objects and as the application is executing, new objects may be added to the page, and existing objects may be removed or changed. Configuring an option and exploring configuration steps are common actions to change the content of the page. By configuring an option its consistent options are loaded in the page. For instance, the selection of an option in the "Model line" group in Fig. 3.12 loads new options to the "Body style" group. Configuring an option may also change the configuration state of other impacted options. For instance, the selection of "High-beam assist" makes unavailable "Light and rain sensors". Note that both cases indicate that there are underlying constraints between those options, consequently, these constraints should be extracted as well. Activation of a step makes available its options in the page and makes unavailable those of other steps.

To extract dynamic data, we need to automatically crawl the configuration space in a Web page. Automatically crawling requires (1) the simulation of users' configuration and exploration actions to systematically generate new content or alter the existing content and then (2) the analysis of the changes made to the page to deduce and extract configuration-specific data. The Web Crawler and the Wrapper collaborate together to deal with these cases.

At present, the Crawler is able to simulate some users' actions, for instance, the selection of items from a list box and the click on elements (*e.g.*, button, radio button, menu, image, etc.). The simulation of user actions may change the content of the page, therefore, after simulating every clickable element, the page's content must be analyzed to identify the newly added content and to deduce from that the configuration-specific data. We observed that when a configuration action is performed by the user, a few identifiable regions on the page are impacted and their content may be changed. Consequently, rather than analyzing the whole page, only those regions should be investigated. Based on this observation, we divide the configuration-specific regions of a page into two groups: *independent* and *dependent* regions. When a configuration action is performed on a configuration-specific object in an independent region, new objects are added to the dependent regions or existing ones are changed.

It may happen that no new option is added to the page once a selection is performed. Instead, the configuration states of existing objects are changed. For instance, in the "Equipment" step in Fig. 3.12, when an option is given a new value, the configurator automatically propagates the required changes to all the impacted options. In this case, crawling is a way to instantiate and then extract such constraints. Technically, when the Crawler configures an option, the Wrapper extracts all the contained options and their states. Therefore, at the end of the process all the visited configuration states of all the options are documented in the output XML file. These state changes are then analyzed to identify which constraints logically impact (*e.g.*, through exclusions or implications) other options.

Tool Support. We developed a *Firebug*[1] extension that consists of the Wrapper and the Crawler components. The extension generates an XML file which presents the output data. The generated XML file is then given to a Java application which converts it to the corresponding TVL model.

Evaluation

Goal and scope. Our approach aims to reverse engineer feature models from Web configurators. We want to (1) evaluate the ability of the approach to deal with variations in presentation and implementation of variability data, (2) assess the *accuracy* of the extracted data, and (3) measure the users' *manual effort* required to perform the extraction.

Questions and metrics. We address four questions (Q):

- **Q1.** How accurate is the extracted data?
- **Q2.** How expressive is the pattern language?
- **Q3.** How applicable is the crawling technique?
- **Q4.** How much manual effort is needed to perform the reverse-engineering process?

Data set. We considered five configurators: S1 is the Dell's laptop configurator. We took the "Inspiration 15" model in this experiment. S2 is the car configurator of BMW. For this study, we chose the "2013 128i Coupe" model. S3 is a dog-tag generator, in S4 the customer can choose her chocolate and create its masterpiece and ingredients, and S5 is a configurator that allows customers to design their shirts.

For each question, we compute specific metrics (*e.g.*, number of patterns required to extract data for Q2).

[1] http://getfirebug.com/

Execution. The first author of the paper E. Khalil Abbasi, M. Acher, P. Heymans and A. Cleve, 'Reverse Engineering Web Configurators', in *17th European Conference on Software Maintenance and Reengineering (CSMR'14)*, IEEE, Ed., Antwerp, Belgium, Feb. 2014 supervised the extraction process. For each Web page, we first inspected its source code to find out which templates are used and then specified the required patterns to extract data.

Accuracy of the extracted data (Q1). For the cases on which we applied the proposed approach, the accuracy of the extracted data is promising. Hundreds of configuration options, their attached descriptive information, and constraints defined over these options are automatically extracted and hierarchically organized. 99% of the extracted options and 99.4% of the extracted constraints are precise data.[2]

Expressiveness of the pattern language (Q2). We could specify patterns to cover all code fragments that implement configuration-specific objects. Pattern-specific elements and operators we designed in the language gave us a lot of support in specification of patterns for templates we identified in this experiment. We specifically observed that there are frequent patterns that are shared across configurators. We also found the notion of multiplicity of an element very practical in this experiment. For instance, the items of list boxes in configurator S5 and the list of attached sub-options in configurator S2 are examples of multi-instantiated elements that we could model them in the patterns.

Applicability of the crawling technique (Q3). Using the crawling technique, we could study the dynamic nature of the configuration process. We gain numerous additional configuration options and constraints with the crawling technique. 18% of the automatically extracted options and 16.3% of the constraints are identified and extracted using the crawling technique. Moreover, dependency between patterns allowed us to document the parent-child relationships between options. All this data are collected by specifying eight dependencies. Nevertheless, we cannot claim that the crawling technique can detect and extract *all* objects that may be dynamically generated at runtime. We neither have base models to which we could compare our generated models nor have access to the developers of the studied configurator who can validate our models. It is worth to mention other experiences in reverse engineering contexts (*e.g.*, see [102, 284] or in this manuscript Section Synthesizing attributed feature models out of tabular data, Reverse engineering architectural variability models Mining variability out of textual descriptions) showing that incomplete feature models may be obtained, thus calling for the intervention of the user or any kind of knowledge/artefact to further refine the model [145].

The manual effort required to perform the extraction process (Q4). In this experiment, overall we specified 13 region and 19 data patterns, wrote 322 lines of code for these patterns, and executed them 38 times to extract all data. 3.7% of the collected options and 8.5% of the constraints are manually added to the automatically extracted data. The manual writing of 322 lines of code to specify the required patterns in this experiment led to generating TVL models with 4478 lines of code. We believe that our semi-automatic and supervised approach provides a realistic mix of manual and automated work. It acts as an interesting starting point for re-engineering a configurator while mining the same amount of information manually is clearly daunting and error-prone.

[2] A limitation of our study is that we did not report on recall. However, metrics about the number of options, groups and constraints suggests that a significant amount of variability data has been recovered.

Summary. Experimental results show that the proposed language is expressive such that using a few patterns the user can extract hundreds of options presented in a page. They also confirm the ability of the Crawler to dynamically and automatically mine numerous additional configuration options and constraints.

replication

The specification of the pattern language, tools, and the complete set of data are available at `http://info.fundp.ac.be/~eab/result.html`

3.4 Reverse engineering architectural variability models

The content of this section is adapted from the following publications:
M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien and P. Lahire, 'Reverse Engineering Architectural Feature Models', in *5th European Conference on Software Architecture (ECSA'11), long paper*, ser. LNCS, Essen (Germany): Springer, Sep. 2011, p. 16
M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien and P. Lahire, 'Extraction and Evolution of Architectural Variability Models in Plugin-based Systems', *Software and Systems Modeling (SoSyM)*, 2013

Large software systems are now commonly organized around a more or less explicit architecture, which defines entities, their properties and relationships. When software product line (SPL) engineering principles are followed from the start, it is feasible to manage variability through one or more *architectural* feature models and then associate them to the system architecture [243]. The major architectural variations are then mapped to given features, allowing for automated composition of architectural elements when features are selected to configure a particular software product from the line. A resulting property of crucial importance is to guarantee that the variability is not only preserved but also kept consistent across all artefacts [98, 34, 204].

In many cases, however, one has to deal with (*legacy*) software systems not initially designed as SPLs [284, 11, 141, 321, 257, 322]. When the system becomes more complex, with many configuration and extension points, its variability must be handled according to SPL techniques. In this context, the task of building an architectural feature model is very arduous for software architects. They typically have to deal with lots of plugins (usual customizations of the Eclipse IDE are made with several hundreds of plugins, corresponding to dozens of high-level features [130, 250]), for which *safe composition* is the topmost requirement [204].

It is then necessary to recover a consistent feature model from the actual architecture. On a large scale both automatic extraction from existing parts and the architect knowledge should ideally be combined to achieve this goal. In particular, a software architect should be able to determine whether her (high-level) representation complies with an automatically extracted model, and to what extent they differ from each other (*e.g.*, in the style of reflexion models [224]). Moreover, since the software architecture and functionalities are naturally evolving over time, it is also necessary to ensure that an architectural feature model is maintained consistent with these changes. In the case of modern dynamic software architectures, which are based on plugins, these modifications can be very complex to handle, especially

in the presence of hidden dependencies between (different versions of) plugins. In this context, evolving the architectural feature model along the modified architecture is tedious. It is therefore needed to reproduce the extraction process and to reason on the new architectural feature model and on its differences.

The FraSCAti Plugin-based System Case Study

We motivate and illustrate our proposal on a case study related to the FraSCAti platform [217], an open source implementation of the OASIS's Service Component Architecture (SCA) standard [233]. SCA is a technology-agnostic component-based standard for building distributed composite service-oriented applications mixing various programming languages and frameworks (*e.g.*, Java, C, C++, WS-BPEL, Spring Framework) for implementing business components, various interface definition languages (*e.g.*, WSDL, Java) for describing business services, and various network communication protocols (*e.g.*, Web Service, Java Messaging Service) for interconnecting distributed applications.

Main SCA component-based concepts are quite generic and present in numerous other component models: a *composite* is a component composed of a set of components, a *component* encapsulates a business logic implemented with a programming language/framework, a *service* and a *reference* are named interfaces respectively provided/required by a component, an *interface* is a set of methods implemented or used by a component, a *binding* explains how both service and reference are accessible via a network communication protocol, and a *wire* connects a source reference to a target service.

Started in 2007, the development of FraSCAti began with a framework based on a basic implementation of the standard, that has then been incrementally enhanced. After six major releases, it now supports several SCA specifications and provides a set of extensions to the standard, including component implementation types binding implementation types, interface description types, and runtime APIs for component introspection and reconfiguration [281, 282].

As its capabilities grew between releases, FraSCAti has itself been refactored and completely architected as an SCA-based application, *i.e.*, an assembly of SCA components. The FraSCAti architecture is composed of three main SCA composites:

- The SCA parser is responsible to load business SCA composite files into memory. As the SCA composite language is extensible, its grammar is described by several meta-models (MM). Then FraSCAti supports various SCA meta-models (*e.g.*, *MMFrascati*, *MMTuscany*).
- The Assembly Factory is responsible to check SCA composites and orchestrate their instantiation. The assembly factory is composed of several plugins for dealing with the various forms of component implementations, interface definition languages, and service bindings (*e.g.*, *rest*, *http*).
- The Component Factory is in charge of instantiating SCA components. This factory generates and compiles Java code for component containers. This factory has two plugins for supported Java compilers (*i.e.*, *JDK*6 and *JDT*).

Thanks to its new component-based architecture, different variants of FraSCAti can be built in order to meet various application requirements and target system constraints. Each SCA application running on FraSCAti could have different requirements in terms of SOA features like supporting SOAP, WSDL, WS-BPEL, REST, OSGi, JMS. All these SOA features

are implemented as SCA components which are plugged to the FraSCAti architecture. Then, application developers could select all the FraSCAti plugins required for their applications. Orthogonally, the target system on which applications are deployed could impose some constraints.

For instance, in case a FraSCAti variant should support the compilation of Java code on the fly, then an embedded Java compiler is required. FraSCAti supports two distinct Java compilers: The standard JDK6 compiler and the Eclipse JDT compiler. FraSCAti plugins could have dependencies, *e.g.*, the REST binding plugin requires the FraSCAti meta-model while the HTTP binding plugin requires the Tuscany meta-model. These FraSCAti plugin dependencies are captured via Apache Maven[3] XML-based descriptors.

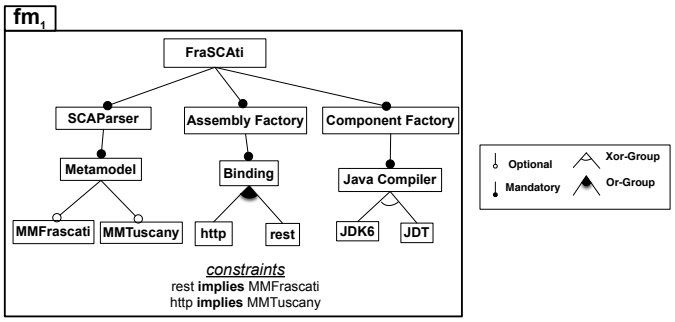


Figure 3.14: Architectural feature model (simplified from our case study)

FraSCAti version 1.5 contains around 60 plugins for a total of around 250.000 lines of code. So, FraSCAti is representative of a large plugin-based system, *i.e.*, a system composed of plugins, each of which is implemented as a set of SCA components that adds specific abilities to FraSCAti.

With all these capabilities, the FraSCAti platform has become highly (re-)configurable in many parts of its own architecture. It exposes a larger number of extensions that can be activated throughout the platform, creating numerous variants of a FraSCAti deployment. It then became obvious to FraSCAti technical leaders that the variability of the platform should be more systematically managed in order to better drive and control its evolution. A possible feature model of FraSCAti is depicted in Figure 3.14 – the reader can recognize features that have been introduced in the text (*e.g.*, *MMFrascati*, *MMTuscany*).

Reverse engineering process along evolution

We present a comprehensive, tool supported process for reverse engineering and evolving architectural feature models. Specifically, we develop automated techniques to extract and combine different variability descriptions of a software architecture, integrating the hierarchical decomposition of the architecture and inter-plugin dependencies. The basic idea is that variability and technical constraints of the plugin dependencies are *projected* onto an architectural model. After the extraction, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted FM. In addition, we show how the extraction process can be reiterated when the architecture evolves. This notably enables the architect to re-integrate his knowledge and to reason about the differences between two successive architectural feature models.

[3]Maven (http://maven.apache.org/) is a software tool for managing a project's build, reporting and documentation

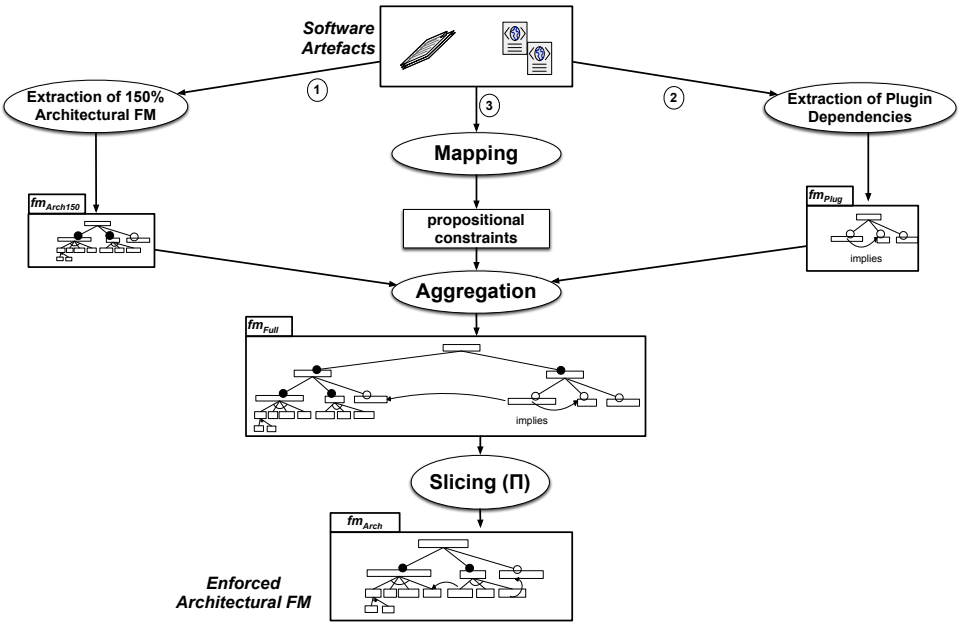


Figure 3.15: Process for Extracting fm_{Arch}

Extraction process

The general principle of the extraction is to combine two sources (an architectural model and a set of plugin dependencies) in order to synthesize a new integrated feature model representing the features of the architecture as well as their variability and their technical constraints. Fig. 3.15 summarizes the steps needed to realize the extraction process.

As a first step, a raw *architectural feature model*, noted $fm_{Arch_{150}}$, is extracted from a *150% architecture* of the system (see ①). The latter consists of the composition of the architecture fragments of *all* the system plugins. We call it a 150% architecture because it is not likely that a FraSCAti configuration may contain them all. Consequently, $fm_{Arch_{150}}$ does include all the *features* provided by the FraSCAti SPL, but it still constitutes an over approximation of the set of *valid combinations* of features of the FraSCAti family. Indeed, some features may actually *require* or *exclude* other features, which is not always detectable in the architecture. Hence the need for considering an additional source of information. We therefore also analyze the specification of the system plugins and the dependencies declared between them, with the ultimate goal of deriving inter-feature constraints from inter-plugin constraints. To this end, we extract a *plugin feature model* fm_{Plug}, that represents the system plugins and their dependencies (see ②). Then, we automatically reconstruct the bidirectional mapping that holds between the features of fm_{Plug} and those of $fm_{Arch_{150}}$ (see ③). Finally, we exploit this mapping as a basis to derive a richer architectural FM, noted fm_{Arch}, where additional feature constraints have been added. As compared to $fm_{Arch_{150}}$, fm_{Arch} more accurately represents the architectural variability provided by the system.

Extracting $fm_{Arch_{150}}$ The architectural feature model extraction process starts from a set of n system plugins (or *modules*), each defining an architecture fragment. In order to extract an architectural feature model representing the entire product family, we need to consider *all* the system plugins at the same time. We therefore produce a *150% architecture* of the system, noted $Arch_{150}$. It consists of a hierarchy of components. In the SCA vocabulary, each component may be a composite, itself further decomposed into other components. Each component may provide a set of *services*, and may specify a set of *references* to other services. Services and references having compatible *interfaces* may be bound together via *wires*. Each wire has a reference as *source* and a service as *target*. Each reference r has a *multiplicity*, specifying the minimal and maximal number of services that can be bound to r. A reference having a 0..1 or 0..N multiplicity is *optional*.

Note that $Arch_{150}$ may not correspond to the architecture of a *legal* product in the system family. For instance, several components may exclude each other because they all define a service matching the same 0..1 reference r. In this case, the composition algorithm binds only one service to r, while the other ones are left unbound in the architecture.

Since the extracted architectural feature model should represent the variability of the system of interest, we focus on its *extension points*, typically materialized by *optional* references [9]. The root feature of the extracted feature model (f_{root}) corresponds to the main composite (*root*) of $Arch_{150}$. The child features of f_{root} are the first-level components of *root*, the latter being considered as the main system features. The lower-level child features are then added through a recursive function. This function looks for all the optional references r of component c and, for each of them, creates an optional child feature f_r, itself further decomposed through a *XOR* or an *OR* group (depending on the multiplicity of r). The child features f_{c_s} of the group correspond to the set of all components c_s providing a service compatible with r.

An algorithm, detailed in [9], specifies how to retrieve this set of *matching components* from the 150% architecture. The set of components matching a given 0..N reference r are obviously those providing a service bound to r via a wire. In the case of a 0..1 reference, in contrast, all compatible services are not necessarily bound to it. Thus, the matching components are all those that provide a service having an interface compatible with reference r.

Extracting fm_{Plug} The extraction of the plugin feature model fm_{Plug} starts from the set of plugins $P = \{p_1, p_2, \ldots, p_n\}$ composing the system. This extraction is straightforward: each plugin p_i becomes a feature f_{p_i} of fm_{Plug}. If a plugin p_i is part of the system core, f_{p_i} is a mandatory feature, otherwise it is an optional feature. Each dependency of the form p_i *depends on* p_j is translated as an inter-feature dependency $f_{p_i} requires f_{p_j}$. Similarly, each p_i *excludes* p_j constraint is rewritten as an *excludes* dependency between f_{p_i} and f_{p_j}.

Mapping $fm_{Arch_{150}}$ **and** fm_{Plug} When producing $Arch_{150}$, we keep track of the relationship between the input plugins and the architectural elements they define, and vice versa. On this basis, we specify a bidirectional mapping between the features of $fm_{Arch_{150}}$ and those of fm_{Plug} by means of *requires* constraints. This mapping allows us to determine (1) which plugin provides a given architectural feature, and (2) which architectural features are provided by a given plugin.

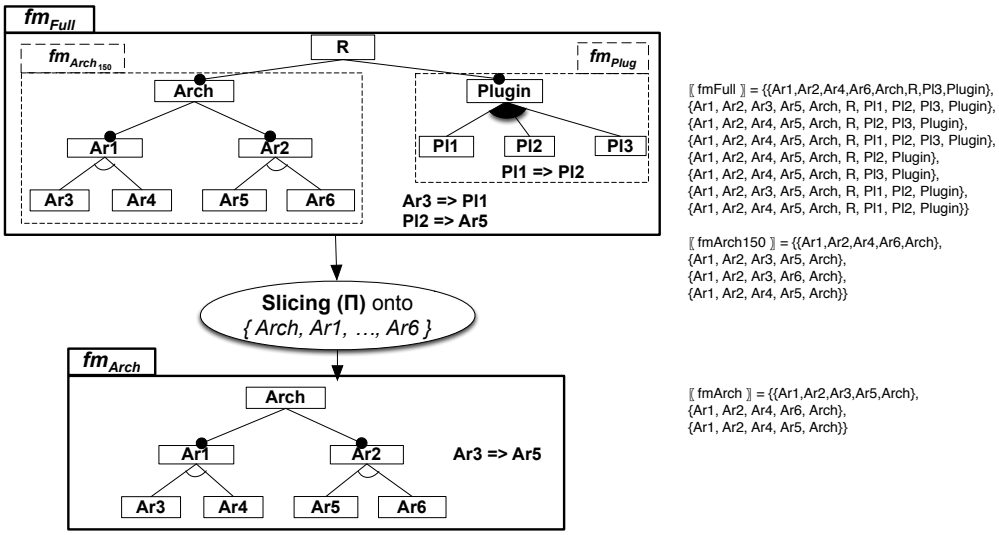


Figure 3.16: Enforcing architectural feature model using aggregation and slicing: an example

Deriving fm_{Arch} We now explain how we derive fm_{Arch} using $fm_{Arch_{150}}$, fm_{Plug}, the mapping between fm_{Plug} and $fm_{Arch_{150}}$, and an operation called *slicing*. We then illustrate the procedure using the example of Fig. 3.16. Intuitively, the variability and technical constraints induced by the plugin dependencies are *projected* onto the architectural model. In our case the use of plugin dependencies restricts the scope of the architectural feature model by precluding some unauthorized configurations in $fm_{Arch_{150}}$.

Projecting Variability onto the Architectural Model. First the two feature models fm_{Plug} and $fm_{Arch_{150}}$ are *aggregated* under a synthetic root $FtAggregation$ so that the root features of the inputs are mandatory child features of $FtAggregation$. The aggregation operation produces a new FM, called FM_{Full} (see Fig. 3.16). The propositional constraints relating features of fm_{Plug} to features of $fm_{Arch_{150}}$ are also added to FM_{Full}.

Second, we compute the projected set of configurations (see Definition 10) of FM_{Full} onto the set of features of $fm_{Arch_{150}}$ (*i.e.*, $\mathcal{F}_{fm_{Arch_{150}}} = \{Arch, Ar1, \ldots, Ar6\}$).

To realize the projection, we use an operation called *slicing* (see Definition 9). Given a subset of features, the slicing operator produces a new FM characterizing the projected set of configurations (see Definition 10).

Definition 9 (Slicing) *We define slicing as an operation on FM, denoted* $\Pi_{\mathcal{F}_{slice}}(fm) = fm_{slice}$ *where* $\mathcal{F}_{slice} = \{ft_1, ft_2, ..., ft_n\} \subseteq \mathcal{F}$ *is a set of features (called the slicing criterion) and* fm_{slice} *is a new feature model (called the slice).*

Definition 10 (Slice and projected set of configurations) *The result of the slicing operation is a new FM,* fm_{slice}*, such that:* $[\![fm_{slice}]\!] = \{ x \cap \mathcal{F}_{slice} \mid x \in [\![fm]\!] \}$ *(called the projected set of configurations).*

As several yet different feature models can represent a given set of configurations [284], we also take the feature hierarchy into account. In particular, we want to avoid slice feature models that are not readable and maintainable (*e.g.*, for a software architect or for users configuring the architecture) due to an inappropriate hierarchy. Therefore we consider that the new feature model produced by the slicing operation should have a hierarchy as close as possible to the hierarchy of the original feature model. Formal details can be found in [9].

Implementation of the projection (slicing). *Syntactical* strategies have severe limitations to accurately represent the set of configurations expected in the slice, especially in the presence of cross-tree constraints. Reasoning directly at the *semantic* level is required. The key ideas of our approach are to *i)* compute the propositional formula representing the projected set of configurations and then *ii)* reuse the reasoning techniques proposed in [49, 99, 32, 46] to construct a feature model from the propositional formula.

Example In the example of Fig. 3.16, the resulting slice is called fm_{Arch}. As we want to focus on the variation points of the architecture, it only contains the features' name of $fm_{Arch_{150}}$. Formally:

$$\Pi_{\mathcal{F}_{fm_{Arch_{150}}}}(fm_{Full}) = fm_{Arch}$$

We can verify that the relationship (see Definition 10) between the input FM, $[\![fm_{Full}]\!]$, and the slice FM, $[\![fm_{Arch}]\!]$, truly holds:

$$[\![fm_{Arch}]\!] = \{\ x \cap \{Ar1, Ar2, Ar3, Ar5, Arch\} \mid x \in [\![fm_{Full}]\!]\ \}$$

Importantly, we can notice that one configuration of the original $fm_{Arch_{150}}$ is no longer present in fm_{Arch}:

$$[\![fm_{Arch_{150}}]\!] \setminus [\![fm_{Arch}]\!] = \{Ar1, Ar2, Ar3, Ar6, Arch\}$$

Indeed the slice feature model fm_{Arch} contains an additional constraint $Ar3 \Rightarrow Ar5$, that was not originally restituted as such in $fm_{Arch_{150}}$.[4] It should also be noted that the hierarchy of the slice correctly restitutes the hierarchical decomposition of the architecture.

This very simple example already shows two key benefits of combining different variability sources and using the slicing operator. First, constraints, not originally present in the 150% architectural FM, are automatically restitued in a new architectural variability model and can be reported back to the software architect. Second, restrictions are applied on the over approximated configurations set characterized by the 150% architectural feature model. Therefore some configurations, actually not supported by the architecture, are now precluded.

Supporting the evolution of architectural feature mdoels

For each version of a plugin-based system like FraSCAti, the architectural feature model synthesized by the extraction procedure should be validated by the *software architect* (SA). In particular, the SA should control that the variability information and the characterized set of configurations do not contradict his/her intention and knowledge of the architecture. For example, the SA may consider that the mandatory status of some features in the extracted feature model is not appropriate.

[4] Similarly, the constraint $Ar4 \Rightarrow Ar6$ could be restituted in the model (using the information of the implication graph, see above). The slicing operator does not add this constraint because of the redundancy with $Ar3 \Rightarrow Ar5$.

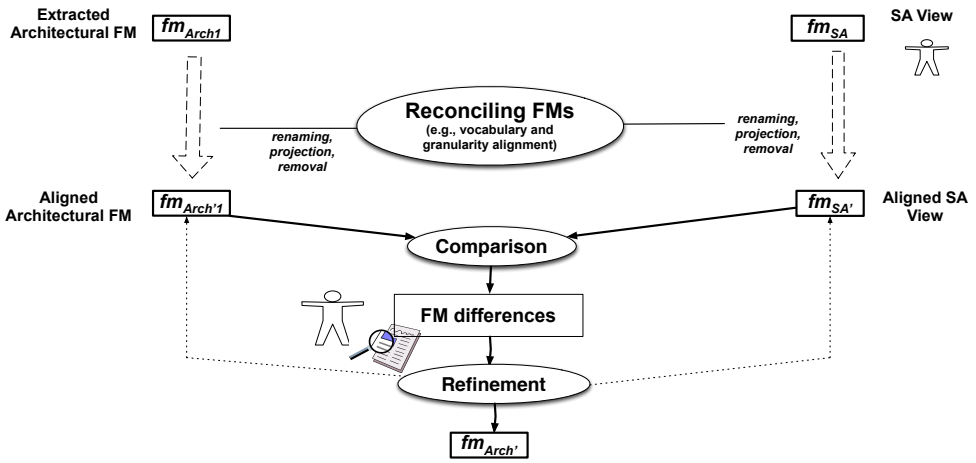


Figure 3.17: Process for integrating the SA knowledge

For assisting the SA, the extracted feature model can be compared with his/her mental representation and with older versions of architectural feature models. As a result, an appropriate support for comparing two feature models and reasoning about an *evolution* of a feature model is highly needed. We now detail some requirements and our tool-supported techniques.

First evolution. At the starting point of the re-engineering of FraSCAti as an SPL, an intentional model of the variability was elaborated by the SA. The resulting FM, denoted fm_{SA}, was the first available representation of the FraSCAti architecture (version 1.3, see Table. 3.2). The extraction process previously described was then applied to produce another representation (fm_{Arch_1}) for the same version of the architecture. Therefore, fm_{Arch_1} can be seen as an *evolution* of fm_{SA} given that the feature model originally elaborated by the SA has now evolved to an automatically extracted feature model.

The absence of a *ground truth* – a feature model for which we are certain that each combination of features is supported by the SPL architecture – makes uncertain the accuracy of the variability specification expressed in fm_{Arch_1} as well as in fm_{SA}. As both the software architect FM and fm_{Arch_1} may represent differently the variability of the architecture, there is need to *reconcile* and *refine* the two feature models. The result of this process is a new FM, $fm_{Arch'_1}$, that integrates the intentional variability and the SA knowledge of fm_{SA} and the explicit variability expressed by fm_{Arch_1}.

Versions and evolutions. As any software project, the FraSCAti architecture evolves. Many features and dependencies are added and removed. Naturally, the extraction procedure is reiterated on different versions (*e.g.*, version 1.4) of a FraSCAti architecture, producing different feature models. Nevertheless, the confidence of the resulting feature models remains unclear: *i)* the extraction procedure may be faulty (*e.g.*, inadequate for a specific version of FraSCAti); *ii)* the variability and the constraints may not be correctly documented in the architecture artefacts; *iii)* the SA knowledge may not be taken into account.

Managing the evolutions. For controlling and hopefully validating the evolution of an FM, the SA should be able to understand and exploit the *differences* between two feature models. A possible solution is to elaborate, for each version of a FraSCAti architecture, a new feature model representing the current variability and then compare it with the extracted FM. Nevertheless, the elaboration from scratch of a new feature model (like the SA did for version 1.3) is time-consuming and error-prone. There is an opportunity to *reuse* feature models resulting from a previous version.

Support for managing evolutions. Fig. 3.17 presents the overall process for comparing two feature models (*e.g.*, for comparing an extracted feature model with a feature model designed by the SA). In the following, we describe dedicated techniques related to the evolution of feature models for supporting the SA activities, namely reconciliation, comparison and refinement.

Reconciling feature models Let us consider fm_{SA} and fm_{Arch} of Fig. 3.17. The SA should be able to determine if the variability choices in fm_{SA} comply with what is expected by himself (*i.e.*, as specified in fm_{Arch}), and vice-versa. In case variability representations are conflicting, the SA can refine the architectural feature model. Similar observations can be made when reasoning about two different versions of a FraSCAti architecture.

There are three steps to support the SA architect in reconciling feature models. The first one is to apply pre-directives for renaming features due to vocabulary mismatch among the feature models. The second step is to deal with the granularity mismatch (*e.g.*, some features in one feature model are not present in the other model). The SA needs to remove features, potentially involved in many cross-tree constraints. For a sound removal of features, we use the slicing operation. For example, the removal of two features *Felix* and *Equinox* of fm_{SA}, leading to a new feature model fm'_{SA}, corresponds to the following slicing operation:
$fm'_{SA} = \Pi_{\mathcal{F}_{fm_{SA}} \setminus \{Felix,Equinox\}}(fm_{SA})$

Once feature models have the same vocabulary and granularity of information, it is possible to *compare* feature models. The third step is here to compute and present *differences* of the two feature models in a comprehensible manner to the SA. The problem of feature model differences is a general problem that may occur in other contexts (see Section Automated feature model management). We present here only the techniques relevant to our specific context.

Comparison through syntactic and semantic diff. From a syntactical perspective, the elements to be considered in the diff are features, feature hierarchies, feature groups or implies / excludes. For instance, it is useful to determine feature groups (Xor and Or) that are in a feature model but not in in the other. (We consider that two feature groups are equal if and only if their parent features match and their child features match.)

Though the syntactic diff is useful, a practitioner also wants to understand the difference between the two feature models in terms of *configuration semantics* (*i.e.*, in terms of sets of configurations). We now address semantically the list of differences. We translate the two feature models into two formulae. Working at the level of abstraction for Boolean variables may produce unexploitable results for a practitioner. Stated differently, a practitioner wants to understand differences in terms of feature modelling concepts rather than in terms of a propositional formula. We thus take care of producing meaningful information based on the

analysis of the two formula. A first strategy consists in analyzing separately each formula and then performs the differences of the information produced. For instance, we can report on the differences *w.r.t.* implications or exclusions. A second general strategy consists in producing relevant information based on the logical combinations of the two formula.

Step-wise refinement. Once differences have been identified and understood, the SA can *edit* the two feature models: *i)* change the variability associated to features (*e.g.*, set optional a mandatory feature); *ii)* add and remove some constraints (*e.g.*, implies constraints); *iii)* modify the feature hierarchy.

The edits to a feature model (*e.g.*, fm_{Arch}) change its syntactic and semantic properties. Once edits are applied, the differences with another feature model (*e.g.*, fm_{SA}) should be re-computed. Therefore managing differences is a multi-step, incremental process. Edits are incrementally applied on the two feature models until obtaining a satisfying relationship between the two feature models.

Implementation. We need a practical support for using the techniques previously described:

- *Extraction* support: the procedure aiming to extract the variability model of the plugin-based architecture at a certain time (fm_{Arch}).
- *Evolution* support: the set of feature model operations designed to assist the architect in monitoring the evolution of the plugin-based architecture.

For both tasks, we rely on FAMILIAR, a language for combining feature model operators.

Evaluation

Performance evaluation. Theoretically, slicing and difference' operations can induce severe computational costs. However, we found that the order of complexity of feature models encountered in FraSCAti is manageable. Feature models exhibit lots of constraints but at worst only 123 features (see Table 3.2, page 106) when combining $fm_{Arch_{150}}$ and fm_{Plug} for the version 1.5. At this scale, we observed no difficulty. The operations on feature models can be efficiently executed in a few seconds using our implementation of the slicing operation and differencing techniques.

Practical evaluation. We applied the tool-supported techniques previously described on different versions of FraSCAti (see Table 3.2). P. Merle, principal FraSCAti developer for six years now, plays the role of the SA in this study. Specifically, we aim at assessing them regarding the two main challenges:

- **(RQ1) Extraction of variability:** Is the extraction procedure accurate or faulty? Are the properties of the produced feature models coherent with what is expected by the SA? To what extent is the SA knowledge needed for recovering the architectural variability? For this purpose, we determine the variability information inferred by the extraction procedure and analyze the differences between fm_{Arch} and fm_{SA}. We also report qualitative insights gained when the SA validates the extracted feature model.
- **(RQ2) Evolution of variability:** Are the differencing techniques exploitable for the SA? Can an evolution be controlled and validated by the SA? We apply previous techniques and report similar quantitative and qualitative observations for two other versions of FraSCAti.

Version	$fm_{Arch_{150}}$	fm_{Plug}	mapping	fm_{Arch}	core features (deduced)	implies constraints (deduced)	bi-implies constraints (deduced)
1.3	50 features $\approx 10^{11}$ config.	41 features 81 constraints	78 constraints	$\approx 10^{6}$ config.	12	9	5
1.4	53 features $\approx 10^{11}$ config.	56 features 87 constraints	80 constraints	$\approx 10^{7}$ config.	12	10	5
1.5	60 features $\approx 10^{14}$ config.	63 features 96 constraints	92 constraints	$\approx 10^{8}$ config.	12	13	7

Table 3.2: Experimental results: properties of the feature models

Extraction (RQ1, key results) Considering different versions of the FraSCAti project (see Table 3.2), the extraction procedure deduces many constraints and drastically restricts the configuration set of $fm_{Arch_{150}}$. The SA validates the variability recovered by the procedure. It even encourages him to correct his initial model. We gain better confidence in the accuracy of the extraction procedure by reiterating the process on different versions of FraSCAti. In some specific cases though the extracted feature model contains faulty variability information. In this case, we have to rely on the knowledge of the SA.

Evolution (RQ2, key results) The differencing techniques appear to be meaningful for the SA. It allows the SA to control the properties of extracted feature models and in turn integrate his knowledge. It also allows the SA to understand and validate the evolutions of the FraSCAti architecture, for example, by controlling what implies constraints have been added and removed between two versions.

Insights about RQ1 and RQ2 I report an excerpt of qualitative insights – more details can be found in [9].

Insight #1 (software architect corrections) For the version 1.4 of FraSCAti, we identified 13 features that are present in fm_{Arch} but not in fm_{SA}. Among others, two metamodels used by the SCA parser, three bindings, two SCA properties, two implementations and one interface were missing. Several reasons were given by the SA:

- **accidental complexity:** the SA recognizes that some features were missing in his feature model. Given the complexity of the FraSCAti project, this is not surprising that the SA forgets some features. Some oversights are related to "helper" features of FraSCAti (such as the features features binding factory or juliac) that are generally not used by developers, while other oversights were qualified as more relevant from a configuration perspective (additional metamodels and binding types).

- **modelling intention:** the SA reveals that he *intentionally* ignored some features in fm_{SA}. He argued that there are mandatory features (*e.g.*, every FraSCAti configuration has a Java interface) and that his focus was on variability rather than commonality. We indeed verify the mandatory nature of the features (*e.g.*, sca_interface_java) in fm_{Arch} (see above).

Another example related to the way features are modeled concerns a feature of fm_{Arch}, juliac, not modeled in fm_{SA}. By simplification, features juliac and delegate-membrane-generation have been merged by the SA into an unique feature MembraneGeneration.

- **obsolete features:** for the feature services, the SA explains that this architectural element is an empty composite that "could have been used but have not yet an interest".

Insight #2 (debating with automated extraction) Three subtle situations of variability mismatch have been encountered and are interesting to explain:

- feature generators is optional and its children tinfi_oo_1, osgi are forming an Or-group in fm_{Arch} whereas feature generators is mandatory and its children tinfi_oo_1, osgi are all optional in fm_{SA}. At first glance, the difference seems important but the intention of the SA is actually similar to the variability expressed in fm_{Arch}. In terms of sets of configurations, fm_{SA} authorizes four combination of features {generators, tinfi_oo_1, osgi}, {generators, osgi}, {generators, tinfi_oo_1}, and {generators}. fm_{Arch} authorizes exactly the same set, except {generators}. It means that in both cases a configuration of a FraSCAti architecture may have zero or some *concrete* generators (*i.e.*, {tinfi_oo_1, osgi}). The feature {generators} can be seen as an *abstract*[5] feature.

  As a result, the two feature models, though modelling differently the variability, have the same intention. It has been decided by the SA to keep the solution of the extraction procedure.

- feature fractal_bootstrap_class_provider is mandatory in fm_{SA} and one of its child feature tinfi_oo is mandatory. On the contrary, fractal_bootstrap_class_provider is optional in fm_{Arch}, and its children form an Or-group. The discussions with the SA reveal that, indeed, the architecture of FraSCAti authorizes a configuration *without* fractal_bootstrap_class_provider. The initial intent of the SCA was to state, that this feature is *often*[6] necessary. He explained the mandatory status of the feature tinfi_oo as a *default* implementation. Nevertheless, the SA recognized that fm_{Arch} accurately restitutes the flexibility of the architecture.

- the feature compiler_provider is optional in fm_{Arch} but mandatory in fm_{SA}. The SA confirms that a FraSCAti architecture has not necessarily to embed a complete Java compiler – minimal ($\leq 4Mo$) FraSCAti architecture for embedded systems can thus be derived and deployed. Therefore fm_{Arch} accurately models the variability of the feature compiler_provider.

Insight #3 (missing constraints) For each version (see Table 3.2), we identified a dozen of implies constraints expressed in fm_{Arch} but not in fm_{SA}. All constraints were validated by the SA, recognizing that the constraints have been forgotten.

Concluding remarks. Overall the results show the software architect increases the *quality* of architectural feature models (*i.e.*, better specifying variability and thus avoiding some unsafe configurations) compared to a feature model that is manually designed or that does not integrate all variability descriptions of the system. Furthermore the architectural feature model takes into account both the software architect viewpoint and the variability actually supported by the system. Without the feature model management support exposed in the

[5] In [304], Thüm *et al.* define a feature as abstract, "if and only if it is not mapped to any implementation artifacts". They "call all other features non-abstract or concrete, *i.e.*, a concrete feature is mapped to at least one implementation artifact". It corresponds to our case.

[6] Many constraints of fm_{Arch} involve features tinfi_oo, osgi_provider, julia, thus confirming that their parent feature fractal_bootstrap_class_provider is needed in many configurations.

article, obtaining similar results would not be possible. Qualitative insights validate the adequacy of our support for reverse engineering variability. They also show the limits of a modelling approach (as done by the SA) as well as the limits of a reverse engineering approach. In other words, both variability modelling (see Chapter 2) and reverse engineering techniques (this chapter) should be combined.

replication

Further details and material (including feature models and FAMILIAR scripts) about the experiment are available here: https://github.com/FAMILIAR-project/FraSCAtiVariabilityEvolution

3.5 Wrap-up, applicability, and limitations

I have shown that reverse engineering can provide the necessary automation for obtaining high-quality variability models. The presented techniques can also mine variability information that is otherwise time-consuming and error-prone to synthesize. Several kinds of artefacts can be considered: informal, textual descriptions of individual products; architectural information and dependencies files; client-side artefacts of a Web configurator; or tabular data. It should be noted that the applicability is far broader to what I have presented in this chapter. For instance, we described how to reverse engineer a family of languages in [308]. In fact, there is a large community involved in reverse engineering variability (see, *e.g.*, the REVE workshop http://reveworkshop.github.io/, co-organized with the SPLC conference since 2013).

In a sense, reverse engineering is a possible answer to the limitations identified in previous chapter. The process can be repeated and variability knowledge can be automatically extracted. Yet, as shown, reverse engineering can be incomplete and unsound as well, since targeted artefacts may contain partial variability information. For instance, the reverse engineering procedure of FraSCAti has limited interest in case the software architect does not correct some inconsistencies. Another limitation is that automation pays off under the conditions some knowledge is injected into the reverse engineering. Attributed feature models obtained out of tabular data can be unreadable and unexploitable without expert knowledge. From this regard, I have contributed to the foundations, design, and development of techniques to supervise the reverse engineering process.

Overall, the potential of reverse engineering mostly resides in the ability to i) automate some tedious and error-prone modelling tasks; ii) confront, refine, and augment an existing variability model with variability information. In both cases, the involvement of an expert or developer seems either beneficial or mandatory to obtain an integrated, high-quality model of variability. Reverse engineering is an interesting toolbox, but some limitations remain. It is time to explore the third research direction of this manuscript.

Chapter 4

Learning Software Variability

In this chapter I present a set of methods and techniques based on supervised, statistical machine learning to model variability constraints and non-functional properties (performance) of a configurable system. This chapter mainly focuses on scenarios in which persons (developers, maintainers, testers, *etc.*) aim to refine or augment variability models. Compared to the two previous chapters, learning-based approaches require the actual build, executions, and observations of a system under study.

Section 4.1 introduces and evaluates a systematic process based on "sampling, observing, learning" for inferring constraints among options. Section 4.2 introduces the use of adversarial machine learning to enforce a configuration classifier or pinpoint problematic cases. Section 4.3 evaluates the effect of sampling strategies on the effectiveness of learning methods for performance prediction. Section 4.4 shows that learning techniques can be effective in the huge configuration space of the Linux kernel and remain so for a period of 3 years provided that suitable transfer techniques are used.

Contents

4.1 Learning variability constraints

The content of this section is adapted from the following publications:
P. Temple, J. A. Galindo Duarte, M. Acher and J.-M. Jézéquel, 'Using Machine Learning to Infer Constraints for Product Lines', in *Software Product Line Conference (SPLC'16)*, Beijing, China, Sep. 2016. DOI: 10.1145/2934466.2934472. https://hal.inria.fr/hal-01323446
P. Temple, M. Acher, J.-M. Jézéquel and O. Barais, 'Learning-Contextual Variability Models', *IEEE Software*, vol. 34, no. 6, pp. 64–70, Nov. 2017. https://hal.inria.fr/hal-01659137

Not all combinations of options (a.k.a. configurations) are possible in a software configurable system. Without proper constraints and a too permissive configuration space, users may derive invalid products and developers would deliver products full of bugs or of bad quality. Unfortunately, the specification of constraints is known to be a time-consuming and error-prone task. The number of potential interactions and dependencies grows as the number of options increases in a software system. In particular, it is easy to forget a constraint and thus mistakenly authorize some invalid products. Developers thus struggle to identify and track constraints throughout the (re-)engineering of more and more complex systems. In essence, capturing constraints of an existing system boils down to anticipating all possible combinations of features and making explicit the configuration *knowledge* that is somewhat implicit in various kinds of artefacts (documentation, code, test cases, *etc.*). The domain knowledge of experts and developers may not be sufficient or up-to-date to capture all constraints. Stated differently, there are shortcomings and limitations to only rely on a manual effort as exposed in the Chapter Modelling Software Variability. Furthermore, the static analysis of artefacts, though extremely useful and effective in many situations (see Chapter 3 about reverse engineering variability) may be hard to develop or exhibit limitations in terms of soundness and completeness.

Another approach, more and more considered, is to dynamically execute the software system over different configurations and statistically *learn* what combinations of options are not appropriate and thus should be used to further constrain the system. In practice, a first key step is to gather a sample of configurations together with their labels. It requires compiling (or executing) some configurations. The resulting qualities of the derived product (*e.g.*, whether it builds or fails) are automatically observed and labels are attached to configurations. Then, statistical machine learning algorithms can identify what individual features or combinations of features (if any) are causing their non-validity (*e.g.*, a product does not build). The generated sample from the first step is exploited to train a classifier (*e.g.*, a decision tree) that can classify any remaining product of the system, even those that have never been tested. Some constraints can eventually be extracted out of a classifier to avoid the derivation of products classified as invalid.

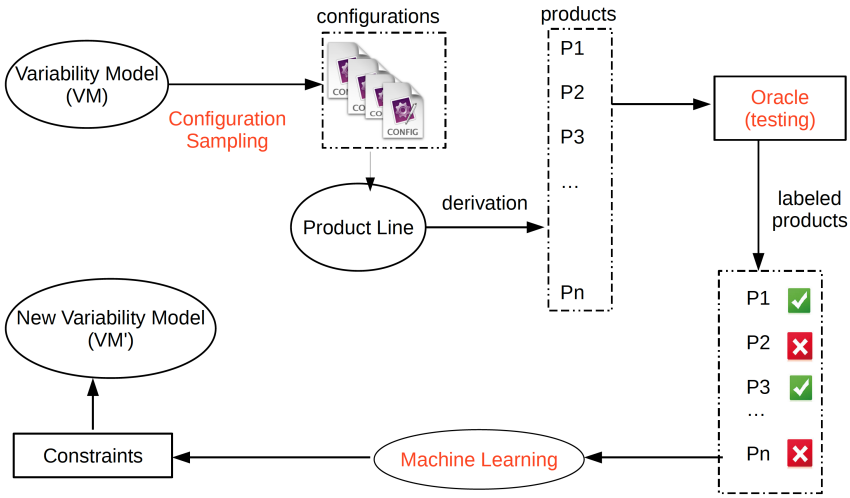


Figure 4.1: Sampling, testing, learning: process for inferring constraints of product lines

I am giving more details about this learning approach in the two next subsections. Section 4.1.1 first introduces the seminal ideas developed in the context of Paul Temple's thesis. I describe how the approach can be used to find subtle constraints of an industrial video generator used in the MOTIV project. Section 4.1.2 extends this line of work along two perspectives: (1) the learning of constraints can be parameterized by a high-level, performance objective; (2) constraints among the contextual features and the software configuration space can be automatically synthesized.

4.1.1 Using machine learning to infer constraints

We describe a general method in which we leverage machine learning techniques to infer the constraints of a product line.

Learning constraints: rationale and method

Figure 4.1 describes the process we followed. We assume that there is a variability model that documents the configuration options of the product line under consideration. In such a variability model, options can be Boolean features or numerical attributes. From a configuration (see Definition 11), product line artifacts are assembled and parameters are set to derive a product. The variability model may already contain some *constraints* that restrict the possible values of options (*e.g.*, the truth value of a Boolean feature implies the setting of a specific numerical value of an attribute).

Definition 11 (Configurations, variability model) *A* configuration *is an assignment of values for all options of a variability model. A configuration is valid if values conform to constraints* (e.g., *cross-tree constraints over attributes). A variability model VM characterizes a set of valid configurations denoted $[\![VM]\!]$.*

A first step in the process is *configuration sampling*. It consists in producing valid configurations (see Definition 11) of the original variability model VM. The set of sampled configurations is a subset of $[\![VM]\!]$. Numerous strategies can be considered such as the generation of T-wise configurations, random configurations, *etc.* [211, 87, 172, 124, 144, 245, 134, 273]

Second, an *oracle* is reused or developed. It tests the validity of the derived products corresponding to configurations. The notion of validity is specific to a product line and a usage context. It may refer to the fact the product does compile, does not crash at run-time, passes the test suite, and/or does meet a particular quality of service. In the remainder, we use the term *"faulty" configuration* for referring to such irrelevant products (see Definition 12). An oracle is used to create two classes of configurations: It labels the configurations in the sampling as either faulty or "non faulty".

Definition 12 (Oracle and faulty configuration) *An oracle is a function that takes a derived product as input and returns true or false. A faulty configuration is a configuration for which the oracle returns false when taking as input the corresponding derived product.*

A third step is to use a *machine learning* procedure that operates on the labeled configurations and automatically infers constraints. A new variability model VM' is created by adding the newly identified constraints to the original variability model. Therefore, the new variability model VM' is a specialization [303] of VM and $[\![VM']\!] \subset [\![VM]\!]$. In other words, VM' forbids faulty configurations that were initially considered as valid in VM.

Instead of using machine learning, an alternate and sound approach is to remove faulty configurations from the original variability model. It consists in negating all faulty configurations and then making their conjunctions (see Definition 13). However, this approach is very limited since (1) it only removes a typically small number of configurations – only those that have been sampled and tested; (2) it does not identify which configuration options and values are involved and the root causes of the fault.

Definition 13 (Sound approach) *Given a set of faulty configurations* $\{fc_1, fc_2, ..., fc_n\}$ *a sound approach computes a new variability model* VM' *such that* $VM' = VM \wedge \epsilon$ *where* $\epsilon = \bigwedge_{i=1..n} \neg fc_n$

A simple removal of faulty configurations is thus not a viable solution for product lines exhibiting a large number of configuration options or numerical values. As an oversimplified example, let say we have faulty configurations $\{fc_1, fc_2, fc_3, \ldots, fc_n\}$. Among values of attributes and features, the attribute A varies as follows: $A = 0.265$ in fc_1, $A = 0.26$ in fc_2, $A = 0.275$ in fc_3, ..., $A = 0.29$ in fc_n. With a basic case by case extraction, we cannot infer that (perhaps) $0.26 < A < 0.3$. The use of machine learning can improve the inference of constraints through the prediction of ranges of values that make configurations faulty.

The expected benefit is to discard much more faulty configurations with the inference of constraints: Figure 4.2 summarizes the potential of machine learning. A rectangle is used to represent $[\![VM]\!]$. The set of configurations that can be detected as faulty is represented as red clouds/rectangles in Figure 4.2. The precise set is a priori unknown; this is precisely the problem.

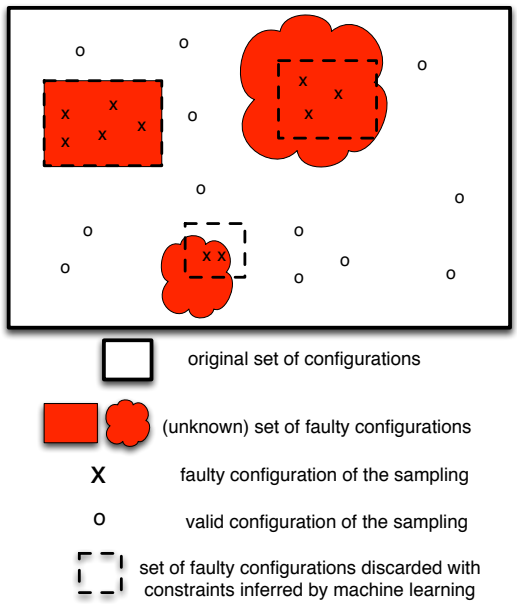


Figure 4.2: Constraining the configuration space

Faulty configurations detected by an oracle, are included in this set (see crosses in Figure 4.2). The enumeration and testing of configurations for covering the whole set of faulty configurations will take a large amount of resources and time. In response, machine learning can infer a set of constraints delimiting *sets* of faulty configurations (instead of only forbidding individual configurations). Thanks to learning and prediction, we expect the capture of additional faulty configurations *without the cost of testing those configurations*.

The ideal case is that machine learning accurately classifies configurations as faulty, including configurations that were not part of the sampling. We represented that as the dashed rectangle exactly corresponding to the red rectangle (see left-hand side Figure 4.2). However, machine learning might produce false positives. That is, some configurations are classified as faulty whereas they are actually valid from the oracle's perspective. An example is given in Figure 4.2 with the red cloud and dashed rectangle at the bottom: some configurations are included outside the red cloud and in the blank area. Another kind of misclassification can happen when the dashed rectangle is included in the red cloud (see right-hand side of Figure 4.2). In this case, machine learning fails to classify some configurations as faulty and is incomplete. Despite specific cases in which machine learning can be unsound and incomplete, we expect that, in general, learning constraints enables to capture more faulty configurations than a simple conjunction of negated configurations.

Case Study

In this section we apply and evaluate the proposed method with a real-world product line, *i.e.*, MOTIV which is a highly-configurable video generator developed in the industry (see Section In search of the right variability language and models). Our objective is to address the following research questions:
RQ1) *Do extracted constraints make sense for a computer vision expert?*
RQ2) *What is the precision and recall of our learning approach?*
RQ3) *What are the strengths and weaknesses of our approach compared to existing techniques?*

Problem. The software generator is written in Lua and implements numerous complex, parameterized transformations for synthesizing variants of videos [7, 124]. Users quickly noticed that the specification of constraints in the variability model is crucial for the video generator. Without constraints, many configurations lead to the generation of unrealistic video variants, due to the incompatibility between features and attributes' values. Beyond usability problems, this is an issue for two major reasons. First, the production of videos has a cost (about half an hour of computation on average per video variant). As a result, the synthesis of large datasets with thousands of video variants (as originally planned by industrials) would produce a lot of irrelevant videos, thus wasting resources and computation. Second, tracking algorithms performed on the synthesized videos are computationally expensive, which, in case of irrelevant videos, is again a waste of time and resources. *Our early modelling effort in Section 2.4 for properly formalizing the variability was thus not sufficient; we need to enforce the generator with constraints to make it usable and useful for practitioners.*

Although some basic constraints have been manually specified, the generator still produced irrelevant video variants. In order to capture additional constraints and gather some knowledge, we have made several iterations with the developers of the video generator through meetings and mail exchanges. Finally, we came to the conclusion that either an analysis of the Lua source code or a further effort for manually specifying constraints presents strong limitations. It is mainly due to the fact that (1) the configuration space is extremely large (see hereafter for more details); (2) there is not enough knowledge to comprehensively capture constraints over features and attributes' values.

A manual exploration of the configuration space requires substantial efforts for both setting configuration values, assessing the quality of the output (videos), and learning from defects. We thus propose to use the method we described in the previous section to automate all these tasks, including the inference of constraints with machine learning.

We now detail how we instantiated every part of our method (sampling, testing, learning) within our case study.

Figure 4.3 presents the entire process of extracting constraints of the video generator. A set of configuration files is first sampled from the variability model; it acts as a training set. The Lua generator derives a video variant for each configuration. An oracle is then used to label videos as faulty or non faulty by computing the quality of each video. Finally, a machine learning process is executed to extract the constraints and re-inject them into the original variability model. We now detail each step of the process.

Generating a training set out of the variability model. In the MOTIV case study, the variability model exhibits numerous features and attributes whose ranges of values are reals and continuous. In total, the variability model contains: 42 real attributes, 46 integer attributes, 20 Boolean features, and 140 constraints (mainly constraints specified by the experts). The ranges vary between 0 and 6 for the integers domains, and in average between zero and 27.64

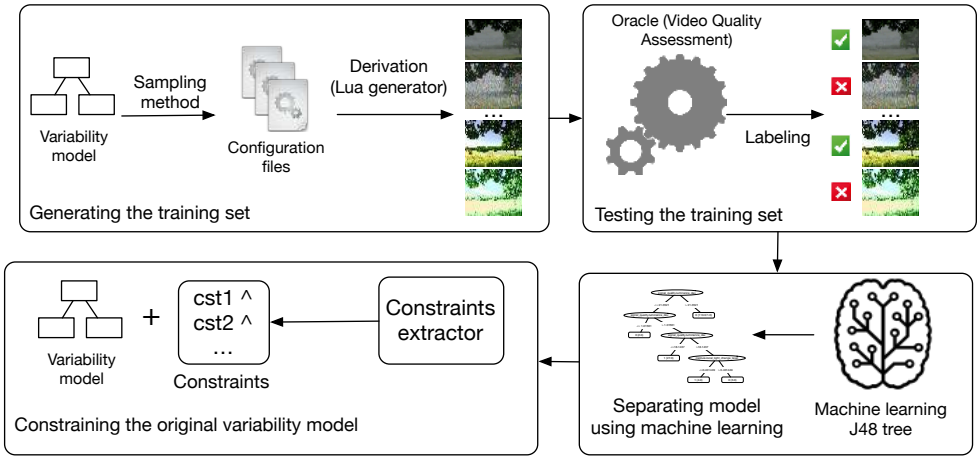


Figure 4.3: Learning method on the video generator

for the real domains with a precision of 10^{-5}. This would end up in approximately a total of 10^{103} configurations (not considering constraints): 2^{20} (because of the boolean variables) $\times 6^{46}$ (because of the integer variables and) $\times 2764000^{42}$ (because of the real variables). Overall the variability model presents the particularities of encoding a large configuration set with lots of non-Boolean values.

To generate a training set for the machine learning process, we need to produce a set of valid configurations. Different sampling techniques [211, 87, 172, 124, 144, 245, 134, 273] can be considered but some of them are not applicable to our case since we have to deal with real and integer values. We implemented the following procedure. First we randomly pick a value for each attribute within the boundaries of its domain. Then, we propagate the attribute values to other values with a solver; the goal is to avoid invalid values. We continue until having a complete and valid configuration. We reiterate the process for collecting a sample of configurations.

Oracle. Some videos of the generator are not of interest for computer vision algorithms and humans. Typically, these are videos in which the vision system cannot perceive anything or cannot distinguish moving objects from other ones (*e.g.*, distractors). Image Quality Assessment (IQA) typically tries to understand the conditions under which the vision system is likely to fail this kind of distinction. We implemented an IQA oracle, presented in [112], that can automatically assess whether a video is faulty. The principle is to perform a Fourier transformation and to reason about the resulting distribution of Fourier frequencies. Such a technique evaluates the quality of a single image. To reduce the time and cost of the oracle, we sample a video into a smaller set of images. After applying the IQA method on sampled images, we aggregate results to decide whether a video is faulty or not. We empirically set a threshold: If, at least, half of the images are declared faulty, then the whole video sequence is considered faulty.

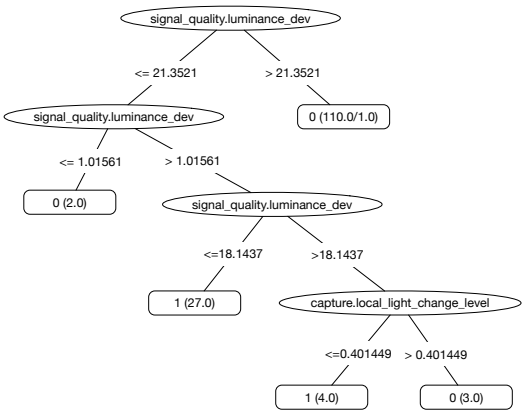


Figure 4.4: An excerpt of the decision tree built from a sample of 500 configurations/videos

Machine Learning (ML). Using our oracle, we assigned a faulty or non faulty label for each video of the sample. We use a *Machine Learning (ML)* algorithm to understand the relationship between faultiness of videos and features/attributes' values. Different ML methods exist. Some of them are sophisticated and hopefully make only a few classification errors. Others are less advanced but are much more understandable when visualizing the output model. In our case, we wanted to obtain a high level of understandability when extracting constraints. Specifically, we employed *Binary Decision Trees*.

Figure 4.4 presents an excerpt of the decision tree obtained from the Weka[1] software. In this tree:

- *ovals* represent features (written inside) on which decisions will have to be taken;
- *labels over edges* represent threshold value to decide which path to take;
- *rectangles* represent leaves of the tree and groups of configurations that are mainly of the same class.

Leaves present several pieces of information. First, there is the label of the most represented class. In our case it is either '1' (faulty) or '0' (non faulty). Second, the number of configurations associated to the label. These are classification errors, typically configurations that are at the boundary of two classes.

As an example, configurations having a value higher than 21..3521 set for feature "signal_quality.luminance_dev" will reach the leaf in the top right corner of Figure 4.4. It means that 110 configurations (out of 147) are labelled '0' and one is classified as faulty.

Extracting constraints. Once the decision model has been created, we traverse the tree and reach every leaf. Only leaves labeled '1' (faulty) are remembered meaning their path is extracted. We consider that a path is a set of decisions, where each decision corresponds to the value of a single feature. We create new constraints by building the negation of the conjunction of the different decisions to make along the path to reach a faulty leaf. In case features are repeated, some simplifications are performed. In the example of Figure 4.4, we can extract the two following simplified constraints:

[1] http://www.cs.waikato.ac.nz/ml/weka/

```
!(signal_quality.luminance_dev > 1.01561 && signal_quality.luminance_dev <= 18.1437)
!(signal_quality.luminance_dev <= 21.3521 && signal_quality.luminance_dev > 18.1437
    && capture.local_light_change_level <= 0.481449)
```

Experiments

Experimental Setup. To generate a training set, we sampled 500 configuration files from the MOTIV variability model. All of them are given to the video generator to create 20 seconds videos. The process of deriving associated video variants takes about 30 minutes on average per video. As we have to create numerous videos, we used a cloud-based architecture for distributing the computations. To decrease the influence of randomly creating training set (which could result in advantageous or disadvantageous settings), we run the experiment of learning and validating results 20 times. After the synthesis of videos, the IQA oracle assigned "non faulty" or "faulty" labels to videos. In total, the oracle labelled 53 videos as faulty on average, *i.e.*, roughly 10% of the videos.

We used decision trees with Weka to perform the ML part. Weka offers different implementations of Binary Decision Trees. We used J48 since it is the most widely used. Various options can be tuned to increase the classification performances. This process of selecting the best set of parameters is application-dependent, so we used the default ones set by Weka.

Results. We report on the results of the three research questions.

RQ1) Do extracted constraints make sense for a computer vision expert?

The first research question focuses on the readability and comprehensibility of extracted constraints from an expert point of view. To be able to answer our question, we asked to a computer vision expert and an advanced user of the video generator whether the extracted constraints make sense. The expert told us that constraints are globally understandable and actually help understanding interactions that can occur between features/attributes. Importantly, he noted that constraints are in line with the definition of the oracle: Some combinations of values can indeed participate in the degradation of the perception of video contents.

Specifically, Figure 4.5 shows a short constraint only constituted by two terms. This constraint puts together two image quality criteria (blur and static noise) that can indeed degrade the quality of videos. In Figure 4.6, the constraint involves other features that have an effect on the quality of videos: compression and illumination. Interestingly, blur is present again. In Figure 4.7, blur, compression, and dynamic noise are features that are related to the quality criteria of videos as well. Overall, all features/attributes previously mentioned make sense *w.r.t.* the IQA oracle we implemented. Too much noise, poor illumination and blur: all these factors can indeed degrade the quality of videos and produce the kind of videos our oracle is able to detect as faulty.

In general, the extracted constraints make sense and the answer to RQ1 is positive. However, there is room for improvement. Specifically the expert complains about the presence of features/attributes that are not relevant and disturb the reading. For instance, in Figure 4.6, $vehicle1.identifier <= 11$ does not make much sense. Indeed, the kind of vehicles should not have any influence on the definition of faulty videos. The expert has to somehow ignore this kind of information.

RQ2) What is the precision and recall of our ML approach?

```
!(signal_quality.blur_level > 0 && signal_quality.static_noise_level <=0.135519)
```

Figure 4.5: A constraint extracted from our case study

```
!(signal_quality.blur_level < 0 && signal_quality.compression_artefact_level >
    0.363438 && capture.illumination_level <= 0.609669 && signal_quality.
    compression_artefact_level>0.436673 && vehicle5.trajectory >6 && vehicle1.
    identifier <=11)
```

Figure 4.6: A constraint extracted from our case study

In the rest of this section, we consider that a ML approach computes a new variability model denoted VM' such that $VM' = VM \wedge \Delta$ where Δ is a conjunction of inferred constraints.

The overall classification performance of ML is not perfect, *i.e.*, 90.56% on average after training the decision tree on 500 configurations/videos. This is due to the fact that despite a perfect knowledge over the training data, ML tries to avoid over-fitting to be able to generalize what have been learnt. To have a better idea of the number of errors that our approach is likely to perform, we tested the output classification model while producing a new data set with configurations that were not in the 500 original configurations.

To do that, we generated another set of 4000 configurations and videos. We used again a cloud-based infrastructure to synthesize these variants. Our oracle labelled every video of this new data set. It resulted in 370 faulty videos on average. Then we compared the decision of the oracle with the decision of the variability model augmented with extracted constraints. We run the experiment 20 times by randomly changing the training set (500 configurations) as well as the validation set (4000 configurations).

In particular, we are interested to know whether a configuration labelled as faulty with the oracle is now forbidden by VM'. This comparison is usually performed through a confusion matrix presented in Table 4.1.

In this table, columns are the labels given by the oracle and rows are labels given by our variability model. Cells present the average of classification over 20 runs as well as standard deviation (under parantheses). The main diagonal of this matrix tells us where the two labels agree. The other diagonal provides classification errors of our variability model compared to the oracle:

```
!(signal_quality.blur_level <= 0 && signal_quality.compression_artefact_level <=
    0.363438 && signal_quality.dynamic_noise_level<=0.428141 && signal_quality.
    force_balance=0 && capture.illumination_level <= 0.12151)
```

Figure 4.7: Another constraint extracted from our case study

		Oracle	
		Faulty	Non-faulty
variability model (VM')	Faulty (invalid)	234 ($\pm$57.899)	69.5 ($\pm$26.973)
	Non-faulty (valid)	141.1 ($\pm$60.440)	3566.2 ($\pm$25.804)

Table 4.1: Confusion matrix of our experiment

False Positives (FP) are configurations "faulty"[2] in VM' whereas they are classified as non-faulty by the oracle. The ML approach has inferred too restrictive constraints that now uselessly forbid "non faulty" configurations.

False Negatives (FN) are configurations "non faulty" in VM' whereas they are classified as faulty by the oracle. The ML approach fails to infer constraints that could have forbidden "faulty" configurations.

Precision is the measurement assessing the number of correct classifications performed for a class (main diagonal) regarding the total number of classification made for this class (sum of a row). Over the 20 runs, the mean precision for the class "non-faulty" is : $P_{mean-non-faulty} = \frac{3566.2}{3566.2+141.1} \simeq 0.96$. Similarly, precision for the class "faulty" is $P_{mean-faulty} \simeq 0.77$.

The overall precision is the mean of the two values: $P_{overall} = \frac{0.96+0.77}{2} = 0.865$, *i.e.*, the classification will roughly perform well 9 times out of 10.

Recall is the measurement assessing the number of correct classification performed for a class regarding the total number of configurations declared by the oracle for this class (sum of a column). Similarly to the precision, recall can be computed for each class and then combined into an overall measure. This gives : $R_{mean-non-faulty} = \frac{3566.2}{3566.2+69.5} \simeq 0.98$; $R_{mean-faulty} \simeq 0.62$ and $R_{overall} = \frac{0.98+0.62}{2} = 0.80$. Here, recall is lower for the "faulty" class which gives us an idea of how difficult it is to understand the distribution of this class. This difficulty can come from the fact that there are fewer examples in the class "faulty" than in the class "non faulty".

RQ3) What are the strengths and weaknesses of our approach?

We now compare the properties of a sound approach and an ML approach in line with results of RQ1 and RQ2. We recall that a sound approach (see Definition 13) takes the output of the oracle and built constraints out of "faulty" configurations/videos. It means that constraints will be very specific to the configurations given to the oracle, involving every feature and attributes with their values.

Meaningfulness of constraints. A consequence is that constraints are typically difficult to read with a case-by-case extraction. A sound approach would have produced the conjunction of 53 constraints, each constraint being constituted by the number of feature/attribute' values of a configuration. As a configuration corresponds to 80+ values in our case, experts

[2]Strictly speaking, configurations are invalid (resp. valid) in VM'. We use the term "faulty" (resp. "non-faulty") for keeping an unified terminology with the oracle.

would have severe difficulties to review and understand what are the precise features and attributes involved in the fault. Our proposed approach computed 5 constraints on average with only a few features/attributes. This drastic reduction in size helps a video expert to better understand the constraints.

One can argue that techniques for reducing constraints (*e.g.*, [263]) can be adapted to numerical values and perhaps improve a sound approach. However, it is unlikely since the configurations do not necessarily share common values, especially in continuous domains. In fact, there is a more fundamental issue: constraints of a sound approach are so precise they cannot be able to capture other faulty configurations even in their close neighborhood. Hence, the value of an ML approach resides in the ability to produce more general and thus meaningful constraints. The fact that constraints are general comes from the way ML algorithms are designed. They try to infer properties out of few examples resulting in an approximation of the real behaviors of data. This approximation can be seen as a convex hull in the space of configuration. The added value of ML algorithms is to allow asperities in the hull to reduce the number of classification errors that could be made by a simple convex hull approximation algorithm.

Precision and recall. The strict strategy of a sound approach has another practical implication. In our case, the sound approach would only remove 53 configurations out of 500 (no more no less). On the other hand, our ML method removes 234 more faulty configurations than a sound approach (see Results in Table 4.1). Indeed, when validating our classification models with 4000 new configurations, the ML approach was able to recognize 234 (as a mean over 20 runs according to Table 4.1) configurations as faulty – without having to produce and test the video variants. The sound approach was unable to detect them because these 234 configurations were simply not in the original sample. Hence, the prediction of faulty configurations with ML gives a factor improvement of $(48 + 234)/53 = 5,3$. (The number of videos classified as faulty during the learning process is 48 since we obtained 90.56% of classification performance. 234 corresponds to the average number of videos classified as faulty while validating the built decision tree on 4000 new configurations. Finally, 53 is the number of faulty videos in the sampling and thus classified as such by the sound approach.)

In order to scale (*i.e.*, capture a similar set of faulty configurations than an ML approach), a sound approach has to sample a significant number of additional configurations. In our case study, there are two major drawbacks. First, covering a large percentage of the configuration space is simply not possible, mainly due to numerical values. Second, the cost of testing a configuration is very high: half an hour to generate a video and a few seconds to process it with the oracle. Concretely, the cost in time and resources for 4000 configurations is $4000 * 30 = 12000 \approx 2000$ hours for one machine. Hence, the use of a sound approach can be very costly since we envision to generate even more videos in the future. Overall, the major strength of ML resides in its ability to reduce the testing cost through the prediction of faulty configurations.

The prediction capability of ML is also a weakness since it induces some errors. In our case, out of 4000 (see Table 4.1), in average 141 configurations were classified as non-faulty by ML (despite being actually faulty). A sound approach is also unable to forbid such configurations in the first place since they are not part of the sample. That is, a sound approach would have suffered from similar issues. Finally 69.5 configurations out of 4000 were, in average, classified by ML as faulty (despite being actually non-faulty). In this case, a sound approach would have kept these configurations and, thus, is superior to a ML technique.

As a summary there is a trade-off to find. On the one hand, the ability of ML to be one step ahead can reduce testing effort, produce meaningful constraints, and forbid more faulty configurations (as in our case study). On the other hand, a sound, conservative approach (see Definition 13, page 112) can be of interest in cases software practitioners do not want to unnecessarily lose some configurations.

replication

Data, oracle, scripts about the MOTIV experiments are available: https://github.com/learningconstraints/SPLC-16

4.1.2 Learning contextual variability models

Modelling how contextual factors relate to the configuration space of a software system is most of the time a manual and error-prone task, highly dependent on expert knowledge. Machine learning techniques have the potential to automatically predict what are the acceptable software configurations of a given context. The key idea is to execute and observe a sample of software configurations within a sample of contexts, and then learn what factors of the context are likely to discard or activate some features of the software. As a result, software developers and product managers can automatically extract the rules that specialize highly-configurable systems for operating on specific contexts.

The approach is quite similar to the method described in previous Section 4.1.1, except that features reside at different levels of abstractions and are located in different feature models. The learning method comes in complement to the modelling effort and supports separation of concerns in variability modelling (as envisioned in Chapter 2): constraints can be injected in the different models.

We consider that the context of a software system is itself a configurable entity. It is constituted of different concerns: execution environment (hardware, operating system, *etc.*), kinds of inputs to process, goals and performance to meet (execution time, quality of the result, *etc.*). Other factors such as country regulations or marketing strategies can also be part of this context and have an (indirect) influence on the software [20, 139, 140]. From this mapping, we expect that given a context configuration, there is at least one corresponding software configuration. Conversely we also want that for each possible software configuration, there is at least one corresponding context configuration. If that is not the case, we could prevent this particular software configuration to be selectable from the beginning through constraints.

Let us take the example of a tracking vision system build on top of OpenCV (see `http://opencv.org/`). A variability model is depicted in Figure 4.8. On the left-hand side, context configurations involve features such as Camera, Video, Position, Light Source, *etc.* Some features are mandatory (*e.g.*, Camera or Video), optional (specific climate conditions), mutually exclusive (Fog or Heat haze), and some values are numeric (Vibrations or Noise level). On the right-hand side, we model (a subset of) OpenCV software variability. For example, Confidence is a parameter of the Detect function. It can be tuned for internally influencing the results of subsequent computations. In the example of Figure 4.8, the two variability models are separated and aggregated as in Chapter 2 since several contexts are involved.

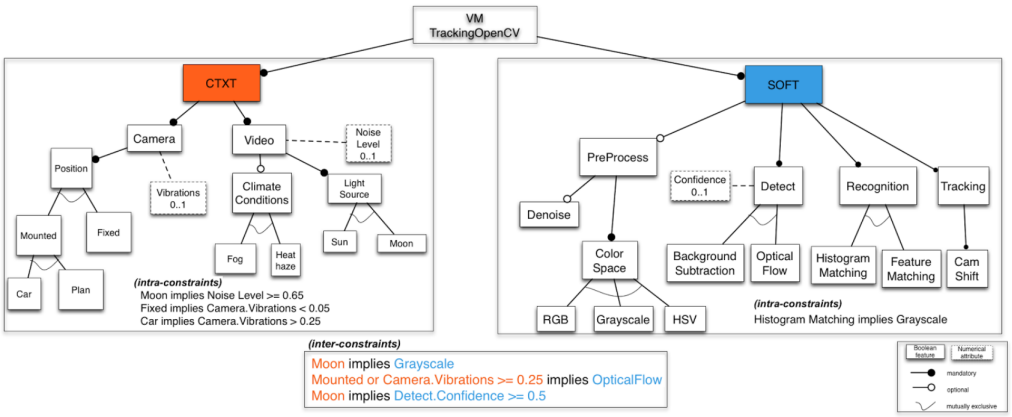


Figure 4.8: modelling context and software configurations with intra-constraints inside software (or inside context), and inter-constraints between the context and software. The learning method is capable to discover such inter or intra-constraints

Specific constraints are usually specified for relating features of the context to features of the software. Figure 4.8 presents possible examples of such inter-constraints in the lower part. Due to the huge number of configurations and complex relations between context and software, it is easy to forget or to wrongly specify a constraint, with possible adverse consequences when the software is deployed. We aim to address the problem of "How to elaborate a variability model (including constraints) that can be configured for different contexts?" with a learning method for capturing constraints. This approach aims to complement the manual effort of an expert or software developer.

Learning intra and inter-constraints

Figure 4.9 introduces the learning process. All steps can be automated except the user specification of what is an acceptable configuration (typically below or above a performance threshold value). First, we generate a sample of N configurations of the variability model. A configuration is composed of contextual features and software features. Numerous sampling strategies (*e.g.*,, random, t-wise) can be used to automatically select valid configurations (see Section 2.3). The number of configurations N (budget) in the sample can be controlled by the user.

Second, we execute and observe each software configuration within the context configuration. In our running example, a configuration of the tracking vision system is used to process a video with specific characteristics (see config1, config2, ..., configN in Figure 4.9). Many properties of a software configuration can be measured: whether it crashes at runtime, whether it violates some invariants, or whether it meets a certain performance goal.

In the case of vision systems, we are mostly interested in measuring the accuracy and precision of the system to track objects, as well as the execution time. We obtain a matrix with N configurations (see Figure 4.9). Third, the user of our learning method specifies the threshold values for which the performance is considered as acceptable. Performance

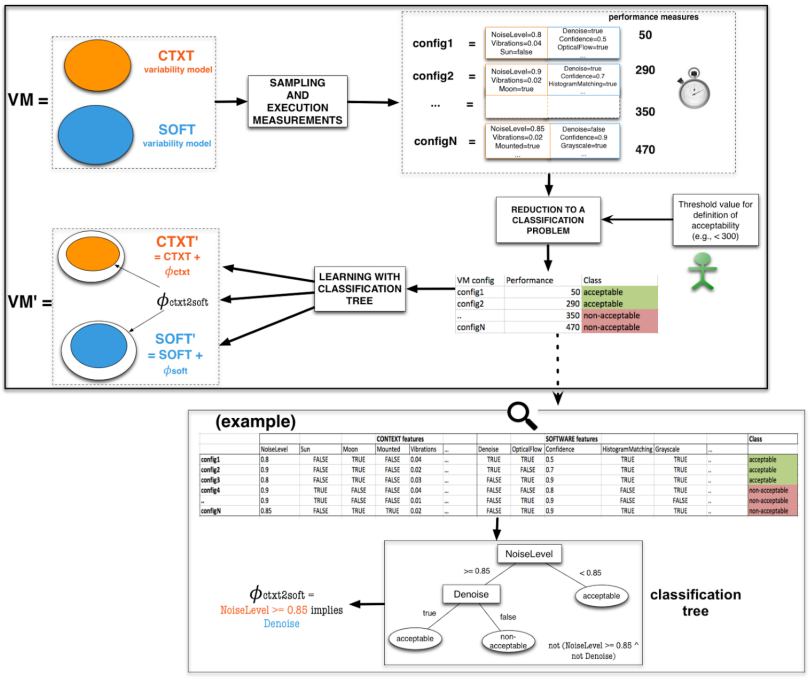


Figure 4.9: From a sample of configurations and measurements, we build a classification tree out of which we can extract constraints (*e.g.*, see inter-constraint given at the bottom)

measurements are compared to the threshold value and we can classify configurations as acceptable or non-acceptable. Finally, we address a statistical binary classification[3] problem in which we can predict which context and software features (part of the configurations) lead to acceptable or non-acceptable configurations.

At the bottom of Figure 4.9, we give an example of a decision tree and an extracted constraint. Nodes test for the values of a certain feature (*e.g.*, NoiseLevel). Edges correspond to the outcome of a test and connect to the next node (*e.g.*, >= 0.85). Leaves of the tree predict the final outcome ("acceptable" or "non-acceptable"). We follow the paths leading to "non-acceptable", build the conjunction of all decisions $NoiseLevel \geq 0.85 \wedge Denoise == false$, and negate the expression.

Results

Feasibility study. We have developed a tracking vision system by strictly following the approach illustrated throughout this subsection. Specifically, we modeled and implemented software variability in C++ using a subset of OpenCV. To provide data appropriate to each context configuration,[4] we synthesized videos with various properties like the noise level, the camera vibration, *etc.* We used the MOTIV video generator presented in Section 4.1.1

[3] As recently shown in [209], it is also possible to resolve a regression problem (the prediction of a performance, quantitative value) and then come down to classification problem (the prediction of whether the performance value is acceptable).

[4] It is also possible to use realistic benchmarks as we did in the context of multimorphic testing [296].

that synthesizes video variants together with their expected results (ground truths). In this way, we can measure the performance (precision and accuracy of the tracking of objects of interests, execution time, *etc.*) of tracking configurations in different contexts. Based on measurements, we have been able to specialize the configuration spaces and learn non-trivial constraints, similar to those in Figure 4.8.

Revisiting the MOTIV case study. We have specialized the industrial video generator introduced in Section 4.1.1. Our learning approach allows one to constrain the generator and only build video variants of a certain *quality* (size, noise frequency, *etc.*). For example, the video generator can synthesize only videos that are less than 500Mb. Hence, we generalize our previous attempt: users simply have to specify their performance objectives (thresholds) in order to get different variants of the video generator – ready for specific usages and contexts.

Controlled experiments. We have considered 10 publicly available configurable systems (see Table 4.2) for which we have reused performance measurements (execution time, footprint, *etc.*) of configurations using benchmarks [131, 310, 311]. We used our learning method to synthesize constraints in such a way we only retain software configurations meeting a certain performance objective. A practical application is that we can specialize configurable systems for targeting specific usages, customers, deployment scenarios, hardware settings, in short any contexts. For example, we can specialize the variability model of the x264 video encoder for achieving a low energy consumption (*e.g.*, for embedding x264 in resource-constrained devices). Empirical results are promising for two reasons. First, the learning phase can reach an accuracy greater than 80% on average for all performance thresholds and for all systems. We can thus narrow the space of possible configurations to a good approximation. Second, only a relatively small training set is needed to achieve a high classification accuracy (see Table 4.2).

Overall, a reasonable number of measurements (from dozens to hundreds) can be sufficient to discover complex relationships that rightly discard a large portion of non-acceptable configurations, without over-constraining the configurable systems. For more details, we invite the reader to consult our technical report [297].

replication

The source code of the scripts, experimental data, results and visualisations (*e.g.*, heatmaps) are available at http://learningconstraints.github.io.

4.2 Adversarial learning for variability

The content of this section is adapted from the following publication:
P. Temple, G. Perrouin, M. Acher, B. Biggio, J.-M. Jézéquel and F. Roli, 'Empirical Assessment of Generating Adversarial Configurations for Software Product Lines', *Empirical Software Engineering (ESE)*, Nov. 2020

> "Testers don't like to break things; they like to dispel the illusion that things work." [174]

System	**Domain**	**Language**	**Variability (number of options)**	**Number of valid configurations**	**Performance objectives**	**Number of measurements needed to reach 80% accuracy (on average, for all perf. thresholds)**	**Corresponding % of measurements (wrt number of valid configs) needed to reach 80% accuracy**
MOTIV video generator	Video Processing	Lua	20 Boolean 88 numerical	~10^100	noise, size	500	10^-98
Apache Web server	Web server	C	9 Boolean	192	response rate	14	7.29
BerkeleyC	Database	C	18 Boolean	2560	I/O time	26	1.02
BerkeleyJ	Database	Java	26 Boolean	400	I/O time	9	2.25
LLVM	Compiler	C++	11 Boolean	1024	optimization time	31	3.03
SQLite	Database	SQLite	39 Boolean	10^6	time	901	0.1
Dune	Solver	C++	8 Boolean 3 numerical	2304	solving time	24	1.04
HIPAcc	Image Proc.	C++	31 Boolean 2 numerical	13485	solving time	135	1
HSMGP	Solver	N/A	11 Boolean 3 numerical	3456	time	35	1.01
JavaGC	Runtime Env.	C++	12 Boolean 23 numerical	10^31	time	1670	10^-28
x264	Video Processing	C	8 Boolean 12 numerical	10^27	energy, speed, size, time, watt	691	10^-25

Table 4.2: Experimental results across subject systems and application domains

A long-standing issue for developers and product managers of an SPL is to gain confidence that all possible products are functionally viable, *e.g.*, all software-intensive products compile and run. This is a hard problem since modern software systems can involve thousands of features inducing a combinatorial explosion of the number of possible products. For example, in the case study of Section 4.1.1 (the MOTIV video generator), the estimated number of configurations is 10^{314} while the derivation of a single product out of a configuration takes 30 minutes on average. At this scale, practitioners cannot test all possible configurations and the corresponding products' qualities. To handle this issue, a promising approach is to sample a number of configurations and predict the quantitative or qualitative properties of the remaining products using Machine Learning (ML) techniques (see Section 4.1 and Section 4.3).

However, we need to trust the ML classifier [42, 229] of an SPL in avoiding misclassifications and costly derivations of non-acceptable products. ML researchers demonstrated that some forged data, called *adversarial*, can fool a given classifier [66]. *Adversarial machine learning* (advML) thus refers to techniques designed to fool (*e.g.*, [229, 61, 62]), evaluate the security (*e.g.*, [64]) and even improve the quality of learned classifiers [127]. Even though results are promising in different contexts, the ML community did not apply advML techniques in the SPL domain. On the other hand, numerous techniques have been developed to test or learn software configuration spaces of SPLs, but none of them considered advML [29]. A strength of advML is that generated adversarial configurations are crafted to force an ML classifier to make errors, by either exploiting its intrinsic properties or its insufficient training. Furthermore, since advML operates on the classifier, there is no need to derive and test additional products of an SPL.

Rationale of adversarial learning for variability

The main idea of this contribution is to shift ideas and techniques from advML to the engineering of SPLs or configurable systems. Specifically, the principle is to generate adversarial configurations with the intent of fooling and improving ML classifiers of SPLs. Adversarial configurations can pinpoint cases for which non-acceptable products of an SPL can still be derived since the ML classifier is fooled and misclassifies them. Such configurations are symptomatic of issues stemming from various sources:

- the variability model (*e.g.*, constraints are missing to avoid some combinations of features);
- the variability implementation (*e.g.*, interactions between features cause bugs);
- the testing environment (*e.g.*, some products are wrongly tested and should not be considered as acceptable);
- or simply the fact that, based on previous observations, configurations are predicted to meet (non-)functional requirements while they actually fail to do so, asking to be fixed.

In SPL engineering, ML brings the benefit of partitioning the configuration space based on a (small) number of assessed variants, which is faster than running the oracle on every single variant. However, this gain comes at the risk of approximations made by the statistical ML classifier. That is, the ML classifier can still make prediction errors when classifying configurations. In a sense, an ML classifier is an *approximate configuration oracle*.[5]

Our idea is to "attack" the ML classifier through the generation of so-called *adversarial configurations* able to fool it. The objective is to synthesize configurations for which the ML classifier performs an inaccurate classification. Such adversarial configurations have a value *per se*. Developers can *debug* and eventually improve the system (including the tests) based on insights brought by adversarial configurations. Another application is to augment the original training set with adversarial configurations: the hope is to obtain a more accurate ML classifier since problematic cases have been included as part of the training phase.

Adversarial learning and evasion attacks

In this part, we detail the foundations, algorithms, and processes to generate adversarial configurations.

AdvML and evasion attacks. According to Biggio *et al.* [66], deliberately attacking an ML classifier with crafted malicious inputs was proposed in 2004. Today, it is called adversarial machine learning and can be seen as a sub-discipline of machine learning. Depending on the attackers' access to various aspects of the ML system (*e.g.*, access to the data sets or ability to update the training set) and their goals, various kinds of attacks are available [61, 62, 64, 65, 63]. A categorization of such adversarial attacks can be found in [42, 66]. In this contribution, we focus on *evasion attacks*: these attacks move labeled data to the other side of the separation (putting it in the opposite class) via successive modifications of features' values. Since areas close to the separation are of low confidence, such adversarial configurations can have a significant impact if added to the training set. To determine in which direction to move the data such that it reaches the separation, a gradient-based method has been proposed by Biggio *et al.* [62]. This method requires the attacked ML algorithm to be differentiable (*e.g.*, algorithms building models for which the classification decision is based on a confidence metric which is not binary; this is the case for SVMs or Bayesian predictors which compute a likelihood to belong to a class). One of such differentiable classifiers is the Support Vector Machine (SVM), parameterizable with a kernel function.[6]

A dedicated evasion algorithm. For generating adversarial configurations, we develop an adaptation of Biggio *et al.*'s evasion attack [62] (see Algorithm 1). First, we select an initial, random configuration x^0 to be moved labeled with the class from which the attack starts. Then, we set the step size (t), a parameter controlling the convergence of the algorithm. Large steps induce difficulties to converge, while small steps may trap the algorithm in a local optimum. While the original algorithm introduced a termination criterion based on the impact of the attack on the classifier between each move (if this impact was smaller than a threshold ϵ, the algorithm stopped; assuming an optimal attack), we chose to set a maximal number of displacements nb_disp in advance and let the technique run until the end. This allows for a controllable computation budget, as we observed that for small step

[5] I introduce this terminology for the first time in this manuscript. A configuration oracle is a procedure capable of determining the property of any configuration. The oracle can be an approximation as opposed to a perfect oracle that would typically necessitate the costly execution of individual configuration

[6] The most common functions are linear, radial-based functions and polynomial.

Algorithm 1 A dedicated algorithm conducting the gradient-descent evasion attack inspired by [62]

Input: x^0, the initial configuration; t, the step size; *nb_disp*, the number of displacements; g, the discriminant function
Output: x^*, the final attack point

(1) m = 0;
(2) Set x^0 to a copy of a configuration of the class from which the attack starts;
while $m < nb\_disp$ **do**
  (3) m = m+1;
  (4) Let $\nabla F(x^{m-1})$ a unit vector, normalisation of $\nabla g(x^{m-1})$;
  (5) $x^m = x^{m-1} - t\nabla F(x^{m-1})$;
end while
(6) return $x^* = x^m$;

sizes the number of displacements required to meet the termination criterion was too large. We assume that there is a differentiable function $g : X \mapsto \mathbb{R}$ that maps a configuration to a real number. It is the discriminant function[7] and is defined by the ML algorithm being used. Only the sign of g is used to assign a label to a configuration x. Thus, $f : X \mapsto Y$ can be decomposed in two successive functions: first $g : X \mapsto \mathbb{R}$ that maps a configuration to a real value and then $h : \mathbb{R} \mapsto Y$ with $h = sign(g)$. However, $|g(x)|$ (the absolute value of g) intuitively reflects the confidence the classifier has in its assignment of x. $|g(x)|$ increases when x is far from the separation and surrounded by other configurations from the same class and is smaller when x is close to the separation.

Concretely, the core of the algorithm consists of a *while* loop that iterates over the number of displacements. Statement (4) determines the direction towards the area of maximum impact with respect to the classifier (explaining why only a unit vector is needed). $\nabla g(x^{m-1})$ is the slope of the gradient of $g(x^{m-1})$. Since evasion attacks is a technique based on gradient descent, the direction of interest towards which the adversarial configuration should move is the opposite of this value. This vector is then multiplied by the step size t and subtracted to the previous move. The final position is returned after the number of displacements has been reached. For statements (4) and (5) we simplified the initial algorithm [62]: we do not try to mimic as much as possible existing configurations as we look forward to some diversity.

In SPLs, the feasible region is given by valid configurations (defined by, among others, allowed features' combinations). However, being able to state all cross-tree constraints and potential domain values remain difficult. This task is nonetheless very important for the adversarial attack algorithm. In this work, we opted for a quite simplistic way of handling constraints. We only took care of the type of features and attribute values (natural integers, floats, Boolean). For example, if a constraint forbids a value to go below zero but a displacement tries to do so, we reset to zero this value (since it is the lower bound that this value can take); a similar principle is done for Boolean values (that can take only values 0 or 1).

[7] The term discriminant function has been used by Biggio *et al.* [62] and should not be confused with the unrelated discriminator component of generative adversarial nets (GANs) by Goodfellow *et al.* [127]. In GANs, the discriminator is part of the "robustification process". It is an ML classifier striving to determine whether an input has been artificially produced by the other GANs' component, called the generator. Its responses are then exploited by the generator to produce increasingly realistic inputs. In this work, we only generate adversarial configurations, though GANs are envisioned as follow-up work.

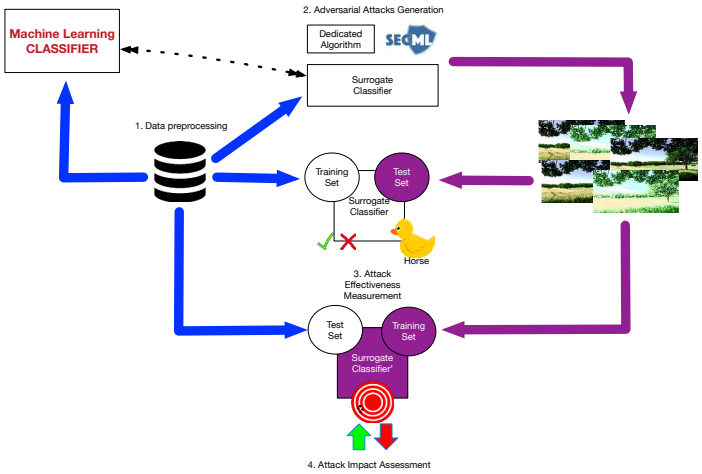


Figure 4.10: Adversarial Pipeline

Implementation. We rely on SecML[8] a Python library that has been developed by researchers from the Pattern Recognition and Applications Laboratory (PRALab), in Sardinia, Italy. This library provides different advML techniques and embeds utilities to create a customized pipeline according to the data to attack, their representations, the ML model that is used in the system to attack among other parameters.

Figure 4.10 presents a generic adversarial pipeline. The first step is to prepare the data that are shared by the original classifier and by the adversarial pipeline. Generally, an adversarial framework relies on a *surrogate classifier* that is learned from the same data when the attacker does not have access to the target classifier or when the attack cannot be conducted directly. Since there is evidence that attacks conducted on a specific ML model can be transferred to others [108, 107, 71], using a surrogate classifier is a legitimate approach in a black-box scenario.

Once the classifier is learned, we can use our dedicated and SecML algorithms to generate attacks in a second step. The third step evaluates the effectiveness of generated adversarial configurations forming the test set. In particular, we check the validity of generated adversarial configurations and their ability to be misclassfied. Finally, the fourth step learns a new classifier with an augmented training set composed of the original training set and some adversarial configurations.

Evaluation

We apply the adversarial generator within two contexts: JHipster and MOTIV. For each case, we address the following research questions: **RQ1**: *How effective is our adversarial generator to synthesize adversarial configurations?* Effectiveness is measured through the capability of our evasion attack algorithm to generate misclassified configurations:

- **RQ1.1**: Can we generate adversarial configurations that are wrongly classified?
- **RQ1.2**: Are all generated adversarial configurations valid *w.r.t.* constraints in the VM?

[8] https://secml.gitlab.io/index.html

- **RQ1.3**: Is using the evasion algorithm more effective than generating adversarial configurations with random modifications?

- **RQ1.4**: Are attacks effective regardless of the targeted class?

RQ2: *What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?*

Specific details about the data, parameterization of the algorithms, *etc.* are available in [300]. I briefly explain the interest of adversarial learning and provide key results hereafter.

MOTIV. First, we consider the industrial video generator (MOTIV) introduced in Section 4.1.1. A highly challenging problem is to identify and even prevent MOTIV configurations that lead to non-acceptable products (videos).

AdvML to the rescue. An ML classifier can make errors, preventing acceptable videos (false positives) or allowing non-acceptable videos (false negatives). Most of these errors can be attributed to the confidence of the classifier coming from both its design (*i.e.*, the set of approximations used to build its decision model) and the training set (and more specifically the distribution of the classes). Areas of low confidence exist if configurations are very dissimilar to those already seen or at the frontier between two classes. In the remainder, we investigate the use of advML to quantify these errors and their impact on the MOTIV SPL and ML classifier.

Key results (RQ1, MOTIV). Our generated adversarial attacks are: 100% effective (always misclassified, RQ1.1), do not depend on the target class (RQ1.4), and yield valid configurations (RQ1.2) if parameterized properly. In contrast, our random baseline was only able to achieve 62.5% of effectiveness at best (RQ1.3). The balance in the data sets does not affect these results and the targeted class affects show the same trends despite small differences (RQ1.4).

Key results (RQ2, MOTIV). When data sets are balanced, configurations generated by evasion attacks can be used and added to the training set to improve the prediction performances of the classifier. This step requires careful empirical tuning. Overall, with only 25 configurations added, we can improve classifier accuracy by up to 3%.

JHipster is an open-source generator for developing Web applications [170]. We introduce the case early in the manuscript, see Section 2.3.

AdvML to the rescue. Instead of deriving all JHipster variants, one can use ML and only a sample of configurations to eventually prevent non-acceptable variants and avoid a costly build. As an outcome, we can identify features of JHipster that cause non-acceptable variants (*i.e.*, build failures) and re-inject this knowledge into the feature model. Build failures can occur in various circumstances such as: (1) implementation bugs in the artefacts, typically due to a dependency wrongly specified in a Maven file or due to unsafe interactions between features in the Java source code; (2) un-properly building environments in which some packages or tools are incidentally missing because some combinations of features were not assessed before. Once the learning process is realized, the question arises as to the quality of the ML classifier and the whole JHipster SPL. Again, we can apply advML.

Key results (RQ1, JHipster). Similar as MOTIV, the two implementations of the evasion attack are able to generate configurations that are systematically misclassified (after tuning the parameters of the implementations) in the context of JHipster.

Key results (RQ2, JHipster). Most of our attempts to improve the performance of the classifier failed since the accuracy remained the same as the baseline accuracy (*i.e.*, without adversarial configurations). This may be due to the low number of adversarial configurations that we added to the training set or it may be the nature of the classification problem that is too easy and achieving performance improvements is more challenging since accuracy is close to perfect in the first place.

replication

Scripts and data are available: https://github.com/templep/EMSE_2020

4.3 Learning variability performance

The content of this section is adapted from the following publication:
J. Alves Pereira, M. Acher, H. Martin and J.-M. Jézéquel, 'Sampling Effect on Performance Prediction of Configurable Systems: A Case Study', in *International Conference on Performance Engineering (ICPE 2020)*, 2020. https://hal.inria.fr/hal-02356290

Options often have a significant influence on performance properties that are hard to know and model *a priori*. There are numerous possible option values, logical constraints between options, and subtle interactions among options [29, 273, 131, 287, 173] that can have an effect while *quantitative* properties such as execution time are themselves challenging to comprehend.

Measuring all configurations of a configurable system is the most obvious path to *e.g.*, find a well-suited configuration *w.r.t.* performance, but is too costly or infeasible in practice. Machine-learning techniques address this issue by measuring only a subset of configurations (known as a sample). Then these configuration measurements are used to build a *performance model* capable of predicting the performance of other configurations (*i.e.*, configurations not measured before).

Contrary to Section 4.1, the focus is on predicting configurations properties that are quantitative, numerical, continuous (*e.g.*, execution time in seconds) as opposed to qualitative, categorical (*e.g.*, "acceptable" or "non-acceptable"). Hence, the statistical learning problems enter in the family of regression problems as opposed to classification ones. Though problems differ, the learning process follows a similar scheme "sampling, measuring, learning" [29].

A crucial step is the way the sampling is realized, since it can drastically affect the performance model accuracy [29, 173]. Ideally, the sample should be small to reduce the cost of measurements and representative of the configuration space to reduce prediction errors. The sampling phase involves a number of difficult activities: *(1)* picking configurations that are valid and conform to constraints among options – one needs to resolve a satisfiability problem; *(2)* instrumenting the executions and observations of software for a variety of configurations – it might have a high computational cost especially when measuring performance

aspects of software; *(3)* guaranteeing a coverage of the configuration space to obtain a representative sample set. An ideal coverage includes all influential configuration options by covering different kinds of interactions relevant to performance. Otherwise, the learning may hardly generalize to the whole configuration space.

With the promise to select a small and representative sample set of valid configurations, several sampling strategies have been devised in the last years (see our technical report [29] for the specific topic of learning but also Section 2.3). For example, random sampling aims to cover the configuration space uniformly while coverage-oriented sampling selects the sample set according to a coverage criterion (*e.g.*, t-wise sampling to cover all combinations of t selected options). Kaltenecker *et al.* [173] analyzed 10 popular real-world software systems and found that their novel proposed sampling strategy, called diversified distance-based sampling, dominates five other sampling strategies by decreasing the cost of labelling software configurations while minimizing the prediction errors.

In this line of work, we conduct a replication of this preliminary study by exclusively considering the x264 case, a configurable video encoder. Though we only consider one particular configurable system, we make vary its workloads (with the use of 17 input videos) and measured two performance properties (encoding time and encoding size) over 1,152 configurations. Interestingly, Kaltenecker *et al.* [173] also analyzed the same 1,152 configurations of x264, but a fixed input video was used and only the time was considered. The goal of our experiments is to determine whether sampling strategies considered in [173] over different subject systems are as effective on the *same* configurable system, but with different factors possibly influencing the distribution of the configuration space. A hypothesis is that practitioners of a configurable system can rely on a one-size-fits-all sampling strategy that is cost-effective whatever the factors influencing the distribution of its configuration space. On the contrary, another hypothesis is that practitioners should change their sampling strategy each time an input (here: videos) or a performance property are targeted.

We investigate to what extent sampling strategies are sensitive to different workloads of the x264 configurable system and different performance properties: What are the most effective sampling strategies? Is random sampling a strong baseline? Is there a dominant sampling strategy that practitioners can always rely on? To this end, we systematically report the influence of sampling strategies on the accuracy of performance predictions and on the robustness of prediction accuracy.

Design Study

Figure 4.11 summarizes our design study, including research questions, subject systems, sampling strategies, and metrics.

Research Questions. We conducted a series of experiments to evaluate six sampling strategies and to compare our results to the original results in [173]. We aim at answering the following two research questions:

- **(RQ1)** What is the influence of using different sampling strategies on the accuracy of performance predictions over different inputs and non-functional properties?
- **(RQ2)** What is the influence of randomness of using different sampling strategies on the robustness of prediction accuracy?

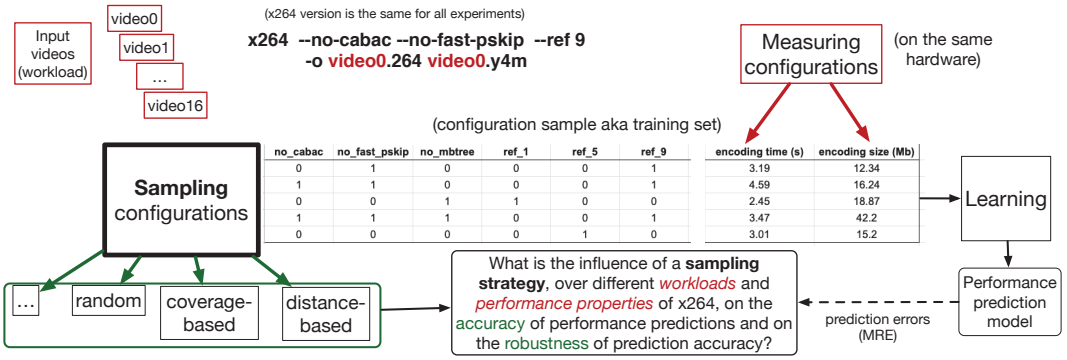


Figure 4.11: Design study: sampling effect on performance predictions of x264 configurations

It is not new the claim that the prediction accuracy of machine learning extensively depends on the sampling strategy. The originality of the research question is to what extent are performance prediction models of the *same* configurable system (here: x264) sensitive to other factors, such as different inputs and non-functional properties. To address *RQ1*, we analyze the sensitivity of the prediction accuracy of sampling strategies to these factors. Since most of the considered sampling strategies use randomness, which may considerably affect the prediction accuracy, *RQ2* quantitatively compares whether the variances (over 100 runs) on prediction accuracy between different sampling strategies and sample sizes differ significantly. We show that the sampling prediction accuracy and robustness hardly depends on the definition of performance (*i.e.*, encoding time or encoding size). As in Kaltenecker *et al.* [173], we have excluded t-wise sampling from *RQ2*, as it is also deterministic in our setting and does not lead to variations.

Subject System. We conduct an in-depth study of x264, a popular and highly configurable video encoder implemented in C. We choose x264 instead of the other case studies documented in Kaltenecker *et al.* [173] because x264 demonstrated more promising accuracy results to the newest proposed sampling approach (*i.e.*, diversified distance-based sampling). With this study, we aim at investigating, for instance, whether diversified distance-based sampling also dominates across different variations of x264 (*i.e.*, inputs, performance properties). As benchmark, we encoded 17 different input videos from raw YUV to the H.264 codec and measured two quantitative properties (encoding time and encoding size).

- Encoding time (in short, **time**): how many seconds x264 takes to encode a video.
- Encoding size of the output video (in short, **size**): compression size (in bytes) of an output video in the H.264 format.

All measurements have been performed over the same version of x264 and on a grid computing infrastructure called IGRIDA.[9] Importantly, we used the same hardware characteristics for all performance measurements. We consider 17 input videos. This number of inputs allows us to draw conclusions about the practicality of sampling strategies for diversified conditions of x264. To control measurement bias while measuring execution time, we have repeated the measurements several times [28].

[9] http://igrida.gforge.inria.fr/

Sampling strategies. Several sampling strategies have been proposed in the literature about software product lines and configurable systems [29, 302, 315]. We now present an overview of six sampling strategies also considered in [173] and used in our study. All strategies have the merit of being agnostic of the domain (no specific knowledge or prior analysis are needed) and are directly applicable to any configurable system.

Random sampling aims to cover the configuration space *uniformly*. Throughout this section, we refer to random as uniform random sampling. The challenge is to select one configuration amongst all the *valid* ones in such a way each configuration receives an equal probability to be included in the sample. An obvious solution is to enumerate all valid configurations and randomly pick a sample from the whole population. However, enumerative approaches quickly do not scale with a large number of configurations. Oh *et al.* [234] rely on binary decision diagrams to compactly represent a configuration space, which may not scale for very large systems [215]. Another line of research is to rely on satisfiability (SAT) solvers. For instance, UniGen [79, 78] uses a hashing-based function to synthesize samples in a nearly uniform manner with strong theoretical guarantees. These theoretical properties come at a cost: the hashing-based approach requires adding large clauses to formulas. In Section 2.3.2, we show that state-of-the-art algorithms are either not able to produce any sample or unable to generate uniform samples for the SAT instances considered. Overall, a true uniform random sampling may be hard to realize, especially for large configurable systems. At the scale of the x264 study [173], though, uniform sampling is possible (the whole population is 1,152 configurations). The specific question we explore here is whether random is effective for learning (in case it is applicable as in x264).

When random sampling is not applicable, several alternate techniques have been proposed typically by sacrificing some uniformity for a substantial increase in performance.

Solver-based. Many works rely on off-the-shelf constraint solver, such as SAT4J [197] or Z3 [105], for sampling. For instance, a random seed can be set to the Z3 solver and internally influences the variable selection heuristics, which can have an effect on the exploration of valid configurations. Henard *et al.* noticed that solvers' internal order yields non-uniform (and predictable) exploration of the configuration space [144]. Hence, these strategies do not guarantee true randomness as in uniform random sampling. Often the sample set consists of a locally clustered set of configurations.

Randomized solver-based. To weaken the locality drawback of solver-based sampling, Henard *et al.* change the order of variables and constraints at each solver run. This strategy, called *randomized solver-based sampling* in Kaltenecker *et al.* [173], increases diversity of configurations. Though it cannot give any guarantees about randomness, the diversity may help to capture important interactions between options for performance prediction.

Coverage-based sampling aims to optimize the sample with regards to a coverage criterion. Many criteria can be considered such as statement coverage that requires the analysis of the source code. In this section and as in Kaltenecker *et al.* [173], we rely on t-wise sampling [87, 172, 193]. This sampling strategy selects configurations to cover all combinations of t selected options. For instance, pair-wise (t=2) sampling covers all pairwise combinations of options being selected. There are different methods to compute t-wise sampling. As in [173], we rely on the implementation of Siegmund *et al.* [288].

Distance-based. Kaltenecker *et al.* [173] propose distance-based sampling. The idea is to cover the configuration space by selecting configurations according to a given probability distribution (typically a uniform distribution) and a distance metric. The underlying benefit is that distance-based sampling can better scale compared to an enumerative-based random sampling, while the generated samples are close to those obtained with a uniform random.

Diversified distance-based sampling is a variant of distance-based sampling [173]. The principle is to increase diversity of the sample set by iteratively adding configurations that contain the least frequently selected options. The intended benefit is to avoid missing the inclusion of some (important) options in the process.

Experiment Setup. In our experiments, the independent variables are the choice of the input videos, the predicted non-functional-property, the sample strategies and the sample sizes.

For comparison, we used the same experiment design as in Kaltenecker *et al.* [173]. To evaluate the accuracy of different sampling strategies over different inputs and non-functional-properties, we conducted experiments using three different sample sizes. To be able to use the same sample sizes for all sampling strategies, we consider the sizes from t-wise sampling with t=1, t=2, and t=3. (Recall that t-wise sampling covers all combinations of t configuration options being selected.) We learn a performance model based on the sample sets along with the corresponding performance measurements defined by the different sampling strategies. Several machine-learning techniques have been proposed in the literature with this purpose [29], such as linear regression [81, 287, 173, 167, 185, 317, 225, 268, 203], classification and regression trees (CART) [298, 299, 273, 167, 225, 227, 228, 309, 312, 320, 325, 191, 253, 329, 328, 293], and random forest [31, 309, 255, 41, 301]. In this study, we use step-wise multiple linear regression [287] as in Kaltenecker *et al.* [173]. According to Kaltenecker *et al.* [173], multiple linear regression is often as accurate as CART and random forests.

To calculate the prediction error rate, we use the resulting performance models to predict the performance of the entire dataset of valid configurations C. We calculate the error rate of a prediction model in terms of the *mean relative error* (MRE - Equation 4.1). MRE is used to estimate the accuracy between the exact measurements and the predicted one.

$$MRE = \frac{1}{|C|} \sum_{c \in C} \frac{|measured_c - predicted_c|}{measured_c} \tag{4.1}$$

Where C is the set of all valid configurations used as the validation set, and $measured_c$ and $predicted_c$ indicate the measured and predicted values of performance for configuration c with $c \in C$, respectively. The exact value of $measured_c$ is measured at runtime while running the configuration c, and the predicted values of $predicted_c$ is computed based on the model built with a sample of configurations t. To address *RQ1*, we computed the mean error rate for each input video and sample size. A lower error rate indicates a higher accuracy. Then, we use a Kruskal-Wallis test [192] and pair-wise one-sided Mann-Whitney U tests [206] to identify whether the error rate of two sampling strategies differs significantly ($p < 0.05$) [38]. In addition, we compute the effect size $\hat{A}_{12}$ [314] (small(>0.56), medium(>0.64), and large(>0.71)) to easily compare the error rates of two sampling strategies.

To address *RQ2*, we compute the variance across the error rates over 100 runs. A lower variance indicates higher robustness. First, we use Levene's test [201] to identify whether the variances of two sampling strategies differ significantly from each other. Then, for these sampling strategies, we perform a one-sided F-tests [292] to compare pair-wisely the variance between sampling strategies.

To reduce fluctuations in the values of dependent variables caused by randomness (*e.g.*, the random generation of input samples), we evaluated each combination of the independent variables 100 times. That is, for each input video, non-functional property, sampling strategy and sampling size, we instantiated our experimental settings and measured the values of all dependent variables 100 times with random seeds from 1 to 100.

Key results

Much more details about the data, results, and insights can be found in [28]. I report and discuss an excerpt of key results hereafter.

Observations. For the property encoding time, uniform random sampling yields the most accurate performance models. Diversified distance-based sampling produces good results when a very limited number of samples are considered (*i.e.*, t=1) and almost reaches the accuracy of random when the sample sizes increase. In terms of robustness, diversified distance-based sampling is more robust than the other sampling strategies.

For the property encoding size, random sampling and randomized solver-based sampling outperform all other sampling strategies for most of the input videos with sample sizes t=2 and t=3; and solver-based sampling outperforms for sample sizes t=1. Overall, random, randomized solver-based, solver-based, and diversified distance-based present good and similar accuracy for t=2 and t=3. Differently from our previous results (for time), there is not a clear winner. In terms of robustness, uniform random sampling is more robust than the other sampling strategies.

An important result is that the property of interest (encoding time or encoding size) as well as the inputs (videos) can drastically change the overall conclusions about the effectiveness of sampling strategies *w.r.t.* accuracy and robustness.

Answering RQ1 (accuracy) Is there a "dominant" sampling strategy for the x264 configurable system whatever inputs and targeted quantitative properties? For the property encoding time, there is a dominant sampling strategy (*i.e.*, uniform random sampling) as shown in Kaltenecker *et al.* [173], and thus the sampling can be reused whatever the input video is. For the property encoding size, although the results are similar around some sampling strategies, they differ in a noticeable way from encoding time and suggest a higher input sensitivity. Overall, random is the state-of-the-art sampling strategy but is not a dominant sampling strategy in all cases, *i.e.*, the ranking of dominance changes significantly given different inputs, properties and sample sizes. A possible hypothesis is that individual options and their interactions can be more or less influential depending on input videos, thus explaining the variations' effect of sampling over the accuracy. Our results pose a new challenge for researchers: Identifying what sampling strategy is the best given the possible factors influencing the configurations' performances of a system.

Answering RQ2 (robustness). We have quantitatively analyzed the effect of a sampling strategy over the prediction variance. Overall, random (for size) and diversified distance-based (for time and size) have higher robustness. We make the observation that uniform random sampling is not necessarily the best choice when robustness should be considered

(but it is for accuracy). In practical terms, practitioners may have to find the right balance between the two objectives. As a sweet-spot between accuracy and robustness, diversified distance-based sampling (for time), and either random or randomized solver-based sampling (for size) are the best candidates. We miss however an actionable metric that could take both accuracy and robustness into account.

Our recommendations for practitioners are as follows:

- uniform random sampling is a very strong baseline independent to the inputs and performance properties. In the absence of specific-knowledge, practitioners should rely on this dominant strategy for reaching high accuracy;
- in case uniform random sampling is computationally infeasible, distance-based sampling strategies are interesting alternatives;
- the use of other sampling strategies does not pay off in terms of prediction accuracy. When robustness is considered as important, uniform random sampling is not the best choice and here we recommend diversified distance-based sampling.

The impacts of our results on configuration and performance engineering research are as follows:

- as uniform random sampling is effective for learning performance prediction models, additional research effort is worth doing to make it scalable for large instances. Recent results (see Section Scalability and quality of uniform samplers) show some improvements, but the question is still open for very large systems (*e.g.*, Linux, see Section Transfer learning across variants and versions: the case of Linux);
- some sampling strategies are surprisingly effective for specific inputs and performance properties. Our insights suggest the existence of specific sampling strategies that could prioritize specific important (interactions between) options. An open issue is to discover them for any input or performance property;
- performance measurements with similar distributions may be grouped together to enable the search for dominants sampling strategies;
- beating random is possible but highly challenging in all situations;
- it is unclear how factors such as the version or the hardware influence the sensitivity of the sampling effectiveness (and how such influence differs from inputs and performance properties);
- we warn researchers that the effectiveness of sampling strategies for a given configurable system can be biased by the workload and the performance property used.

replication

We provide general instructions on how to reproduce the results of the study: https://github.com/jualvespereira/ICPE2020. We also provide the variability model and the measurements of each video input for encoding time and encoding size. Our awarded paper [28] got the ACM badges "Artifacts Available" and "Artifacts Evaluated – Reusable"

4.4 Transfer learning across variants and versions: the case of Linux kernel size

The content of this section is adapted from the following publication:
M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil and O. Barais, 'Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes', Inria Rennes - Bretagne Atlantique, Research Report, Oct. 2019. https://hal.inria.fr/hal-02314830
H. Martin, M. Acher, J. A. Pereira, L. Lesoil, J. Jézéquel and D. E. Khelladi, 'Transfer learning across variants and versions: The case of linux kernel size', *Transactions on Software Engineering (TSE)*, 2021

With now more than 15,000 configuration options, including more than 9,000 just for the x86 architecture, the Linux kernel is one of the most complex configurable open-source system ever developed. If all these options were binary and independent, that would indeed yield 2^{15000} possible variants of the kernel. Of course not all options are independent (leading to fewer possible variants), but some of them have tri-states values: yes, no, or module instead of simply boolean values (leading to more possible variants). The Linux kernel is mentioned in numerous papers about configurable systems and machine learning for motivating the problem and the underlying approach. However, only a few works truly explore such a huge configuration space. In this line of work, we take up the Linux challenge.

Specifically, we consider a quantitative, non-trivial property of kernels – binary size. The goal is to predict the binary size of any configuration of Linux without actually building it. Linux kernels are used in a wide variety of systems, ranging from embedded devices and cloud services to powerful supercomputers [305]. Many of these systems have strong requirements on the kernel size due to constraints such as limited memory or instant boot [158, 237]. As elaborated in [25], the effort of the Linux community to document options related to kernel binary size is highly valuable, but mostly relies on human expertise, which makes the maintenance of this knowledge quite challenging on the long run. Furthermore, numerous works have showed that quantifying the performance influence of each individual option is not meaningful in most cases [29, 273, 131, 287, 173]. That is, the performance influence of n options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option. As our empirical results will show, the Linux kernel binary size is not an exception: options such as `CONFIG_DEBUG_*` or `CC_OPTIMIZE_FOR_SIZE` have cross-cutting, non-linear effects and cannot be reduced to additive effects, hence basic linear regression models, which are unable to capture interactions among options, give poorly accurate results.

In the first part of this section, I am reporting the results of learning methods used in the state-of-the-art over Linux. The Linux case indeed questions whether learning methods used in the state of the art would scale (*w.r.t.* training time), provide accurate models, and interpretable information at such an unprecedented scale. We aim to predict a non-functional property (the binary size) of the Linux kernel by considering *all 9K+ options without a priori selection based on documentation or expert knowledge*. At this scale, applying machine learning to Linux has never been done before. Prior works considered only a few options and configurations. Sincero *et al.* [291] considered 352 options and 146 random configurations for the non-functional property scheduling performance. Siegmund *et al.* [290, 289] considered

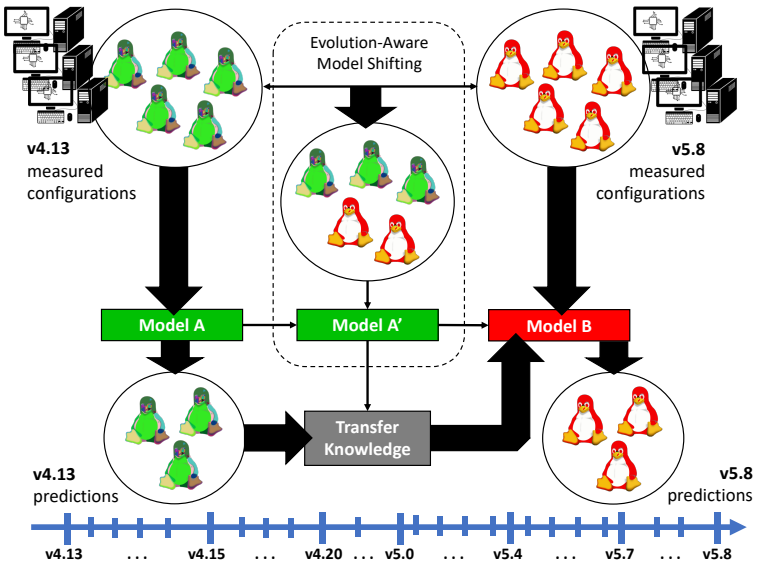


Figure 4.12: An overview on how to predict the performance of Linux kernel configurations over versions 4.13 and 5.8.

25 options and 100 random configurations for binary size. In this study and in contrast to prior works, we make no assumption about the supposed influence of some options and our experiments consider the entire 9K+ options of the Linux kernel on the x86_64 architecture. We also considered a baseline of 95K+ different configurations for 4.13 version and 20K+ configurations for 7 other versions. Beyond Linux, only dozens of options over a few configurations are usually considered in the literature of configurable systems [29]. The gap with the Linux case is substantial, up to the point some learning algorithms, including the ones proposed in the literature, may not scale or have poor accuracy.

In the second part, we first show that a size prediction model learned from one specific version quickly becomes obsolete and inaccurate, despite a huge initial investment (15K hours of computation for building a training set of 90K configurations). In response, we developed a heterogeneous transfer evolution-aware model shifting (tEAMS) learning technique (see Figure 4.12). tEAMS is capable of handling new options that appear in new versions while learning the function that maps the novel effects of shared options among versions. Our results show that the transfer of a prediction model leads to accurate results (the prediction error remains low and constant) without the need of collecting a very large corpus of measurements configurations. With only 5K configurations, we can transfer the model made in September 2017 for the 5.8 version released in August 2020 with similar accuracy.

Learning methods to the test of Linux

In this part, we present empirical results with the Linux kernel (version 4.13). More details about the study design, data, and insights are available in [25].[10]

Gathering configuration data. Our approach, like other learning-based performance models, requires engineering effort to measure non-functional properties (here: binary size of a Linux kernel) out of a sample of configurations. We have developed the tool TuxML to build the Linux kernel in the large *i.e.*, whatever options are combined. TuxML uses Docker to host the numerous packages and tools needed to compile, build, and measure the Linux kernel. Inside Docker, a collection of Python scripts automates the build process. Docker offers a *reproducible* and *portable* environment – clusters of heterogeneous machines can be used with the same libraries and tools (*e.g.*, compiler versions). In particular, we can use a grid computing or a cloud infrastructure to build a large set of configurations.

The two main steps followed by TuxML to measure kernel binary sizes are as follows:

1. *Sampling configurations.* For this step, we relied on on randconfig to randomly generate Linux kernel configurations. randconfig has the merit of generating valid configurations respecting the numerous constraints between options. It is also a mature tool that the Linux community maintains and uses [213]. Though randconfig does not produce uniform, random samples, there is still a wide diversity within the values of options (being *'y'*, *'n'*, or *'m'*). We use this sample to create a .config file.

2. *Kernel size measurement.* Given a set of .config files, TuxML builds the corresponding kernels. Throughout the process, TuxML measures the size of vmlinux, a statically linked executable file containing the kernel. We saved the configurations and the resulting sizes in a database.

The outcome is a dataset of Linux configurations together with their binary size. Each line of the dataset is composed of a set of configuration option values and a quantitative value. Due to discrepancy in vocabulary between the configurable systems domain and the machine learning domain, we clarify how both terms, "option" and "feature" will be used hereafter. An *option* is a variable in a configuration of a system. In the Linux case, every available option has a value in the configuration file .config.[11] A *feature* is an independent variable given to a machine learning algorithm.

Feature encoding Learning algorithms targeting regression problems requires encoding three possible values of options (*e.g.*, 'y', 'n', 'm') into numerical values. An encoding of 'n' as 0, 'y' as 1, and 'm' as 2 is the most obvious solution. However, some learning algorithms (*e.g.*, linear regression) will assume that two nearby values are more similar than two distant values (here 'y' and 'm' would be more similar than 'm' and 'n'). This encoding will also be confusing when interpreting the negative or positive weights of a feature. There are many techniques to encode categorical variables (*e.g.*, dummy variables [114]). We observe that the 'm' value has no direct effect on the size since kernel modules are not compiled into the

[10] In fact a revised version of this preliminary report is available in Hugo's Martin PhD thesis. Hugo has significantly improved the report with new experiments, better explanations and insights. He has investigated the use of *tree-based feature selection* as a means to identify a subset of relevant options for a learning model. Empirical results show that tree-based feature selection can achieve low prediction errors over a reduced set of options and even outperforms the accuracy of learning without feature selection. Furthermore, the interpretable information extracted from learning models is consistent with experts' knowledge of Linux.

[11] Technically, the Linux options that are not specified in a .config file have the value 'n'.

Algorithm	MAPE				
	N=10	N=20	N=50	N=80	N=90
OLS Regression	74.54±2.3	68.76± 1.03	61.9 ± 1.14	50.37±0.57	49.42±0.08
Lasso	34.13±1.38	34.32±0.12	36.58±1.04	38.07±0.08	38.04±0.17
Ridge	139.63±1.13	91.43±1.07	62.42±0.08	55.75±0.2	51.78±0.14
ElasticNet	79.26±0.9	80.81±1.05	80.58±0.77	80.57±0.71	80.34±0.53
Decision Tree	15.18 ± 0.13	13.21 ± 0.12	11.32±0.07	10.61± 0.10	10.48± 0.15
Random Forest	12.5±0.19	10.75±0.07	9.27±0.07	8.6±0.07	8.4 ±0.07
GB Tree	11.13±0.23	9.43±0.07	7.70±0.04	7.02±0.05	6.83±0.10
N. Networks	-	13.92 ± 0.99	9.46 ± 0.15	8.29 ± 0.18	8.08 ± 0.12
Polynomial Regression	-	-	-	-	-

Table 4.3: MAPE of learning algorithms for the prediction of vmlinux size, with N being the percentage of the dataset used as training

kernel and can be loaded as needed. Therefore, we consider that 'm' values have the same effect as 'n' values, and these values can be merged. For non-tri-state options, which are only 319, we simply discarded them. With this encoding, the hypothesis is that the accuracy of the prediction model is not impacted whereas the problem is simpler to handle for learning algorithms and easier to interpret.

(RQ1) How do state-of-the-art techniques perform on large configurable systems? Most of the studied techniques could perform their task in the time and memory limits we had set. We however failed to get results from two of the techniques we have tried: SPLConqueror and Polynomial regression:

- Importing the Linux dataset into SPLConqueror raises an error about insufficient memory and cannot perform anything on the dataset.[12] As stated in [185], for p options, there are p possible main influences, $p \times (p-1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options. In the worst case, all 2-wise or 3-wise interactions among the 9K+ options are included in the model, which is computationally intractable. Even if a subset of options is kept, there is a combinatorial explosion of possible interactions. It may hinder the scaling of the method or dramatically increase the training time. Kolesnikov *et al.* [185] reported that it take hundreds of minutes for systems with less than 30 options, which is far from 9K+ options. Furthermore, linear regressions, used as part of the stepwise process to keep relevant options or interactions, are not accurate enough in the context of Linux (see Table 4.3).

- Polynomial regression integrates interactions among features (in the same vein as performance-influence model) and does not scale for a degree 2.

Table 4.3 reports the highly variable accuracy in *mean absolute percentage error (MAPE)* of all successfully tested techniques with various training set sizes. Most of the selected algorithms are sensitive to *hyperparameters*, which may affect accuracy results. Selecting the right values for hyperparameters should not be neglected. Otherwise, the best algorithm could be sub-optimal after the hyperparameter optimization. We optimize their hyperparameters, and explore a wide range of values as part of our study using grid search and cross-validation.

[12]We used SPLConqueror from commit 9b68ce on Ubuntu 20.04 LTS and got the message "System.OutOfMemoryException: Insufficient memory to continue the execution of the program."

For decision trees, we find the best hyperparameters to be at max depth of 27 and minimum samples split of 45. For random forests, a maximum depth of 20 and a minimum samples split of 10, over 48 estimators (to match the 4 cores/8 threads capacity of the machine). For gradient boosting trees, the maximum depth was 15 and the minimum sample split was 100, over 50 estimators. Our implementation of neural networks is a multilayer feed forward network. Linux configurations go through three dense layers with ReLU activation functions. We rely on an Adam Optimizer [182], since in our case, it had better convergence properties compared to a standard stochastic gradient descent. Besides, the architecture of the neural network has not been designed for relatively small training sets. For this reason, we only experiment with 20K+ configurations. We fed the network with batches of 50 configurations.

We can distinguish multiple groups of similar performance and techniques:

- Linear regression based techniques: OLS Regression, Ridge, ElasticNet and Lasso all present poor results, with Lasso being the only one with less than 50% MAPE, but still 34% at best. The results show that linear regressions are not suited for Linux and that the problem of predicting size cannot be trivially resolved with a simple additive, linear model (as hypothesized early in the section);

- Tree-based techniques: Decision Tree, Random Forest and Gradient Boosting Tree all show MAPE at less than 20% even with "only" 10% of the dataset, even reaching 6.83% for Gradient Boosting Tree at 90% of the dataset. Decision Trees is inferior to Random Forest and Gradient Boosting Tree;

- Neural Networks work quite well but require much more data to be efficient compared to tree-based techniques.

Key results (RQ1). At the exception of Performance-Influence model and Polynomial Regression, most of the techniques studied can handle the Linux dataset in reasonable time and memory limits. On the accuracy side, we can notice that Linear regression based techniques do not present very accurate results, and only Tree-based and neural network technique are suited for Linux.

Impacts of Linux evolution on learning models

As any software system, configurable systems evolve with many commits that may modify the entire architecture and source code. In addition, options may be added or removed during evolution. All these modifications can have an impact on the performance distribution of the configuration space: the effects of individual options may change as well as the interactions among them. Thus, for large and complex configurable systems, one has to manage both the combinatorial explosion of possibly thousands of options (yielding *variants*, *i.e.*, variability in space) and the continuous rapid evolution (yielding *versions*, *i.e.*, variability in time). In general, learning variability in both space and time is indeed challenging. Linux is an extreme case of a highly complex configurable system that rapidly evolves [195, 196, 262, 244, 110].

We aim to quantify the impact of Linux evolution (*i.e.*, the release of a new kernel version) on configuration binary size. Mühlbauer *et al.* [222] investigated the history of software performance to isolate when a performance shift happens over time. If we know evolution can impact the performance of a configurable software, we do not actually know if and how much it can impact a performance prediction model.

A hypothesis is that the evolution has no significant impact and the Linux community can effectively reuse a binary size prediction model across all versions. The counter hypothesis is that the evolution changes the binary size distributions: in this case, a measurable and practical consequence would be that a binary size model becomes inaccurate for other versions. In other words, if the degradation of the accuracy of a prediction model is to be expected, it is necessary to know whether such degradation is sharp enough to be a problem for the Linux community. However, none of these hypotheses have been investigated in the literature. Therefore, quantifying the impact of evolution is crucial and boils down to address the following research question: **(RQ2) To what extent does Linux evolution degrade the accuracy of a binary size prediction model trained on a specific version?** To address it, we measure the accuracy of a performance prediction model, specifically a binary size prediction model, trained in one specific version (*i.e.*, 4.13), when applied to later versions (*e.g.*, up to 5.8).

Datasets. We compiled and measured Linux kernels on seven different versions. Table 4.4 further details each considered release version:

- 4.13: this release was the starting point of our work with huge investments (builds and measurements of 90K+ configurations);
- 4.15: the release was the first to deal with the serious chip security problems meltdown/spectre [254] that mainly apply to Intel-based processor (x86 architecture). A broad set of mitigations has been included in the kernel, which can have an effect on kernel sizes;
- 4.20: the last version before 5.0, with several x86/x86_64 optimizations. As part of the in-depth analysis on the evolution of core operation performance in Linux [262], Ren *et al.* identified several changes in latency for versions between 4.15 and 4.20;
- 5.0: interestingly, this version has been the subject of some debates about the decrease of kernel performance on some macro-benchmarks (*e.g.*, see [159]);
- 5.4: it is a long term support release that will be maintained 6 years. This version also includes modifications for dealing with Linux performance [159, 157];
- 5.7: a recent version, more than a half-year after 5.4;
- 5.8: Linus Torvalds commented[13] *"IOW, 5.8 looks big. Really big."* and reported *"over 14k non-merge commits (over 15k counting merges), 800k new lines, and over 14 thousand files changed"*, suggesting an important and challenging evolution to tackle.

As depicted in Table 4.4, the continuous evolution from 4.13 to 5.8 is significant in terms of numbers of added/deleted options, delta of the commits and the changes files. Note that those changes are computed for each release *w.r.t.* 4.13. For all versions, we specifically targeted the x86-64 architecture, *i.e.*, technically, all configurations have values CONFIG_X86=y and CONFIG_X86_64=y. Overall, we span different periods during 3 years, with some modifications (security enhancements, new features) suggesting possible impacts on kernel non-functional properties (*e.g.*, size). For each version, we build thousands of random configurations (see Table 4.4 column [Examples]). Owing to the computational cost, we balance the budget to measure at least and around 20K+ configurations per version. Such data is used to test the accuracy of a prediction model. We used TuxML and relied on randconfig to ran-

[13] https://lore.kernel.org/lkml/CAHk-=whfuea587g8rh2DeLFFGYxiVuh-bzq22osJwz3q4SOfmA@mail.gmail.com/

Version	Release Date	Examples	Seconds/config	Options	Features	Deleted features	New features
4.13	2017/09/03	92,562	271†	12,776	9,468	-	-
4.15	2018/01/28	39,391	263†	12,998	9,425	342	299
4.20	2018/12/23	23,489	225	13,533	10,189	468	1,189
5.0	2019/03/03	19,952	247	13,673	10,293	494	1,319
5.4	2019/10/24	25,847	285	14,159	10,813	663	2,008
5.7	2020/05/31	20,159	258	14,586	11,338	715	2,585
5.8	2020/08/02	21,923	289	14,817	11,530	730	2,792

Table 4.4: Dataset properties for each version. The number of deleted/new features are *w.r.t.* 4.13. † for versions 4.13 and 4.15, the build time (number of seconds to build one configuration) should be interpreted with caution since we used heterogeneous machines and did not seek to control their workload

domly generate Linux kernel configurations. The distribution of binary size in our dataset varies depending on the version. While the mean binary size on version 4.13 is 47 MiB, for other versions that mean value is between 89 MiB and 118 MiB. The minimum size for all version is around 10 MiB and the maximum around 2 GiB.

Base prediction models We chose *gradient boosting trees (GBTs)* that obtain the best results whatever the training set size on version 4.13. We trained GBTs with 85.000 examples on version 4.13. As a performance model only matches a specific set of features (here: the features of 4.13), we deleted features only contained in further, target versions (*e.g.*, 4.15).

Results (RQ2). Figure 4.13 shows the degradation of models trained on the Linux Kernel version 4.13 by plotting their error rate (meaning lower is better) on later versions. The models get on average 5% MAPE on 4.13, and less than two versions after, on 4.15, the error rate is 4 times higher at 20%. It keeps this error rate for multiple versions, at least up to 5.0, and goes even higher, at 32% for the version 5.7 and 5.8, *i.e.*, an error rate 6 times higher. Note that the degradation does occur independently from the training set size, *i.e.*, with both 20K and 85K configurations.

A direct reuse of the prediction model is inaccurate for the early version 4.15 and subsequent ones (4.20, and 5.0). Moreover, the degradation slightly decreases between 5.7 and 5.8. A possible explanation is that the binary size distributions of 5.8 is closer to 4.13, at least for the way the basic transfer is performed. It also suggests an effect of the evolution between 5.7 and 5.8. Besides model reuse with 20K is more accurate than model reuse with much more budget (85K) for all target versions, except 5.8. It is not what we would have expected for a learning model: a larger training set for the source model should lead to improved accuracy. This shows that despite the evolution changes both at the code and options level among the releases (see Table 4.4), reusing a model does not follow a logical or explainable reason from a machine learning point of view. Thus, overall, simply reusing a prediction model is neither accurate nor reliable: the evolution of the configurations binary size is not captured. We also measured the degradation of prediction models trained on other versions with 15K (see Figure 4.13). We can observe that the degradation is less immediate than with the version 4.13 but is still happening, especially on version 5.8 as error rate is raising to 40% - 50%.

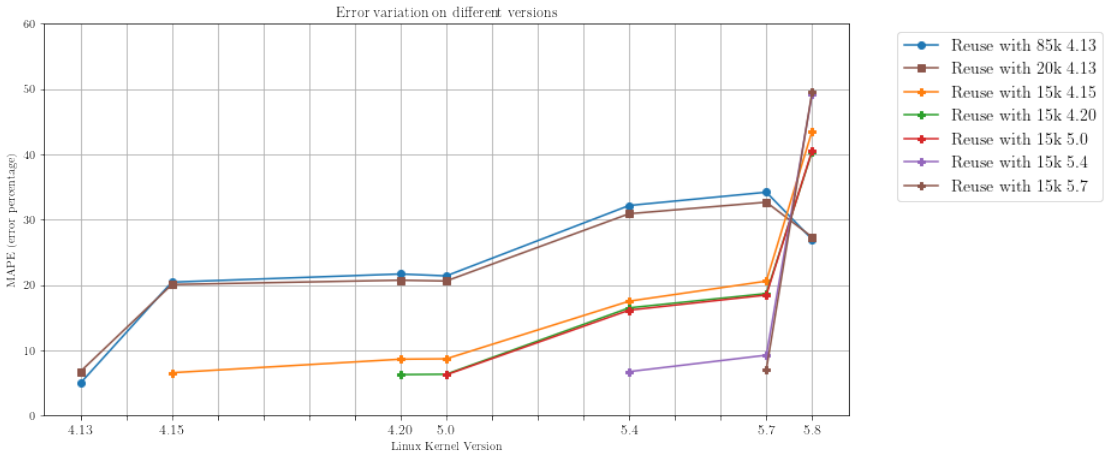


Figure 4.13: Accuracy of prediction models, trained on 4.13, with training set size 20K and 85K, when applied on later versions.

One of the main results is that the prediction models cannot be reused as such and degrade over time, regardless of the size of the training set. It calls to a more accurate and sustainable solution like transfer learning [29, 312, 168].

Insights about evolution and options. We compare feature importances from models trained on versions 4.13, 4.15, and 5.8. To do so, we rely on *feature importance* a model agnostic, widely considered score for computing the increase in the prediction error of the model after we permuted the feature's values [220]. Feature importance provides an integrated and global insight into the prediction model of a given version: the score takes into account both the main feature effect and the interaction effects on model prediction. We perform the computation out of GBTs. Our observations show that numerous features involved in different evolution patterns can cause the degradation of a prediction model. The impact of new features, unknown by the old model, but also the changes in importance of known features, challenge a prediction model trained on a specific version. Interestingly, we did not find important features that were removed between version 4.13 and 4.15. Another important insight is that a large subset of features remain important across versions. For example, when comparing 4.13 and 4.15, out of the top 50 features from both list (the top 50 representing 95% of the feature importance), 29 are the same. One can leverage this knowledge for building a prediction model. Overall, there is a potential to transfer a model from one version to another under the conditions new features together with the effects of important features are correctly handled. In fact, the insights drive the design of tEAMS: more details hereafter.

Key results (RQ2) The evolution does impact configuration prediction and the degradation quickly occurs: from less than 5% to 20% (only after 4 months of evolution) and up to 32% for recent versions. The reuse of a prediction model on different versions is not a satisfying solution, calling for other approaches.

Transfer learning across variants and versions

To overcome this issue, we propose to transfer the learning across versions. Figure 4.12, page 139 gives the general principle about how transfer learning works. Here, to make predictions over the Linux kernel version 5.8, we could directly reuse the performance model A built from the source version 4.13. Basically, the source model A is adapted to consider the aligned set of features from both source and target domains (*i.e.*, model A'). Finally, the target model B is trained with only a few measured configurations in the target domain B plus the knowledge from the modified source model A'.

Heterogeneous transfer learning problem However, the evolution of Linux brings a specific scenario for transfer learning: configuration options for the source version (*e.g.*, 4.13) are not the same as further, target versions (*e.g.*, 5.8). In terms of machine learning, since options are encoded as features, the feature spaces between the source and target version are non-equivalent. Since the feature space (*i.e.*, the set of configuration options) can change across versions, our approach falls under the category known as *heterogeneous* transfer learning, opposite to *homogeneous* transfer learning, that assumes the feature space remains unchanged during evolution. Identifying how to efficiently apply transfer knowledge of the learned model as the systems evolve is challenging. This is indeed a well-known general problem in machine learning [239, 319], made even more difficult because of the heterogeneity of configuration spaces (due to the fact that features come and go across versions) may cause bias on cross-version feature representation [104].

Principles. The intuition is that, for Linux, the shared set of features can be exploited to effectively transfer predictions. We now present *evolution-aware model shifting (*tEAMS*)*, a method to transfer a prediction model for a given source version onto a target version. The major challenge is to bridge the gap between the feature spaces. We rely on two steps: (1) feature alignment, which deals with the differences between features' sets among two versions; (2) the learning of a transfer function that map features' source onto target size. For realizing *feature alignment*, we distinguish three cases:

- **commonality**: options that are common across versions (*i.e.*, options have the same names) are encoded as unique, shared features. There are two benefits: we can reuse a prediction model obtained over a source version *"as is"*, without having to retrain it with another feature scheme; we do not double the number of features, something that would increase the size of the learning model up to the point some learning algorithms might not scale. The anticipated risk is that some Linux options, though common across versions, may drastically differ at the implementation level, thus having different effects on sizes. We deal with this risk through the learning of a transfer function that aims to find the correspondences between the source and the target (see below);

- **deleted features**: options that are in the source version, but no longer in the target version: we add features in the target version with one possible value, *"0"* or *"1"*. Observations show that putting *"1"* as the default always gives slightly better accuracy.

- **new features**: options that are not in the source version, but only introduced in the target version: we ignore them when predicting the performance value since the source model cannot handle them, but we keep them in the target dataset.

Feature alignment alone is not sufficient; it is mainly a syntactical mapping at this step. There is a need to capture the *transfer function*, *i.e.*, the relationship between the source features, the source labels (kernel size of each configuration under source version), the target features, and the target labels. This transfer function should be learned. Owing to the complexity of the evolution, a "simple" linear function is unlikely to be accurate – our empirical results confirm the intuition, see next section. In contrast to existing works that rely on linear regression models for "shifting" the prediction models [312, 168, 190], we rely on more expressive learning models, capable of capturing interactions between source and target information. Note that the feature alignment is a completely automated process and relies on the high similarity between the features spaces. In case of too disjoint features spaces, this solution would likely fail, and other solutions should be considered [104]

Algorithm. There are four key steps as part of tEAMS (a Python-like pseudocode is also given below): ① Target dataset and Source model acquisition: Train or acquire a robust model on measurements from the source version and a dataset from the target version; ② Feature alignment: If the source and the target do not have the same set of options, an alignment of the feature spaces is applied (*e.g.*, as previously described); ③ Target prediction: Using the source model, predict the value of the target data and add this prediction as a new feature in the target dataset; ④ Shifting model training: Using the enhanced target dataset, train a new model (*e.g.*, with a Gradient Boosting Tree algorithm capable of handling interactions).

```
def model_shifting(source, target):
    # Training source model on the source data
    source_model = GBT.train(source)
    # Align feature from source to target
    target = feature_alignment(source.columns)
    # Predict the performance of the target data
    # using the source model
    pred_perf = source_model.predict(target)
    # Include the prediction
    # in the target dataset
    target["pred_perf"] = pred_perf
    # Train another model
    # on the enhanced target data
    shifting_model = GBT.train(target)
    return source_model, shifting_model
```

Note that the source model is usually already trained beforehand, and its training step can be skipped in this case. Overall, our solution is fairly easy to implement and deploy.

Variant: Incremental Transfer A possible variant of this technique is to use it in an incremental fashion, and to replace the source model by an already transferred model for a previous version. In the end, such a model consists of a source model, shifted multiple times in a row through multiple intermediate target versions until the final target version. This variant could potentially give more accurate results, since the complexity of the transfer is spread over multiple models. The farther two versions are from each other, in terms of software evolution, the more performances-impacting changes can happen. A transfer model that handles two distant versions has to deal with all changes between these two versions

at once, while in an incremental process, each model only has to deal with a fraction of the changes. On the other hand, we know that machine learning models are imperfect and error prone, even if the error is limited. Relying on a series of machine learning models can turn out to be risky, as these errors can be spread and amplified over the multiple models.

Effectiveness of transfer learning

Our goal is to evaluate the cost-effectiveness of our approach tEAMS in the context of Linux evolution. The effectiveness is the accuracy of the prediction model and its ability to minimize prediction errors as much as possible. If the source version and the target have very little in common, configuration performance knowledge may not transfer well. In such situations, transfer learning can be unsuccessful and can lead to a "negative" transfer [104]. Specifically, we consider that the transfer is negative when learning from scratch directly onto the target version – without transfer and using only the limited measured target data available for transfer – leads to better accuracy than a transfer model with the same budget. Thus, we aim to answer the following research question:

(RQ3) What is the accuracy of our evolution-aware model shifting (tEAMS) compared to learning from scratch and other transfer learning techniques? The accuracy of tEAMS depends on the investment realized for creating or updating the prediction models. Specifically:

- the number of configuration measurements over the target model used to train the prediction model: non-transfer learning (*i.e.*, from scratch) uses the same training set and we can confront our results;
- the number of configuration measurements over the source version used to train the prediction model: from large training sets to relatively small ones;

Hence, we address RQ2 through different cost scenarios and we can identify for which investments tEAMS is effective.

Experimental settings. For training and validating the prediction models, we use the same kernel versions and configuration measurements as in Table 4.4.

Training size for targeted versions We vary the number of configuration measurements amongst the following values $\{1K, 5K, 10K\}$. $5K$ corresponds to around 5% of the 95K configurations in the dataset of 4.13: it is representative of a scenario in which a relatively small fraction is used to update the model for a target version. As we have invested around 20K per version, we needed to take care of having a sufficiently large testing set for computing the accuracy. In particular, we cannot use the whole configuration measurements since otherwise we cannot simply compute the accuracy of the models. We stop at $10K$ since then the testing set can be set to around $10K$ too. Moreover, we repeat experiments 5 times with different training sets and report on standard deviations.

We compare tEAMS with different methods. We give some details hereafter.

Source prediction model for tEAMS: We use a prediction model trained with 4.13. It is the oldest version in our dataset and as such, we investigate an extreme scenario for the evolution and potentially the most problematic for transfer learning. We rely on GBTs, the most accurate solution whatever the training set size is. We train GBTs over 4.13 with two different budgets: 85K configurations and 20K configurations. Hereafter, we call these prediction models `4.13_85K` and `4.13_20K` respectively.

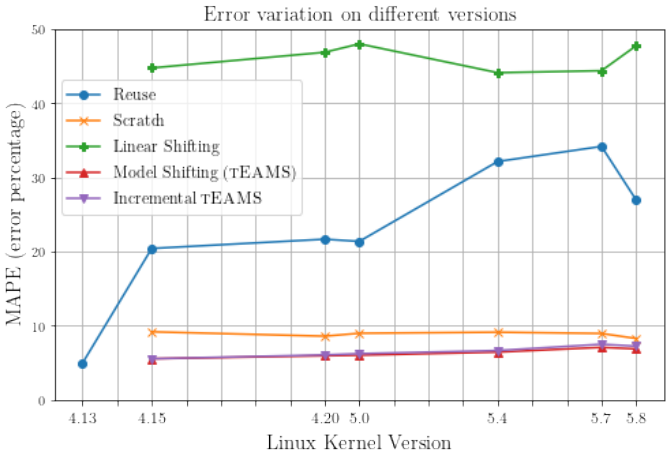


Figure 4.14: Accuracy of TEAMS 4.13_85K compared to other techniques using 5K examples for the target

Incremental TEAMS: We use the incremental method in the same way as without increment, only changing the base model for each increment by the model trained on the previous version. We also have two different series of increments, one based on the 4.13_85K model, the other on the 4.13_20K model. For instance, the first series starts with the transfer from model 4.13_85K to version 4.15 with a shifting model T4.15. This process creates a prediction model 4.15 composed of the two models : 4.15 = T4.15(4.13_85K). The next step is to transfer that model to version 4.20 : 4.20 = T4.20(4.15). At the end of that series, we have a model looking like this :

5.8 = T5.8(T5.7(T5.4(T5.0(T4.20(T4.15(4.13_85K))))))

Learning from scratch: The simplest way to create a prediction model for a given version is to learn from scratch with an allocated budget. We use the GBTs algorithm to create prediction models from scratch, for each version of our dataset. As previously shown, GBTs are a scalable and accurate solution compared to other state-of-the-art ones. Furthermore, the superiority of GBTs is more apparent when small training sets are employed. This quality of GBTs is even more important when learning for the target version where the budget for updating the model is typically limited – we investigate budget with less than 20K measurements (see above *"Training size for targeted versions"*). We replicated the experiments of 4.13 on other versions: linear models, decision trees, random forests, and neural networks give inferior accuracy compared to GBTs, especially for small sampling size (*e.g.*, 10K). Thus, we do not report results of other learning algorithms and keep only GBTs, the strongest baselines for learning from scratch or for transfer learning techniques.

TEAMS *with linear-based transfer function* In most state-of-the-art cases, model shifting processes use a simple linear learning algorithm to create a shifting model and they perform quite well (*e.g.*, [312, 190]). We rely on such a linear transfer function and also apply feature alignment as part of TEAMS.

Results (RQ3). Figure 4.14 depicts the evolution of the MAPE for the reuse of the model 4.13_85K (*i.e.*, 4.13 with a 85K of training set), and the 4 studied techniques trained using 5K examples. We can quickly see that linear model shifting has more than 40% MAPE over all versions and is not accurate at all. It is surprisingly the worst by far, in particular, in comparison with the direct reuse of the prediction model. The standard deviation for Linear model shifting is between 1.5 and 3, while all other techniques are much more stable with a standard deviation always at 0.1 or less. Moreover, learning from scratch with 5K examples allows to create models having an MAPE between 8.2% and 9.2% quite consistently. On the other side, TEAMS with the same budget offers a lower MAPE from 5.6% on version 4.15 to 6.8% in version 5.8 with a peak at 7.1 in version 5.7. It is worth noting that TEAMS MAPE increases a little bit at each version. However, the increase is not significant and remains low. Comparing TEAMS with its incremental variant shows a very slight but constant advantage over the variant, which also shows better results than learning from scratch.

Table 4.5 gives the results for MAPE with combinations of different models used (4.13_20K and 4.13_85K) and training set sizes (1K, 5K, 10K) for the scratch baseline, TEAMS and Incremental TEAMS. The other techniques performing poorly, Table 4.5 hence focuses on the three best solutions of Figure 4.14. We now report on their results.

Impact of training set size over target. As illustrated in Table 4.5, when decreasing the training transfer set for the newer versions to 1K examples (1% of the original set), the MAPE increases to 14.9%-16.7% depending on the version. Whereas, the MAPE for TEAMS only increases to 6.7%-10.6%, with the same trend consistently increasing MAPE over time (and versions). For Incremental TEAMS, the error rate increases faster up to 13.3% on version 5.8. On the other hand, if we increase the training set to 10K (10% of the original set), accuracy when learning from scratch gets better, with 7.0% to 7.7% MAPE. For TEAMS, the accuracy also gets better, from 5.2% MAPE on version 4.15 to 6.1% on version 5.7 and then slightly further improves to 6.1% on version 5.8. We observe the same trend for Incremental TEAMS, going up to 6.5% on 5.7 and then to 6.2 on 5.8.

Impact of the TEAMS *source model.* We measured the same variations using the model 4.13_20K as the source model, which was built from 20,000 examples instead of 85,000. This affects TEAMS by slightly increasing the MAPE. In particular, we observe that for TEAMS: 1) with 1K, the MAPE varies from 8.5% to 11.6%, 2) With 5K, it varies from 6.7% to 7.9%, and 3) with 10K, it varies from 6.2% to 6.7%. Whereas for learning from scratch, we observe that: 1) with 1K, the MAPE varies from 14.9% to 16.7%, 2) With 5K, it varies from 8.3% to 9.2%, and 3) with 10K, it varies from 7.04% to 7.67%. Therefore, our results show that TEAMS outperforms the two baselines, regardless of the size of training sets. In this situation, Incremental TEAMS also shows slightly better results than TEAMS in some cases. At 10k, Incremental TEAMS beats TEAMS on all versions except 5.7, and at 5k, only for versions 4.20 and 5.0. Given the fair increase in error rate at 1k, Incremental TEAMS seems to be very sensitive to higher error rate from previous versions.

Computational cost of training. We performed our experiments on a machine with an Intel Xeon 4c/8t, 3,7 GHz, 64GB memory. Training from scratch with 1K, 5K and 10K examples takes respectively 21, 195 and 407 seconds. Learning with TEAMS takes a little more time with 60, 288 and 604 seconds for the same number of examples. The training time of TEAMS for updating a prediction model is thus affordable and negligible compared to the time taken to build and measure the kernel configurations. The overall cost of training is mainly due

Version	Scratch			tEAMS						Incremental tEAMS					
				4.13_20K			4.13_85K			4.13_20K			4.13_85K		
	1k	5k	10k	1k	5k	10k	1k	5k	10k	1k	5k	10k	1k	5k	10k
4.15	16.72	9.19	7.46	8.46	6.69	6.21	6.73	5.56	5.19	8.46	6.69	6.21	6.73	5.56	5.19
4.20	16.39	8.60	7.12	8.85	6.94	6.22	7.64	5.96	5.44	9.49	6.89	6.15	8.39	6.08	5.46
5.0	15.50	8.99	7.07	9.14	7.04	6.34	7.80	6.03	5.48	10.32	6.99	6.15	8.84	6.24	5.63
5.4	16.06	9.14	7.67	9.76	7.06	6.39	9.01	6.45	5.71	11.64	7.23	6.10	10.56	6.66	6.07
5.7	15.63	8.96	7.59	11.56	7.85	6.69	10.13	7.09	6.12	13.77	7.90	6.75	12.57	7.51	6.50
5.8	14.91	8.29	7.04	11.47	7.27	6.41	10.62	6.88	6.06	13.82	7.58	6.39	13.29	7.26	6.19

Table 4.5: MAPE for learning from scratch and tEAMS, with varying source models and training set sizes for the target

to the training of the source model (details can be found in [208]) which is done only once. As a final note, building kernels and gathering configuration data (see Table 4.4) is by far the most costly activity – the time needed to train the prediction model out of data through either transfer learning or from scratch (a few minutes) is negligible.

Key results (RQ3, summary). tEAMS and Incremental tEAMS are more accurate solutions than learning from scratch and linear model shifting to predict Linux Kernel size on different versions. Also, Incremental tEAMS shows results that are mostly worse than tEAMS and without significant improvement. Even with different source models and training set sizes, tEAMS keeps better and acceptable accuracy with 6.9% MAPE on the latest 5.8 version leveraging a model trained on 3 years old data.

Integration of tEAMS in the Linux project. We envision to integrate tEAMS as part of the ongoing continuous integration effort on the Linux kernel. We have released a tool, called kpredict,[14] that predicts the size of the kernel binary size given only a .config file. kpredict is written in Python, available on pip, and supports all kernel versions mentioned in this article. A usage example is as follows:

```
> curl -s http://tuxml-data.irisa.fr/data/configuration/167950/config -o .config
> kpredict .config
> Predicted size :  68.1MiB
```

whereas the actual size of the configuration (see http://tuxml-data.irisa.fr/data/configuration/167950/ for more details) is 67.82 MiB.

Recently KernelCI [177], the major community-effort supported by several organizations (Google, Redhat, *etc.*), has added the ability to compute kernel sizes and this functionality is activated by default, for any build. tEAMS will benefit from such data. Besides, the current focus of KernelCI and many CI effort is mostly driven by controlling that the kernels build (for different architectures and configurations). It is not incompatible with the prediction of kernel sizes since we did not employ a sampling strategy specifically devoted to this property. We rely on random configurations that are used to cover the kernel and find bugs (see *e.g.*, [24]). In passing tEAMS can benefit from kernel sizes data while the CI effort continues to track bugs.

[14] https://github.com/HugoJPMartin/kpredict/

replication

Scripts (*e.g.*, notebooks), analysis about Kconfig documentation, and data about 4.13 are available: https://github.com/TuxML/size-analysis/. We also provide a replication package with all artifacts (including datasets for the 7 versions and learning procedures): https://zenodo.org/record/4960172.

4.5 Wrap-up, applicability, and limitations

I have described a systematic process "sampling, measuring, learning" with a high applicability. Many subject systems and non-functional properties can be considered (see our survey [29]), even on the huge configuration space of Linux. The key idea is that statistical, supervised machine learning techniques operate over a sample of configurations' observations. I have shown how variability knowledge can be recovered: constraints among options can be synthesized and (performance) prediction models can be derived. Such information is usually hard to know or formalize since it requires the knowledge of the whole configuration space. As shown, developers, maintainers, or testers can use machine learning to discover unknown variability knowledge. Learning variability can thus play two roles: (1) reinforcing a variability model and improving its quality; or (2) augmenting a variability model with non functional properties (*e.g.*, performance). Learning as reverse engineering has the merit of pushing automation and being linked to the actual (artefacts of the) system. The process can be repeated (*e.g.*, throughout evolutions).

Despite all these qualities, it is worth noticing that statistical learning is unsound and incomplete. In the quest of generalizing, bias (and prediction errors) is a by-product rather than an explicitly enforced property. The effectiveness (accuracy) depends on the quality and quantity of dynamic observations, which may exhibit a high cost. A pre-requisite to apply learning techniques is the availability of procedures to observe the system. It is challenging to deploy at scale and at reasonable costs reliable measurements of thousands of configurations. Yet, developers, maintainers or performance engineers can embrace such limitations: they would be unable to formalize or know this knowledge anyway. Besides, learning also benefits from human supervision (*e.g.*, for encoding variability data). In a sense, learning is effective when jointly used with modelling.

Overall, the process of learning variability has the potential to provide an accurate configuration oracle capable of predicting a property of interest. Yet, the underlying costs and the prediction errors can be a show-stopper (*e.g.*, for critical systems). It is also worth noting that a human (*e.g.*, a developer) can well elaborate a perfect variability model at zero computational cost. For these reasons, I have focused on and contributed to learning process that synthesizes interpretable variability information. Developers, maintainers, or users can read, understand, review, and possibly re-inject this information as part of a global, integrated process.

Chapter 5

Conclusion

This document described the research I conducted from 2012 to 2021 at University of Rennes 1 in the DiverSE team (IRISA, Inria). The presented work concerns the rich field of software variability. As early illustrated in Section 1.1 and I hope demonstrated in the manuscript, software variability is ubiquitous in numerous domains, systems, artefacts, and engineering scenarios. When teaching, I am asking the students for examples of software systems with variability. The conclusion of the discussions is most of the time a question: Are there systems without variability?

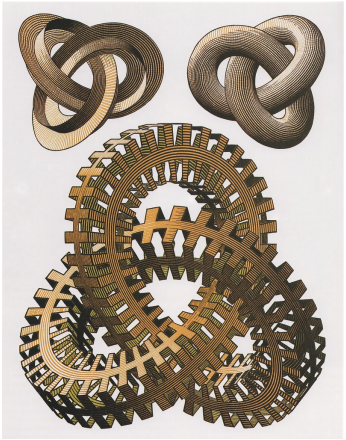
Variability is challenging mainly due to the combinatorial explosion of possible variants. It is also a source of opportunities to better understand a domain, create reusable artefacts, deploy performance-wise optimal systems, or find specialized solutions to many kinds of problems. In both cases, a model of variability is either beneficial or mandatory to explore, observe, and reason about the space of possible variants. For instance, without a variability model, it is impossible to establish a sampling strategy that would satisfy the constraints among options and meet coverage or testing criteria. I thus address a central question in this HDR manuscript: **How to model software variability?**

My answer is a supervised, iterative *process* (1) based on the combination of reverse engineering, modelling, and learning techniques; (2) capable of integrating multiple variability information (*e.g.*, expert knowledge, legacy artefacts, dynamic observations). Modelling in the sense of manually developing a model (Chapter 2) has the advantage of integrating knowledge about a domain or a system, expressing an intention and defining a scope, but is insufficient. The knowledge can be incomplete, unsound, not available, hard to formally express, and in the long run, when systems frequently evolve, the manual effort cannot be repeated. Reverse engineering (Chapter 3) can provide the necessary automation and mine variability information that even experts cannot synthesize. Yet, reverse engineering can be incomplete and unsound as well, since targeted artefacts may contain partial variability information. Another limitation is that automation pays off under the condition some knowledge is injected into the reverse engineering process. In a sense, reverse engineering can benefit to modelling but also needs modelling. Learning techniques (Chapter 4) have a high potential to recover variability knowledge related to *e.g.*, constraints and performance. By construction, it is worth noticing that statistical learning is unsound and incomplete. The effectiveness (accuracy) also depends on the quality and quantity of dynamic observations,

which may exhibit a high cost. Yet, developers, maintainers or performance engineers can embrace such limitations: they would be unable to formalize or know this knowledge anyway. Similarly as reverse engineering, learning also benefits from human supervision (*e.g.*, for encoding variability data).

Overall, modelling, reverse engineering, and learning alone can well be not sufficient. Going back to Figure 1.4, page 12, modelling as well as reverse engineering and learning approximate the actual variability space (coloured part). I have provided some evidence in this manuscript. Section Using machine learning to infer constraints shows that, despite months of modelling effort and construction of specific languages (Section In search of the right variability language and models), the constraints were incomplete. Hence, modelling is not enough: learning and modelling should be combined. Section Reverse engineering architectural variability models shows that an intentional variability model of the architecture was unsound and incomplete. On the one hand, reverse engineering techniques were crucial to enforce the original variability model. On the other hand, reverse engineering alone would provide an incomplete view of the variability. As suggested, modelling as well as reverse engineering alone are not enough and should rather be combined. The key thus resides in the combinations (*"and"*): the supervised process is about modelling, reverse engineering, and learning.

In fact, the manuscript could well be entitled "Modelling software variability" (*i.e.*, simply "Modelling"). After all, each of the contributions is about finding the right models. However, the process to eventually obtain the models differs. Modell*ing* is one kind of process, similarly as reverse engineering and learning. We can argue that reverse engineering (resp. learning) is modelling. I tend to agree, especially if we consider that reverse engineering (resp. learning) leads to high quality models under the conditions experts injects some models of knowledge. However, the considered artefacts, the degree of automation, the underlying assumptions, the cost differ from a traditional modelling activity, thus justifying to distinguish reverse engineering and learning. I also think the distinction helps to identify the applicability *i.e.*, when to apply reverse engineering, modelling, and/or learning.



M.C. Escher, Knots, Woodcut, 1965. © 2021 The M.C. Escher Company – the Netherlands. All rights reserved. Used by permission. www.mcescher.com

I have contributed together with students, colleagues, and partners in providing a toolbox for capturing variability in a wide range of situations. How to systematically use this toolbox as part of a supervised process is specific to an engineering context. A research direction is thus to assess the different techniques and their combinations, and if possible, to identify what is missing in this toolbox. For instance, can we use reverse engineering techniques to improve the learning of the Linux kernel variability space? How to involve a community of users for modelling variability of Linux or reducing the cost of learning? How to communicate the outcome of the learning to developers, integrators, or users of Linux? Similar questions can be formulated for many software systems. I plan to investigate these questions in the context of real-world software-intensive projects together with the whole variability community.

Chapter 6

Perspectives

In this section, I outline four possible research directions. It is mainly the result of discussions and work with students, colleagues, and partners. Some of the ideas presented here are adapted from an ERC starting grant[1] I have written in 2018. Some of the research directions are ongoing, others are more prospective.

A central observation is that variability in software systems is *deep* and spans different layers (hardware, operating system, build, input data, *etc.*). So-called deep variability is both a threat to the generalization of (variability) models (*e.g.*, performance prediction models) and a source of opportunities to specialize or tune software systems (*e.g.*, *w.r.t.* security). As deep variability exacerbates the combinatorial explosion, we need "smart" techniques to explore in a cost-effective way such large spaces. The challenge is to provide tomorrow's developers, engineers, scientists and citizens with the means to abstract, explore and reason about variability in and with software.

6.1 Deep Software Variability

The content of this section is adapted from the following publication:
L. Lesoil, M. Acher, A. Blouin and J.-M. Jézéquel, 'Deep Software Variability: Towards Handling Cross-Layer Configuration', in *VaMoS 2021 - 15th International Working Conference on Variability Modelling of Software-Intensive Systems*, Krems / Virtual, Austria, Feb. 2021. https://hal.inria.fr/hal-03084276

Software systems can be configured to reach specific functional and performance goals, either statically at compile time or through the choice of command line options at runtime.

An observation is that only considering the software layer might be naive to tune the performance of the system or test that the functionality behaves correctly. In fact, many layers (hardware, operating system, input data, *etc.*), themselves subject to variability, can alter performance or the functionalities of software configurations (see Figure 6.1). For instance,

[1]The project was called "REVARY for Resurrecting Software Variability". The cut was 29% and I was ranked between 30% and 40%. Despite rejection, the writing effort was really worth doing

configuration options of the x264 video encoder may have very different effects on x264's encoding time when used with different input videos, depending on the hardware on which it is executed [199]. That is, only considering the software layer might provide non-optimal values for configuration options of the software.

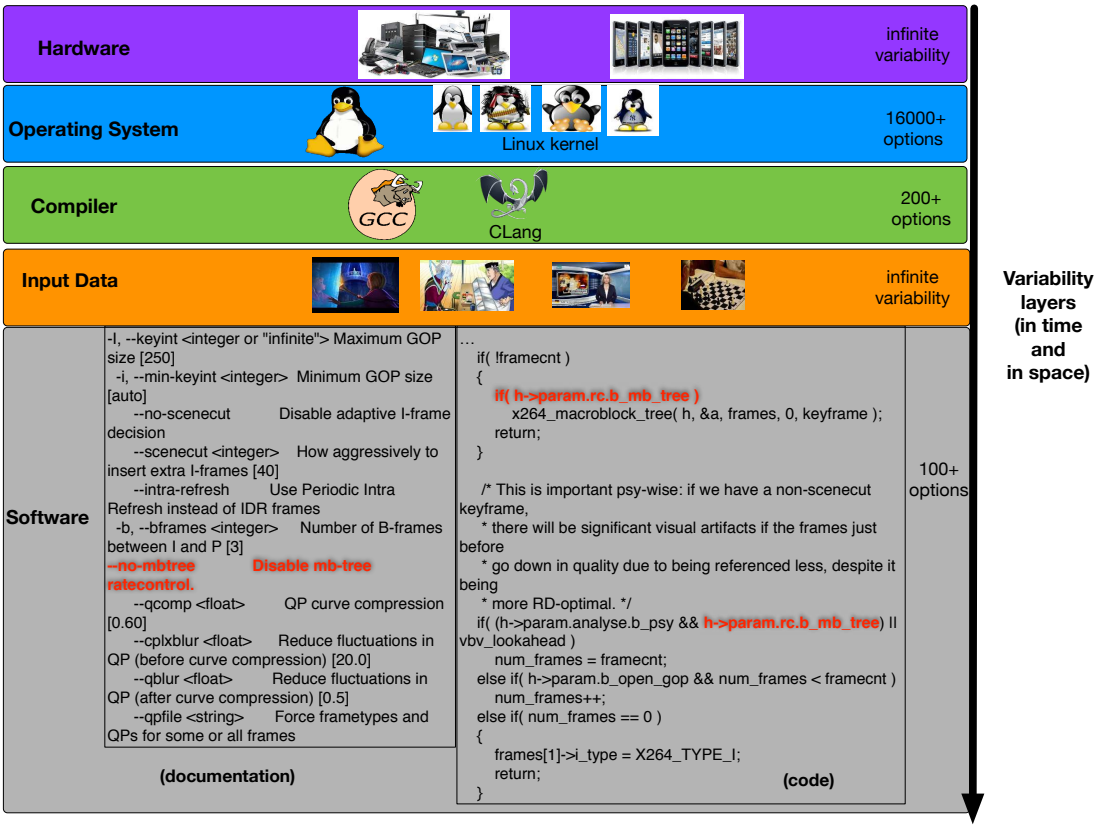


Figure 6.1: x264 deep variability

We call **deep software variability** the interaction of all variability layers that could modify the behavior or non-functional properties of a software. Deep software variability calls to investigate how to systematically handle cross-layer configuration (see Figure 6.1).

Though there is preliminary evidence [28, 200, 167, 199, 238, 126, 88], we ignore to what extent deep software variability impacts the performance of software at this step of the research. May configurable systems be more or less sensitive to deep variability, depending on quantitative properties of interest (*e.g.*, energy consumption, execution time)? Empirical studies and controlled experiments are definitely needed. Once their effects have been identified and quantified, the influential variability factors can be leveraged to improve software performance, while the others remain fixed and can be forgotten (*e.g.*, when benchmarking or transfer learning).

The diversification of the different layers is also an opportunity to test the robustness and resilience of the software layer in multiple environments. That is, developers and ops can exploit deep software variability to detect and hopefully prevent more (performance) bugs. Assisting users in charge of configuring software is a still an open issue *per se*; deep software variability exacerbates this problem. Impacts of different layers are only partially reported in the documentation. Another interesting challenge is to tune the software for one specific executing environment. Instead of enduring deep software variability, the goal would be to pre-select the right environment for the software, tuning each layer separately in such a way it improves the overall software performance.

Overall, there are many challenges and opportunities. In essence, deep software variability questions the generalization of the configuration knowledge. Stated differently, deep software variability is a potential threat to all variability models that have been reverse engineered, elaborated by experts, or learned (*e.g.*, in a fixed computational environment).

6.2 Software Variability and Security

6.2.1 Debloating software variability

The ability of varying is a mandatory feature of any modern (sub-)system. Despite its ubiquitous presence, variability is also an enormous source of complexity with a combinatorial explosion of possible configurations. Several works observed that configuration options can be the source of compilation failures, can crash the system, can compute wrong results, or can degrade the performance [324].

There is another important issue: the attack surface of configurable systems, dynamic adaptive system or software product line is extremely large. Each (combination of) option can be subject to an attack. For example, an attacker can inspect many configuration-related paths in a program. Overall, our observation is twofold:

- *"variability is pointless"*: users do not need necessarily the ability to configure their systems, especially at deployment time. A default configuration can be sufficient for a majority of the use cases while most of the options will simply never been used;
- *"variability is a security threat"*: configuration options are not only pointless, they are an unnecessary opportunity and entry point for attackers [48, 8].

Our idea is conceptually simple: *as configuration options are pointless and represent a threat, we should automatically remove them from the system*.

To realize this idea, there are however a number of challenges to address, the more difficult and important being to automatically remove configuration-related code out of programs.

Identification of irrelevant/relevant options. Depending on the intended usage of the system, the set of important or unimportant options can differ. Some options are mandatory because of the functionality one wants to achieve while some other should be deactivated because they have no effect on the execution time. Previous works show that it is possible to learn important options of a system given an objective (*e.g.*, execution time, see Chapter 4). In the context of sensitive applications, we plan to bring more attention to security concerns as part of the identification process of options.

Tracing configuration options in the code. Once the set of options to be removed is identified, we first need to locate where they are in the source code. Static analysis can be used but has some limitations since options are scattered in the source code and their manifestation may only occur at runtime. For this reason, we also plan to rely on dynamic analysis techniques (*e.g.*, testing) typically to identify what are the paths of program impacted by options.

Removing options in the code. An unexplored problem is to *automatically* remove variability identified as irrelevant. So far, developers and researchers tend to focus on the problem of extending software and adding more and more options [33, 35]. Removing variability code is both counter-intuitive and difficult. An option most likely interacts with other options and its implementation is scattered in different files.

Program specialization is somehow related to the idea [210, 280]. It is a software engineering technique that adapts a program to a given execution context. Information about this context can be provided by the programmer or derived from invariants present in the code. In contrast to compiler optimizations, program specialization is explicitly initiated by the programmer, and thus can adopt a very aggressive strategy for propagating such information [210, 280].

Recently, software debloating has been proposed to keep only the "features" that users utilize and are deem necessary. Some applications and promising results (operating systems, libraries, Web servers like Nginx, OpenSSH, *etc.*) [149, 194, 186] have been reported. The notion of features is usually coarse-grained and debloating is applied over entire source files. Our idea follows a similar direction, but our focus is on the removal of run-time options that are scattered in the source code of a program. Overall, we believe software debloating has an important potential but has to be adapted to efficiently remove configuration options and variability of systems.

Specializing the configuration space of a software system has a long tradition since the seminal paper of Czarnecki *et al.* [298, 209, 163, 96, 95]. However, specialization of configurable systems is focused on the specialization of variability models where constraints among individual options (or across several options) are added to enforce the configuration space (as done in Chapter Learning Software Variability). A missing part (and our idea) is to propagate this specialization to the source code. The a priori knowledge of options' values can be leveraged to produce a new specialized source code: some portions of code (functions, function calls, loops, etc.) related to run-time options can be eliminated (aka. *debloated*). There are at least some specificities and challenges to tackle. First, irrelevant or must-be-included options will initiate the removal. The removal will be on-demand, depending on contextual factors and driven by specific objectives. Second, automatic removal of portions of code impacted by variability is challenging, since variability crosscuts many artefacts, its location is not necessarily made explicit, and there is a risk of breaking original functionality and other options.

Variability specialization can occur offline, but not all variability will be removed since there are options that can still be relevant and selected depending on contextual changes. For this reason, variability specialization can also occur online. Both research directions have not been considered in the literature.

Demonstrating the benefits for security. When options are removed, we aim to show that some attacks are no longer possible while the usability or the functionality of the system is not altered. We plan to study real-world systems, attacks, and options to empirically study the effectiveness of our techniques. We also plan to provide some theoretical guarantees about our removal process.

The work about software debloating is ongoing through the SLIMFAST project funded by the Brittany region and the DGA (see Section 1.6).

6.2.2 Variability data and security

The work about Reverse engineering Web configurators can also be revisited from a security point of view. The motivation is as follows: if it is straightforward or at least possible to reverse engineer variability information of a Web system, is not it a threat to the business of a company?

Products, options, and the underlying constraints a configurator is in charge of are key information of an organization. Such information is particularly interesting from the perspective of (online) *market intelligence* (also called *competitive intelligence*). Market intelligence can be defined as the "information relevant to a company's markets, gathered and analyzed specifically for the purpose of accurate and confident decision-making in determining market opportunity, market penetration strategy, and market development metrics." Lixto, a company offering data extraction tools and services, showed that it is technically feasible to acquire and exploit unstructured and semi-structured data in several case studies (*e.g.*, in the domain of computers and electronics consumer goods [44]).

Most information on pricing, product availability, product options, and product constraints is potentially available on Web sales configurators. Specifically, competitors can use this information (1) for getting a comprehensive overview of the options and constraints in the market; (2) to be (continually) informed about strengths and weaknesses of other competitors' product lines; (3) to publicly reveal a certain superiority or marketing practice, *etc.*

Web data extraction systems [44, 122] can be specialized for acquiring configurators' information. Our attempts in Section Reverse engineering Web configurators showed that it is indeed feasible. Static analysis techniques can locate templates of options and some constraints in a Web page. Combined with crawling techniques for deep navigation and dynamic content pages, there is the potential to comprehensively gather relevant information. In case the static and dynamic analysis of variability can be seamlessly realized, there is a risk for companies developing Web configurators to reveal trade secrets [48, 8].

6.2.3 Linux configurations and security

In Section Transfer learning across variants and versions: the case of Linux, we have targeted a specific property of Linux kernels: binary size. By no means our approach is restricted to this quantitative property. Learning techniques can be used to prevent and diagnose build failures of Linux configurations [24]. Also, other non-functional properties of configurations like build-time, ability to boot, boot time, and resilience to fuzzing are in our view. In addition, one could well observe security properties of Linux kernel configurations. Technically, it would require to extend TuxML to boot the kernel (in a virtual machine or in physical devices) and then automatically test security aspects at runtime. We have made recent progress to reuse some specific procedures of KernelCI. Overall I am confident we can gather a large number of configurations' measurements related to security. Then the challenge will be to identify whether specific (combinations of) Linux options are likely to increase or decrease security. An open question is whether learning techniques will be as effective as with binary size and produce accurate and/or interpretable models.

6.3 Smart Build of Software Configurations

The goal of this research direction is to develop what we call incremental build of configurations. Given a base configuration, we want to modify it (through the re-setting of some options values) and then build it without starting from scratch (*e.g.*, without a "make clean"). Similarly, we aim to build a given set of (random) configurations without starting from scratch each time. The promise is to dramatically reduce the cost of building software, a stressing topic when you think about the environmental and financial costs that companies and public organizations should have to bear. Society relies on software, but building software has an enormous cost: we aim to mitigate this trend.

The usual compilation and build process works quite well when small modifications are made (*e.g.*, modification of one source file), but building several configurations involve large modifications that span numerous source files. There are two extremes: (1) small modifications, with very low cost since the incremental compilation is fast (2) large modifications, with high cost since almost everything should be recompiled. In-between, we want to find a good trade off between diversity of the configurations and cost of compiling them.

In a sense, we want to explore the configuration space in a smart, efficient way. There are at least four research questions:

- **RQ1** Is incremental build of configurations safe? (*e.g.*, Do we obtain the same exact binary as with a standard compilation?)

- **RQ2** What is the gain of applying incremental compilation? The expected gain is here the time needed to build *e.g.*, the Linux kernels.

- **RQ3** Can we explore a diverse set of configurations with incremental builds?

- **RQ4** Is there a build strategy that reduces the cost of builds without trading diversity?

Several subject systems can be considered, with different languages, compilers, and build properties. A clear case is the Linux kernel. We have built 95K+ configurations (for version 4.13.3) with a high computational cost (8 minutes on average per configuration, and thus thousands of CPU machines). We believe incremental, smart build can save a large amount of resources since we can reuse shared compilations among configurations. Other systems, like JHipster or Chromium, are also targeted.

The ultimate goal is to integrate our idea in mainstream testing infrastructure (*e.g.*, KernelCI), for exploring further configurations at lower cost. The outcome of this research is to formulate the foundations of incremental build, invent new algorithms integrated into mainstream compilers and build systems, and assess the solution on widely used software projects.

6.4 Software Variability and Science

Software is the new *lingua franca*[2] of science. Biology, medicine, physics, astrophysics: all these scientific domains need to process large amount of data with more and more complex software-intensive pipelines. The promise of software is that a result obtained by an experiment can be achieved again with a high degree of agreement. Unfortunately, several studies reports that the same data analyzed with different software can lead to different results. For instance, applications of different analysis pipelines [188, 76], alterations in software version [126], and even changes in operating system [129] have both shown to cause variation in the results of a neuroimaging study. Similar observations have been made in the machine learning community [147].

I am seeing this problem as a manifestation of "deep software variability" (see Section 6.1): many factors (operating system, libraries, versions, inputs, the way the software is compiled, *etc.*) themselves subject to variability can alter the results, up to the point it can dramatically change the conclusions of some scientific studies. On a modest scale, Section Learning variability performance showed that a variation in dataset could change the conclusion about the effectiveness of a sampling strategy. In our study, it can arguably be seen as an empirical contribution but in general the complexity of modern software engineering studies can hide accidental threats. Besides, studies about climate modelling and change are based on complex software analysis operating over large data [160]. Unfortunately, the knowledge and recommendations built on top these simulations can be subject to variability threats. These are all refutable hypotheses worth looking at in the near future.

From a dissemination point of view, I want to raise awareness of the variability of some scientific, software-based results. I am convinced it is more and more urgent to develop critical thinking in society. A change in some software parameters (and thus underlying assumptions of a study) may radically change the conclusion and is a way to criticize or nuance an experiment. Here the variability problem would not be accidental: It is just explained by the assumptions you put in your study through software. I am planning to revisit existing studies and provide decision-making tools (*e.g.*, configurators) to play with such parameters on important society questions. That is, software variability will be a means to discuss tradeoffs and potentially explore alternatives.

As a final note, deep software variability also calls to consider ethical decisions and the role of machines in our society, since dramatic consequences can occur. What would you do if a system deciding on the release of a prisoner changes its prediction when slightly varying its executing environment? The brittleness of such systems and the trend to automate and abstract everything, without questioning the underlying "decisions" made by a machine, are threats to our society.

[2] We are at the end of the manuscript and it is time to exagerate (overclaim) a bit. From an etymological point of view, "lingua franca" is a Mediterranean Lingua Franca, used as the main language of commerce and diplomacy from late medieval times to the 18th century [100]. It is composed of Old French, Italian, Spanish as well as Arabic and Turkish and other languages. The terminology usage of lingua franca usually refers to a bridge or common language that can be make communication possible between groups of people (*e.g.*, scientists). Some people argue that mathematics is the lingua franca of science, others argue that it is English or TEX [125]. On the other hand, Paul C. Clements (BigLever) argued in a talk at SPLC 2015 that features are the lingua franca of software product line engineering [128]. I argue that software can play the role of a medium to support communication among scientists and the whole society. An important promise is the reproducibility that software is supposed to bring.

Abstract

The society expects software to deliver the right functionality, in a short amount of time and with fewer resources, in every possible circumstance whatever are the hardware, the operating systems, the compilers, or the data fed as input. For fitting such a diversity of needs, it is common that software comes in many variants and is highly configurable through configuration options, runtime parameters, conditional compilation directives, menu preferences, configuration files, plugins, etc. As there is no one-size-fits-all solution, software variability ("the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context") has been studied the last two decades and is a discipline of its own. Though highly desirable, software variability also introduces an enormous complexity due to the combinatorial explosion of possible variants. For example, the Linux kernel has 15000+ options and most of them can have 3 values: "yes", "no", or "module". Variability is challenging for maintaining, verifying, and configuring software systems (Web applications, Web browsers, video tools, etc.). It is also a source of opportunities to better understand a domain, create reusable artefacts, deploy performance-wise optimal systems, or find specialized solutions to many kinds of problems. In many scenarios, a model of variability is either beneficial or mandatory to explore, observe, and reason about the space of possible variants. For instance, without a variability model, it is impossible to establish a sampling strategy that would satisfy the constraints among options and meet coverage or testing criteria. I address a central question in this HDR manuscript: How to model software variability? I detail several contributions related to modelling, reverse engineering, and learning software variability.

I first contribute to support the persons in charge of manually specifying feature models, the de facto standard for modeling variability. I develop an algebra together with a language for supporting the composition, decomposition, diff, refactoring, and reasoning of feature models. I further establish the syntactic and semantic relationships between feature models and product comparison matrices, a large class of tabular data. I then empirically investigate how these feature models can be used to test in the large configurable systems with different sampling strategies. Along this effort, I report on the attempts and lessons learned when defining the "right" variability language. From a reverse engineering perspective, I contribute to synthesize variability information into models and from various kinds of artefacts. I develop foundations and methods for reverse engineering feature models from satisfiability formulae, product comparison matrices, dependencies files and architectural information, and from Web configurators. I also report on the degree of automation and show that the involvement of developers and domain experts is beneficial to obtain high-quality models. Thirdly, I contribute to learning constraints and non-functional properties (performance) of a variability-intensive system. I describe a systematic process "sampling, measuring, learning" that aims to enforce or augment a variability model, capturing variability knowledge that domain experts can hardly express. I show that supervised, statistical machine learning can be used to synthesize rules or build prediction models in an accurate and interpretable way. This process can even be applied to huge configuration space, such as the Linux kernel one.

Despite a wide applicability and observed benefits, I show that each individual line of contributions has limitations. I defend the following answer: a supervised, iterative process (1) based on the combination of reverse engineering, modelling, and learning techniques; (2) capable of integrating multiple variability information (*e.g.*, expert knowledge, legacy artefacts, dynamic observations).

Finally, this work opens different perspectives related to so-called deep software variability, security, smart build of configurations, and (threats to) science.

Bibliography

[1] I. Abal, C. Brabrand and A. Wasowski, '42 variability bugs in the linux kernel: A qualitative analysis', in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, Vasteras, Sweden: ACM, 2014, pp. 421–432, ISBN: 978-1-4503-3013-8 (cit. on pp. 51, 52).

[2] A. Abele, Y. Papadopoulos, D. Servat, M. Törngren and M. Weber, 'The CVM framework - A prototype tool for compositional variability management', in *Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, 2010, pp. 101–105. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf (cit. on pp. 68, 69).

[3] R. Abraham and M. Erwig, 'Type inference for spreadsheets', in *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '06, Venice, Italy: ACM, 2006, pp. 73–84, ISBN: 1-59593-388-3. DOI: 10.1145/1140335.1140346. http://doi.acm.org/10.1145/1140335.1140346 (cit. on p. 38).

[4] R. Abraham and M. Erwig, 'Ucheck: A spreadsheet type checker for end users', *J. Vis. Lang. Comput.*, vol. 18, no. 1, pp. 71–95, 2007 (cit. on p. 38).

[5] M. Acher, 'Learning the Linux Kernel Configuration Space: Results and Challenges', in *ELC Europe 2019 - Embedded Linux Conference Europe 2019*, Lyon, France, Oct. 2019, pp. 1–49. https://hal.inria.fr/hal-02342130 (cit. on p. 9).

[6] M. Acher, 'Managing Multiple Feature Models: Foundations, Language and Applications', 2011, p. 246 (cit. on pp. 31, 32).

[7] M. Acher, M. Alferez, J. A. Galindo, P. Romenteau and B. Baudry, 'ViViD: A Variability-Based Tool for Synthesizing Video Sequences', Anglais, in *18th International Software Product Line Conference (SPLC'14), tool track*, Florence, Italie, 2014. http://hal.inria.fr/hal-01020933 (cit. on pp. 10, 70, 114).

[8] M. Acher, G. Bécan, B. Combemale, B. Baudry and J.-M. Jézéquel, 'Product lines can jeopardize their trade secrets', in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, 2015, pp. 930–933 (cit. on pp. 157, 159).

[9] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien and P. Lahire, 'Extraction and Evolution of Architectural Variability Models in Plugin-based Systems', *Software and Systems Modeling (SoSyM)*, 2013 (cit. on pp. 96, 100, 102, 106).

[10] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien and P. Lahire, 'Reverse Engineering Architectural Feature Models', in *5th European Conference on Software Architecture (ECSA'11), long paper*, ser. LNCS, Essen (Germany): Springer, Sep. 2011, p. 16 (cit. on p. 96).

[11] M. Acher, A. Cleve, G. Perrouin, P. Heymans, P. Collet, P. Lahire and C. Vanbeneden, 'On Extracting Feature Models From Product Descriptions', in *Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, ser. VaMoS, Leipzig, Germany: ACM, Jan. 2012, p. 10. https://nyx.unice.fr/publis/acher-cleve-etal:2012.pdf (cit. on pp. 22, 96).

[12] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan and J.-P. Rigault, 'Modeling Context and Dynamic Adaptations with Feature Models', in *4th International Workshop Models@run.time at Models 2009 (MRT'09)*, Oct. 2009, p. 10 (cit. on p. 22).

[13] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat and R. France, 'Composing multiple variability artifacts to assemble coherent workflows', *Software Quality Journal (special issue: Quality Engineering for Software Product Lines)*, pp. 1–46, 2011 (cit. on p. 29).

[14] M. Acher, P. Collet, P. Lahire and R. France, 'Comparing Approaches to Implement Feature Model Composition', in *6th European Conference on Modelling Foundations and Applications (ECMFA)*, vol. LNCS, Springer, Jun. 2010, p. 16 (cit. on pp. 22, 25, 29).

[15] M. Acher, P. Collet, P. Lahire and R. France, 'Composing Feature Models', in *2nd International Conference on Software Language Engineering (SLE'09)*, ser. LNCS, LNCS, Oct. 2009, p. 20 (cit. on p. 29).

[16] M. Acher, P. Collet, P. Lahire and R. France, 'FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models', *Science of Computer Programming (SCP) Special issue on programming languages*, vol. 78, no. 6, pp. 657–681, 2013. DOI: {http://dx.doi.org/10.1016/j.scico.2012.12.004} (cit. on pp. 12, 20, 22, 30, 63, 68, 69).

[17] M. Acher, P. Collet, P. Lahire and R. France, 'Separation of Concerns in Feature Modeling: Support and Applications', in *Aspect-Oriented Software Development (AOSD'12)*, ser., ACM, Mar. 2012. http://hal.inria.fr/docs/00/76/74/23/PDF/acher-collet-etal-2012.pdf (cit. on p. 29).

[18] M. Acher, P. Collet, P. Lahire, S. Moisan and J.-P. Rigault, 'Modeling Variability from Requirements to Runtime', in *16th International Conference on Engineering of Complex Computer Systems (ICECCS'11)*, ser., Las Vegas: IEEE, Apr. 2011 (cit. on p. 22).

[19] M. Acher, B. Combemale and P. Collet, 'Metamorphic Domain-Specific Languages: A Journey Into the Shapes of a Language', Anglais, in *Onward! Essays (co-located with SPLASH and OOPSLA)*, Portland, États-Unis, Sep. 2014. DOI: 10.1145/2661136.2661159. http://hal.inria.fr/hal-01061576 (cit. on pp. 14, 15).

[20] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire and R. B. France, 'Composing your Compositions of Variability Models', in *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, 2013 (cit. on pp. 12, 20, 121).

[21] M. Acher, P. Heymans, A. Cleve, J.-L. Hainaut and B. Baudry, 'Support for reverse engineering and maintaining feature models', in *VaMoS'13*, ACM, 2013 (cit. on pp. 25, 29).

[22] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire and P. Merle, 'Feature model differences', in *CAiSE'12*, ser. LNCS, Springer, 2012, pp. 629–645 (cit. on p. 29).

[23] M. Acher, R. E. Lopez-Herrejon and R. Rabiser, 'A Survey on Teaching of Software Product Lines', Anglais, in *Eight International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'14)*, Nice, France: ACM, Jan. 2014. http://hal.inria.fr/hal-00916746 (cit. on p. 14).

[24] M. Acher, H. Martin, J. Alves Pereira, A. Blouin, D. Eddine Khelladi and J.-M. Jézéquel, 'Learning From Thousands of Build Failures of Linux Kernel Configurations', Inria ; IRISA, Technical Report, Jun. 2019, pp. 1–12. https://hal.inria.fr/hal-02147012 (cit. on pp. 151, 159).

[25] M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil and O. Barais, 'Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes', Inria Rennes - Bretagne Atlantique, Research Report, Oct. 2019. https://hal.inria.fr/hal-02314830 (cit. on pp. 12, 138, 140).

[26] M. Acher, P. Temple, J.-M. Jézéquel, J. A. Galindo Duarte, J. Martinez and T. Ziadi, 'VaryLaTeX: Learning Paper Variants That Meet Constraints', in *12th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'18)*, Madrid, Spain, Feb. 2018. https://hal.inria.fr/hal-01659161 (cit. on p. 10).

[27] M. Alférez, M. Acher, J. A. Galindo, B. Baudry and D. Benavides, 'Modeling Variability in the Video Domain: Language and Experience Report', *Software Quality Journal*, vol. 27, no. 1, pp. 307–347, 2019. DOI: 10.1007/s11219-017-9400-8. https://doi.org/10.1007/s11219-017-9400-8 (cit. on pp. 12, 61, 63, 64, 67, 68).

[28] J. Alves Pereira, M. Acher, H. Martin and J.-M. Jézéquel, 'Sampling Effect on Performance Prediction of Configurable Systems: A Case Study', in *International Conference on Performance Engineering (ICPE 2020)*, 2020. https://hal.inria.fr/hal-02356290 (cit. on pp. 12, 14, 131, 133, 136, 137, 156).

[29] J. Alves Pereira, H. Martin, M. Acher, J.-M. Jézéquel, G. Botterweck and A. Ventresque, 'Learning Software Configuration Spaces: A Systematic Literature Review', *Journal of Systems and Software (JSS)*, Jun. 2021. DOI: 10.1145/nnnnnnn.nnnnnnn (cit. on pp. 12, 14, 126, 131, 132, 134, 135, 138, 139, 145, 152).

[30] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl and A. Rummler, 'An exploratory study of information retrieval techniques in domain analysis', in *SPLC'08*, IEEE, 2008, pp. 67–76 (cit. on p. 82).

[31] B. Amand, M. Cordy, P. Heymans, M. Acher, P. Temple and J.-M. Jézéquel, 'Towards Learning-Aided Configuration in 3D Printing: Feasibility Study and Application to Defect Prediction', in *VaMoS 2019 - 13th International Workshop on Variability Modelling of Software-Intensive Systems*, Leuven, Belgium, Feb. 2019, pp. 1–9. https://hal.inria.fr/hal-01990767 (cit. on pp. 16, 135).

[32] N. Andersen, K. Czarnecki, S. She and A. Wasowski, 'Efficient synthesis of feature models', in *Proceedings of SPLC'12* (cit. on pp. 25, 82, 85, 91, 102).

[33] S. Apel, D. Batory, C. Kästner and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013 (cit. on pp. 6, 61, 75, 82, 158).

[34] S. Apel and C. Kästner, 'An overview of feature-oriented software development', *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, Jul. 2009 (cit. on p. 96).

[35] S. Apel, C. Lengauer, B. Moller and C. Kästner, 'An algebra for features and feature composition', in *12th Int'l Conference on Algebraic Methodology and Software Technology (AMAST)*, ser. LNCS, vol. 5140, Springer-Verlag, 2008, pp. 36–50. DOI: http://dx.doi.org/10.1007/978-3-540-79980-1_4 (cit. on p. 158).

[36] S. Apel, A. von Rhein, P. Wendler, A. Größlinger and D. Beyer, 'Strategies for Product-line Verification: Case Studies and Experiments', in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, Piscataway, NJ, USA: IEEE, 2013, pp. 482–491, ISBN: 978-1-4673-3076-3 (cit. on p. 54).

[37] A. Arcuri and L. Briand, 'Formal analysis of the probability of interaction fault detection using random testing', *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1088–1099, Sep. 2012, ISSN: 0098-5589 (cit. on pp. 55, 56).

[38] A. Arcuri and L. Briand, 'A practical guide for using statistical tests to assess randomized algorithms in software engineering', in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 1–10, ISBN: 978-1-4503-0445-0 (cit. on p. 135).

[39] E. Bagheri, F. Ensan and D. Gasevic, 'Decision support for the software product line domain engineering lifecycle', English, *Automated Software Engineering*, vol. 19, no. 3, pp. 335–377, 2012, ISSN: 0928-8910 (cit. on p. 82).

[40] K. Bak, K. Czarnecki and A. Wasowski, 'Feature and meta-models in clafer: Mixed, specialized, and coupled', in *SLE'10*, Eindhoven, The Netherlands, 2011 (cit. on pp. 63, 68–70, 75, 76).

[41] L. Bao, X. Liu, Z. Xu and B. Fang, 'Autoconfig: Automatic configuration tuning for distributed message systems', in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2018, pp. 29–40 (cit. on p. 135).

[42] M. Barreno, B. Nelson, R. Sears, A. D. Joseph and J. D. Tygar, 'Can machine learning be secure?', in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, Taipei, Taiwan: ACM, 2006, pp. 16–25 (cit. on pp. 126, 127).

[43] D. S. Batory, 'Feature models, grammars, and propositional formulas', in *SPLC'05*, ser. LNCS, vol. 3714, 2005, pp. 7–20 (cit. on pp. 29, 30).

[44] R. Baumgartner, G. Gottlob and M. Herzog, 'Scalable web data extraction for online market intelligence', *PVLDB*, vol. 2, no. 2, pp. 1512–1523, 2009. http://www.vldb.org/pvldb/2/vldb09-1075.pdf (cit. on p. 159).

[45] G. Bécan, 'Metamodels and feature models : complementary approaches to formalize product comparison matrices', Theses, Université Rennes 1, Sep. 2016. https://tel.archives-ouvertes.fr/tel-01416129 (cit. on pp. 15, 43).

[46] G. Bécan, M. Acher, B. Baudry and S. Ben Nasr, 'Breathing Ontological Knowledge Into Feature Model Management', Anglais, INRIA, Rapport Technique RT-0441, Oct. 2013, p. 15. http://hal.inria.fr/hal-00874867 (cit. on p. 102).

[47] G. Bécan, M. Acher, B. Baudry and S. Ben Nasr, 'Breathing Ontological Knowledge Into Feature Model Synthesis: An Empirical Study', *Empirical Software Engineering (ESE)*, vol. 21, no. 4, pp. 1794–1841, 2016. DOI: 10.1007/s10664-014-9357-1. https://hal.inria.fr/hal-01096969 (cit. on pp. 12, 20, 29, 30, 77, 91).

[48] G. Bécan, M. Acher, J.-M. Jézéquel and T. Menguy, 'On the Variability Secrets of an Online Video Generator', in *Variability Modelling of Software-intensive Systems (VaMoS'15)*, Hildesheim, Germany, Jan. 2015, pp. 96–102. DOI: 10.1145/2701319.2701328. https://hal.inria.fr/hal-01104797 (cit. on pp. 10, 157, 159).

[49] G. Bécan, R. Behjati, A. Gotlieb and M. Acher, 'Synthesis of Attributed Feature Models From Product Descriptions', in *19th International Software Product Line Conference (SPLC'15)*, (research track, long paper), Nashville, TN, USA, Jul. 2015 (cit. on pp. 12, 74, 76, 77, 79, 80, 91, 102).

[50] G. Bécan, R. Behjati, A. Gotlieb and M. Acher, 'Synthesis of Attributed Feature Models From Product Descriptions: Foundations', Inria Rennes, Rapport de Recherche RR-8680, Feb. 2015. https://hal.inria.fr/hal-01116663 (cit. on p. 79).

[51] G. Bécan, S. B. Nasr, M. Acher and B. Baudry, 'WebFML: Synthesizing Feature Models Everywhere', in *SPLC'14*, 2014 (cit. on p. 12).

[52] G. Bécan, N. Sannier, M. Acher, O. Barais, A. Blouin and B. Baudry, 'Automating the formalization of product comparison matrices', in *ASE*, 2014 (cit. on pp. 12, 33, 74).

[53] S. Ben Nasr, G. Bécan, M. Acher, J. B. Ferreira Filho, B. Baudry, N. Sannier and J.-M. Davril, 'Matrixminer: A red pill to architect informal product descriptions in the matrix', in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: ACM, 2015, pp. 982–985, ISBN: 978-1-4503-3675-8 (cit. on pp. 85, 87).

[54] S. Ben Nasr, G. Bécan, M. Acher, J. B. Ferreira Filho, N. Sannier, B. Baudry and J.-M. Davril, 'Automated Extraction of Product Comparison Matrices From Informal Product Descriptions', *Journal of Systems and Software (JSS)*, vol. 124, pp. 82–103, 2017. DOI: 10.1016/j.jss.2016.11.018. https://hal.inria.fr/hal-01427218 (cit. on pp. 12, 82).

[55] D. Benavides, S. Segura and A. Ruiz-Cortes, 'Automated analysis of feature models 20 years later: A literature review', *Information Systems*, vol. 35, no. 6, 2010 (cit. on pp. 27, 68, 75, 76).

[56] D. Benavides, S. Segura, P. Trinidad and A. R. Cortés, 'Fama: Tooling a framework for the automated analysis of feature models.', *VaMoS*, 2007 (cit. on p. 76).

[57] D. Benavides, P. Trinidad, A. R. Cortés and S. Segura, 'Fama', in *Systems and Software Variability Management*, 2013, pp. 163–171 (cit. on pp. 63, 68, 69).

[58] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki and A. Wasowski, 'Three cases of feature-based variability modeling in industry', in *MODELS*, 2014 (cit. on pp. 74, 75).

[59] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki and A. Wasowski, 'A survey of variability modeling in industrial practice', in *VaMoS'13*, 2013 (cit. on pp. 20, 61, 68, 70).

[60] T. Berger, S. She, R. Lotufo, A. Wasowski and K. Czarnecki, 'A study of variability models and languages in the systems software domain', *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, p. 1, 2013, ISSN: 0098-5589. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.34 (cit. on p. 70).

[61] B. Biggio, L. Didaci, G. Fumera and F. Roli, 'Poisoning attacks to compromise face templates', in *2013 International Conference on Biometrics (ICB)*, Madrid, Spain: IEEE, Jun. 2013, pp. 1–7. DOI: 10.1109/ICB.2013.6613006 (cit. on pp. 126, 127).

[62] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto and F. Roli, 'Evasion attacks against machine learning at test time', in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Dublin, Ireland: Springer Berlin, 2013, pp. 387–402 (cit. on pp. 126–128).

[63] B. Biggio, G. Fumera and F. Roli, 'Pattern recognition systems under attack: Design issues and research challenges', *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 28, no. 07, p. 1 460 002, 2014 (cit. on p. 127).

[64] B. Biggio, G. Fumera and F. Roli, 'Security evaluation of pattern classifiers under attack', *IEEE transactions on knowledge and data engineering*, vol. 26, no. 4, pp. 984–996, 2014 (cit. on pp. 126, 127).

[65] B. Biggio, B. Nelson and P. Laskov, 'Poisoning attacks against support vector machines', in *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ser. ICML'12, Edinburgh, Scotland: Omnipress, 2012, pp. 1467–1474, ISBN: 978-1-4503-1285-1. http://dl.acm.org/citation.cfm?id=3042573.3042761 (cit. on p. 127).

[66] B. Biggio and F. Roli, 'Wild patterns: Ten years after the rise of adversarial machine learning', *Pattern Recognition*, vol. 84, pp. 317–331, 2018 (cit. on pp. 126, 127).

[67] F. Bonin, F. Dell'Orletta, G. Venturi and S. Montemagni, 'A contrastive approach to multi-word term extraction from domain corpora', in *Proceedings of the "7th International Conference on Language Resources and Evaluation", Malta*, 2010, pp. 19–21 (cit. on p. 86).

[68] J. Bosch, 'Toward compositional software product lines', *IEEE Software*, vol. 27, pp. 29–34, 2010, ISSN: 0740-7459. DOI: http://doi.ieeecomputersociety.org/10.1109/MS.2010.32 (cit. on p. 22).

[69] M. Bošković, G. Mussbacher, E. Bagheri, D. Amyot, D. Gašević and M. Hatala, 'Aspect-oriented feature models', in *Proceedings of MODELS'10 workshops*, ser. MODELS'10, Oslo, Norway: Springer-Verlag, 2011, pp. 110–124, ISBN: 978-3-642-21209-3. http://dl.acm.org/citation.cfm?id=2008503.2008518 (cit. on p. 22).

[70] Q. Boucher, E. Abbasi, A. Hubaux, G. Perrouin, M. Acher and P. Heymans, 'Towards More Reliable Configurators: A Re-engineering Perspective', in *Third International Workshop on Product LinE Approaches in Software Engineering at ICSE 2012 (PLEASE'12)*, ser., Zurich, Jun. 2012 (cit. on p. 90).

[71] T. Brown, D. Mane, A. Roy, M. Abadi and J. Gilmer, 'Adversarial patch', *https://arxiv.org/pdf/1712.09665.pdf*, 2017 (cit. on p. 129).

[72] S. Buhne, K. Lauenroth and K. Pohl, 'Modelling requirements variability across product lines', in *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, Washington, DC, USA: IEEE Computer Society, 2005, pp. 41–52, ISBN: 0-7695-2425-7. DOI: http://dx.doi.org/10.1109/RE.2005.45 (cit. on p. 22).

[73] J. Carbonnel, 'L'analyse formelle de concepts: Un cadre structurel pour l'étude de la variabilité de familles de logiciels', PhD thesis, Université Montpellier, 2018 (cit. on p. 82).

[74] J. Carbonnel, M. Huchard, A. Miralles and C. Nebut, 'Feature model composition assisted by formal concept analysis', in *ENASE: Evaluation of Novel Approaches to Software Engineering*, SciTePress, 2017, pp. 27–37 (cit. on p. 82).

[75] J. Carbonnel, M. Huchard and C. Nebut, 'Modelling equivalence classes of feature models with concept lattices to assist their extraction from product descriptions', *Journal of Systems and Software*, vol. 152, pp. 1–23, 2019, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.02.027. https://www.sciencedirect.com/science/article/pii/S0164121219300378 (cit. on pp. 38, 82).

[76] J. Carp, 'On the plurality of (methodological) worlds: Estimating the analytic flexibility of fmri experiments', *Frontiers in neuroscience*, vol. 6, p. 149, 2012 (cit. on p. 161).

[77] M. Cashman, M. B. Cohen, P. Ranjan and R. W. Cottingham, 'Navigating the maze: The impact of configurability in bioinformatics software', in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 757–767. DOI: 10.1145/3238147.3240466. http://doi.acm.org/10.1145/3238147.3240466 (cit. on p. 9).

[78] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia and M. Y. Vardi, 'On parallel scalable uniform SAT witness generation', in *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 304–319 (cit. on pp. 55, 56, 134).

[79] S. Chakraborty, K. S. Meel and M. Y. Vardi, 'A scalable and nearly uniform generator of sat witnesses', in *International Conference on Computer Aided Verification*, Springer, 2013, pp. 608–623 (cit. on pp. 55, 56, 134).

[80] C. Chambers and M. Erwig, 'Automatic detection of dimension errors in spreadsheets', *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 269–283, Aug. 2009, ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2009.04.002. http://dx.doi.org/10.1016/j.jvlc.2009.04.002 (cit. on p. 38).

[81] K. Chen, W. Zhang, H. Zhao and H. Mei, 'An approach to constructing feature models based on requirements clustering', in *RE'05*, 2005, pp. 31–40, ISBN: 0-7695-2425-7. DOI: 10.1109/RE.2005.9 (cit. on pp. 82, 86, 135).

[82] E. J. Chikofsky and J. H. Cross II, 'Reverse engineering and design recovery: A taxonomy', *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, Jan. 1990 (cit. on p. 73).

[83] D. Clarke and J. Proenca, 'Towards a Theory of Views for Feature Models', in *Proceedings of the First Intl. Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010). Technical Report, University of Lancaster, U.K.*, vol. 2, Lancester University, Sep. 2010, pp. 91–100 (cit. on p. 22).

[84] A. Classen, Q. Boucher and P. Heymans, 'A text-based approach to feature modelling: Syntax and semantics of TVL', *Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability*, vol. 76, no. 12, pp. 1130–1143, 2011 (cit. on p. 69).

[85] A. Classen, Q. Boucher and P. Heymans, 'A text-based approach to feature modelling: Syntax and semantics of TVL', *Sci. Comput. Program.*, vol. 76, no. 12, 2011 (cit. on pp. 68, 76, 92).

[86] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay and J.-F. Raskin, 'Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking', *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1069–1089, Aug. 2013 (cit. on p. 45).

[87] M. Cohen, M. Dwyer and Jiangfan Shi, 'Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach', *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008 (cit. on pp. 44, 46, 112, 115, 134).

[88] M. Colmant, R. Rouvoy, M. Kurpicz, A. Sobe, P. Felber and L. Seinturier, 'The next 700 CPU power models', *J. Syst. Softw.*, vol. 144, pp. 382–396, 2018. DOI: 10.1016/j.jss.2018.07.001. https://doi.org/10.1016/j.jss.2018.07.001 (cit. on p. 156).

[89] Companion web page, *https://github.com/FAMILIAR-project/familiar-documentation/blob/master/manual/composition.md* (cit. on p. 30).

[90] M. Cordy, P.-Y. Schobbens, P. Heymans and A. Legay, 'Beyond boolean product-line model checking: Dealing with feature attributes and multi-features', in *ICSE*, 2013 (cit. on pp. 70, 75).

[91] J. Cunha, M. Erwig and J. Saraiva, 'Automatically inferring classsheet models from spreadsheets', in *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, Sep. 2010, pp. 93–100. DOI: 10.1109/VLHCC.2010.22 (cit. on p. 38).

[92] J. Cunha, J. P. Fernandes, H. Ribeiro and J. Saraiva, 'Towards a catalog of spreadsheet smells', in *Proceedings of the 12th International Conference on Computational Science and Its Applications - Volume Part IV*, ser. ICCSA'12, Salvador de Bahia, Brazil: Springer-Verlag, 2012, pp. 202–216, ISBN: 978-3-642-31127-7. DOI: 10.1007/978-3-642-31128-4_15. http://dx.doi.org/10.1007/978-3-642-31128-4_15 (cit. on p. 38).

[93] J. Cunha, J. Visser, T. L. Alves and J. Saraiva, 'Type-safe evolution of spreadsheets', in *FASE'11*, ser. LNCS, vol. 6603, 2011, pp. 186–201 (cit. on p. 38).

[94] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid and A. Wąsowski, 'Cool features and tough decisions: A comparison of variability modeling approaches', in *Proceedings of VaMoS'12*, Leipzig, Germany: ACM, 2012, pp. 173–182, ISBN: 978-1-4503-1058-1. DOI: 10.1145/2110147.2110167. http://doi.acm.org/10.1145/2110147.2110167 (cit. on p. 20).

[95] K. Czarnecki, S. Helsen and U. Eisenecker, 'Formalizing Cardinality-based Feature Models and their Specialization', in *Software Process Improvement and Practice*, 2005, pp. 7–29 (cit. on p. 158).

[96] K. Czarnecki, S. Helsen and U. Eisenecker, 'Staged configuration through specialization and multilevel configuration of feature models', *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005 (cit. on pp. 22, 158).

[97] K. Czarnecki, C. H. P. Kim and K. T. Kalleberg, 'Feature models are views on ontologies', in *SPLC*, 2006 (cit. on p. 75).

[98] K. Czarnecki and K. Pietroszek, 'Verifying feature-based model templates against well-formedness ocl constraints', in *GPCE'06*, ACM, 2006, pp. 211–220, ISBN: 1-59593-237-2. DOI: http://doi.acm.org.gate6.inist.fr/10.1145/1173706.1173738 (cit. on p. 96).

[99] K. Czarnecki and A. Wasowski, 'Feature diagrams and logics: There and back again', in *SPLC'07*, 2007 (cit. on pp. 21, 27, 29, 102).

[100] J. Dakhlia, 'Lingua franca. histoire d'une langue métisse en méditerranée', 2008 (cit. on p. 161).

[101] A. Darwiche and P. Marquis, 'A knowledge compilation map', *J. Artif. Intell. Res. (JAIR)*, vol. 17, pp. 229–264, 2002 (cit. on p. 29).

[102] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang and P. Heymans, 'Feature Model Extraction from Large Collections of Informal Product Descriptions', in *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, 2013, pp. 290–300 (cit. on pp. 12, 95).

[103] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang and P. Heymans, 'Feature model extraction from large collections of informal product descriptions', in *ESEC/FSE*, 2013 (cit. on pp. 82, 83, 85).

[104] O. Day and T. M. Khoshgoftaar, 'A survey on heterogeneous transfer learning', *Journal of Big Data*, vol. 4, no. 1, p. 29, 2017 (cit. on pp. 146–148).

[105] L. De Moura and N. Bjørner, 'Z3: An efficient smt solver', in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340 (cit. on p. 134).

[106] T. Degueule, J. B. F. Filho, O. Barais, M. Acher, J. Lenoir, O. Constant, S. Madelenat, G. Gailliard and G. Burlot, 'Tooling Support for Variability and Architectural Patterns in Systems Engineering', in *19th International Software Product Line Conference (SPLC'15)*, (demonstration and tool track), Nashville, TN, USA, Jul. 2015 (cit. on p. 15).

[107] A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru and F. Roli, 'On the intriguing connections of regularization, input gradients and transferability of evasion and poisoning attacks', *CoRR*, vol. abs/1809.02861, 2018. arXiv: 1809.02861. http://arxiv.org/abs/1809.02861 (cit. on p. 129).

[108] A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru and F. Roli, 'Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks', in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/demontis (cit. on p. 129).

[109] A. van Deursen and P. Klint, 'Domain-specific language design requires feature descriptions', *Journal of Computing and Information Technology*, vol. 10, no. 1, pp. 1–17, 2002 (cit. on pp. 68, 69).

[110] N. Dintzner, A. van Deursen and M. Pinzger, 'FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems', *EMSE*, vol. 23, no. 2, pp. 905–952, 2018. DOI: 10.1007/s10664-017-9557-6 (cit. on p. 142).

[111] H. Dohrn and D. Riehle, 'Design and implementation of the sweble wikitext parser: Unlocking the structured data of wikipedia', in *WikiSym'11*, ser. WikiSym '11, ACM, 2011, pp. 72–81 (cit. on p. 37).

[112] R. W. Dosselman and X. D. Yang, 'No-reference noise and blur detection via the fourier transform', University of Regina, CANADA, Tech. Rep., 2012 (cit. on p. 115).

[113] W. Dou, S.-C. Cheung and J. Wei, 'Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation', in *ICSE'14*, 2014 (cit. on p. 38).

[114] N. R. Draper and H. Smith, *Applied regression analysis*. John Wiley & Sons, 1998, vol. 326 (cit. on p. 140).

[115] C. Dumitrescu, R. Mazo, C. Salinesi and A. Dauron, 'Bridging the gap between product lines and systems engineering: An experience in variability management for automotive model based systems engineering', in *SPLC*, 2013, pp. 254–263 (cit. on p. 70).

[116] R. Dutra, K. Laeufer, J. Bachrach and K. Sen, 'Efficient sampling of SAT solutions for testing', in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 549–559. DOI: 10.1145/3180155.3180248. http://doi.acm.org/10.1145/3180155.3180248 (cit. on pp. 55–58).

[117] H. Eichelberger and K. Schmid, 'A systematic analysis of textual variability modeling languages', in *SPLC'13*, 2013. DOI: 10.1145/2491627.2491652. http://doi.acm.org/10.1145/2491627.2491652 (cit. on p. 68).

[118] H. Eichelberger and K. Schmid, 'Mapping the design-space of textual variability modeling languages: A refined analysis', English, *STTT*, pp. 1–26, 2014, ISSN: 1433-2779. DOI: 10.1007/s10009-014-0362-x. http://dx.doi.org/10.1007/s10009-014-0362-x (cit. on p. 76).

[119] G. Engels and M. Erwig, 'Classsheets: Automatic generation of spreadsheet applications from object-oriented specifications', in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, Long Beach, CA, USA: ACM, 2005, pp. 124–133, ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101929. http://doi.acm.org/10.1145/1101908.1101929 (cit. on p. 38).

[120] S. Ermon, C. P. Gomes, A. Sabharwal and B. Selman, 'Embed and project: Discrete sampling with universal hashing', in *Advances in Neural Information Processing Systems*, 2013, pp. 2085–2093 (cit. on p. 56).

[121] S. Ermon, C. P. Gomes and B. Selman, 'Uniform solution sampling using a constraint solver as an oracle', *arXiv preprint arXiv:1210.4861*, 2012 (cit. on p. 56).

[122] E. Ferrara, P. D. Meo, G. Fiumara and R. Baumgartner, 'Web data extraction, applications and techniques: A survey', *Knowledge-Based Systems*, vol. 70, no. 0, pp. 301–323, 2014 (cit. on p. 159).

[123] A. Ferrari, G. O. Spagnolo and F. dell'Orletta, 'Mining commonalities and variabilities from natural language documents', in *SPLC*, 2013 (cit. on p. 82).

[124] J. A. Galindo, M. Alferez, M. Acher, B. Baudry and D. Benavides, 'A variability-based testing approach for synthesizing video sequences', in *International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014 (cit. on pp. 65, 112, 114, 115).

[125] E. Ghys, *La lingua franca des mathématiciens https://images.math.cnrs.fr/la-lingua-franca-des-mathematiciens.html*, 2014 (cit. on p. 161).

[126] T. Glatard, L. B. Lewis, R. Ferreira da Silva, R. Adalat, N. Beck, C. Lepage, P. Rioux, M.-E. Rousseau, T. Sherif, E. Deelman *et al.*, 'Reproducibility of neuroimaging analyses across operating systems', *Frontiers in neuroinformatics*, vol. 9, p. 12, 2015 (cit. on pp. 156, 161).

[127] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, 'Generative adversarial nets', in *Advances in neural information processing systems*, 2014, pp. 2672–2680 (cit. on pp. 126, 128).

[128] S. P. Gregg, R. Scharadin and P. Clements, 'The more you do, the more you save: The superlinear cost avoidance effect of systems product line engineering', in *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, D. C. Schmidt, Ed., ACM, 2015, pp. 303–310. DOI: 10.1145/2791060.2791065. https://doi.org/10.1145/2791060.2791065 (cit. on p. 161).

[129] E. H. Gronenschild, P. Habets, H. I. Jacobs, R. Mengelers, N. Rozendaal, J. Van Os and M. Marcelis, 'The effects of freesurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements', *PloS one*, vol. 7, no. 6, e38234, 2012 (cit. on p. 161).

[130] P. Grünbacher, R. Rabiser, D. Dhungana and M. Lehofer, 'Model-Based Customization and Deployment of Eclipse-Based Tools: Industrial Experiences', in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 247–256, ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.11. http://dx.doi.org/10.1109/ASE.2009.11 (cit. on p. 96).

[131] J. Guo, K. Czarnecki, S. Apel, N. Siegmund and A. Wasowski, 'Variability-aware performance prediction: A statistical learning approach', in *ASE*, 2013 (cit. on pp. 124, 131, 138).

[132] M. Hahsler, B. Grün and K. Hornik, 'Arules – A computational environment for mining association rules and frequent item sets', *Journal of Statistical Software*, vol. 14, no. 15, pp. 1–25, Oct. 2005, ISSN: 1548-7660 (cit. on p. 49).

[133] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau and G. Saake, 'IncLing: efficient product-line testing using incremental pairwise sampling', in *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences - GPCE 2016*, ACM, 2016, pp. 144–155 (cit. on p. 52).

[134] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau and G. Saake, 'Similarity-based prioritization in software product-line testing', in *SPLC'14* (cit. on pp. 112, 115).

[135] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin and B. Baudry, 'Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack', *Empirical Software Engineering (ESE)*, vol. 24, no. 2, pp. 674–717, Jul. 2019, Empirical Software Engineering journal. DOI: 10.07980. https://doi.org/10.1007/s10664-018-9635-4 (cit. on pp. 9, 12, 16, 44–47, 58).

[136] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin and P. Heymans, 'Yo Variability! JHipster: A Playground for Web-Apps Analyses', in *11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'17)*, Eindhoven, Netherlands, Feb. 2017, pp. 44–51. DOI: 10.1145/3023956.3023963. https://hal.inria.fr/hal-01468084 (cit. on pp. 16, 51).

[137] S. O. Hallsteinsen, M. Hinchey, S. Park and K. Schmid, 'Dynamic software product lines', *IEEE Computer*, vol. 41, no. 4, pp. 93–95, 2008. DOI: 10.1109/MC.2008.123. http://dx.doi.org/10.1109/MC.2008.123 (cit. on p. 9).

[138] N. Hariri, C. Castro-Herrera, M. Mirakhorli, J. Cleland-Huang and B. Mobasher, 'Supporting domain analysis through mining and recommending features from online product listings', *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, p. 1, 2013, ISSN: 0098-5589. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.39 (cit. on pp. 82, 85).

[139] H. Hartmann and T. Trew, 'Using feature diagrams with context variability to model multiple product lines for software supply chains', in *SPLC'08*, IEEE, 2008, pp. 12–21 (cit. on pp. 22, 121).

[140] H. Hartmann, T. Trew and A. Matsinger, 'Supplier independent feature modelling', in *SPLC'09*, IEEE, 2009, pp. 191–200 (cit. on pp. 22, 29, 30, 121).

[141] E. N. Haslinger, R. E. Lopez-Herrejon and A. Egyed, 'Reverse engineering feature models from programs' feature sets', in *WCRE'11*, IEEE, 2011, pp. 308–312 (cit. on p. 96).

[142] M. Heinz, R. Lämmel and M. Acher, 'Discovering Indicators for Classifying Wikipedia Articles in a Domain: A Case Study on Software Languages', in *SEKE 2019 - The 31st International Conference on Software Engineering and Knowledge Engineering*, Lisbonne, Portugal, Jul. 2019, pp. 1–6. https://hal.inria.fr/hal-02129131 (cit. on p. 13).

[143] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans and Y. Le Traon, 'Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines', *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014 (cit. on pp. 44, 46, 52).

[144] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans and Y. L. Traon, 'Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines', *IEEE Trans. Software Eng.*, 2014 (cit. on pp. 112, 115, 134).

[145] C. Henard, M. Papadakis, G. Perrouin, J. Klein and Y. Le Traon, 'Towards automated testing and fixing of re-engineered feature models', in *ICSE '13 (NIER track)*, 2013 (cit. on p. 95).

[146] C. Henard, M. Papadakis, G. Perrouin, J. Klein and Y. L. Traon, 'Pledge: A product line editor and test generation tool', in *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, ser. SPLC '13 Workshops, Tokyo, Japan: ACM, 2013, pp. 126–129, ISBN: 978-1-4503-2325-3 (cit. on p. 52).

[147] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup and D. Meger, 'Deep reinforcement learning that matters', in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018 (cit. on p. 161).

[148] D. Hendry and T. Green, 'Creating, comprehending and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model', *International Journal of Human-Computer Studies*, vol. 40, no. 6, pp. 1033–1065, 1994, ISSN: 1071-5819. DOI: 10.1006/ijhc.1994.1047 (cit. on p. 38).

[149] K. Heo, W. Lee, P. Pashakhanloo and M. Naik, 'Effective program debloating via reinforcement learning', in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: ACM, 2018, pp. 380–394, ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243838. http://doi.acm.org/10.1145/3243734.3243838 (cit. on p. 158).

[150] F. Hermans, M. Pinzger and A. v. Deursen, 'Detecting and visualizing inter-worksheet smells in spreadsheets', in *ICSE'12*, IEEE, 2012, pp. 441–451, ISBN: 978-1-4673-1067-3 (cit. on p. 38).

[151] F. Hermans, M. Pinzger and A. van Deursen, 'Automatically extracting class diagrams from spreadsheets', in *ECOOP'10*, ser. LNCS, vol. 6183, Springer-Verlag, 2010, pp. 52–75 (cit. on p. 38).

[152] F. Hermans, M. Pinzger and A. van Deursen, 'Detecting code smells in spreadsheet formulas', in *ICSM*, IEEE, 2012, pp. 409–418, ISBN: 978-1-4673-2313-0 (cit. on p. 38).

[153] F. Hermans, M. Pinzger and A. van Deursen, 'Supporting professional spreadsheet users by generating leveled dataflow diagrams', in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 451–460, ISBN: 978-1-4503-0445-0 (cit. on p. 38).

[154] A. Hervieu, B. Baudry and A. Gotlieb, 'PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models', in *IEEE 22nd International Symposium on Software Reliability Engineering - ISSRE '11*, IEEE, 2011, pp. 120–129 (cit. on pp. 44, 46).

[155] G. Holl, P. Grünbacher and R. Rabiser, 'A systematic review and an expert survey on capabilities supporting multi product lines', *Information and Software Technology*, vol. 54, no. 8, pp. 828–852, Aug. 2012, ISSN: 09505849. DOI: 10.1016/j.infsof.2012.02.002 (cit. on p. 22).

[156] http://www.bestbuy.com, *Bestbuy*, 2014 (cit. on pp. 85, 88).

[157] https://kernelnewbies.org/Linux_5.4, *TKernelNewbies: Linux_5.4*, Nov. 2019 (cit. on p. 143).

[158] https://tiny.wiki.kernel.org/, *Linux kernel tinification*, last access: july 2019 (cit. on p. 138).

[159] https://www.phoronix.com/scan.php?page=article&item=linux-416-54&num=1, *The Disappointing Direction Of Linux Performance From 4.16 To 5.4 Kernels*, Nov. 2019 (cit. on p. 143).

[160] https://www.wcrp-climate.org/, *World climat research program (wcrp)* (cit. on p. 161).

[161] M. Hu and B. Liu, 'Mining and summarizing customer reviews', in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2004, pp. 168–177 (cit. on p. 86).

[162] A. Hubaux, M. Acher, T. T. Tun, P. Heymans, P. Collet and P. Lahire, 'Domain engineering: Product lines, conceptual models, and languages (editors: Reinhartz-berger,i. and sturm, a. and clark, t. and bettin, j. and cohen, s.)', in. Springer, 2013, ch. Separating Concerns in Feature Models: Retrospective and Multi-View Support (cit. on p. 22).

[163] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder and E. K. Abbasi, 'Supporting multiple perspectives in feature-based configuration', *Software and Systems Modeling*, pp. 1–23, 2011 (cit. on pp. 22, 158).

[164] S. Ida and S. Ketil, 'Technology research explained', Tech. Rep., 2007 (cit. on p. 66).

[165] N. Itzik and I. Reinhartz-Berger, 'SOVA - A tool for semantic and ontological variability analysis', in *Joint Proceedings of the CAiSE 2014 Forum and CAiSE 2014 Doctoral Consortium*, 2014, pp. 177–184. http://ceur-ws.org/Vol-1164/PaperDemo06.pdf (cit. on p. 82).

[166] S. Iyengar, *The Art of Choosing*. Twelve, 2010 (cit. on p. 82).

[167] P. Jamshidi, N. Siegmund, M. Velez, A. Patel and Y. Agarwal, 'Transfer learning for performance modeling of configurable systems: An exploratory analysis', in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Press, 2017, pp. 497–508 (cit. on pp. 135, 156).

[168] P. Jamshidi, M. Velez, C. Kästner and N. Siegmund, 'Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems', in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2018, pp. 71–82 (cit. on pp. 145, 147).

[169] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund and P. Kawthekar, 'Transfer learning for improving model predictions in highly configurable software', in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Buenos Aires: IEEE Computer Society, May 2017, pp. 31–41. DOI: http://dx.doi.org/10.1109/SEAMS.2017.11 (cit. on p. 9).

[170] JHipsterTeam, *JHipster website*, https://jhipster.tech, accessed Feb. 2021., 2021. https://jhipster.tech (cit. on p. 130).

[171] D. Jin, X. Qu, M. B. Cohen and B. Robinson, 'Configurations everywhere: implications for testing and debugging in practice', in *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, ACM, 2014, pp. 215–224 (cit. on p. 45).

[172] M. F. Johansen, Ø. Haugen and F. Fleurey, 'An algorithm for generating t-wise covering arrays from large feature models', in *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*, vol. 1, ACM, 2012, p. 46 (cit. on pp. 51, 112, 115, 134).

[173] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo and S. Apel, 'Distance-based sampling of software configuration spaces', in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019 (cit. on pp. 131–136, 138).

[174] C. Kaner, J. Bach and B. Pettichord, *Lessons Learned in Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 2001, ISBN: 0471081124 (cit. on p. 124).

[175] K. Kang, S. Cohen, J. Hess, W. Novak and S. Peterson, 'Feature-Oriented Domain Analysis (FODA)', SEI, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990 (cit. on pp. 68, 69).

[176] C. Kästner, A. Dreiling and K. Ostermann, 'Variability mining: Consistent semiautomatic detection of product-line features', *IEEE Transactions on Software Engineering*, 2013, (to appear) (cit. on p. 82).

[177] *Kernelci*. https://kernelci.org/ (cit. on p. 151).

[178] E. Khalil Abbasi, M. Acher, P. Heymans and A. Cleve, 'Reverse Engineering Web Configurators', in *17th European Conference on Software Maintenance and Reengineering (CSMR'14)*, IEEE, Ed., Antwerp, Belgium, Feb. 2014 (cit. on pp. 12, 90, 93, 95).

[179] E. Khalil Abbasi, A. Hubaux, M. Acher, Q. Boucher and P. Heymans, 'The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases', Anglais, in *25th International Conference on Advanced Information Systems Engineering (CAiSE'13)*, M. Norrie and C. Salinesi, Eds., Valencia, Espagne, Jun. 2013. http://hal.inria.fr/hal-00796555 (cit. on pp. 9, 12, 90, 92, 93).

[180] C. H. P. Kim, D. S. Batory and S. Khurshid, 'Reducing combinatorics in testing product lines', in *AOSD'11* (cit. on p. 44).

[181] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros and M. d'Amorim, 'Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems - esec/fse '13', in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ACM, 2013, pp. 257–267 (cit. on p. 44).

[182] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG] (cit. on p. 142).

[183] N. Kitchen and A. Kuehlmann, 'Stimulus generation for constrained random simulation', in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, IEEE Press, 2007, pp. 258–265 (cit. on p. 56).

[184] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke and I. Schaefer, '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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 291–302. DOI: 10.1145/3106237.3106252. http://doi.acm.org/10.1145/3106237.3106252 (cit. on pp. 56, 57).

[185] S. Kolesnikov, N. Siegmund, C. Kästner, A. Grebhahn and S. Apel, 'Tradeoffs in modeling performance of highly configurable software systems', *Software & Systems Modeling*, vol. 18, no. 3, pp. 2265–2283, Jun. 2019, ISSN: 1619-1374. DOI: 10.1007/s10270-018-0662-9. https://doi.org/10.1007/s10270-018-0662-9 (cit. on pp. 135, 141).

[186] H. Koo, S. Ghavamnia and M. Polychronakis, 'Configuration-driven software debloating', in *Proceedings of the 12th European Workshop on Systems Security*, ser. EuroSec '19, Dresden, Germany: ACM, 2019, 9:1–9:6, ISBN: 978-1-4503-6274-0. DOI: 10.1145/3301417.3312501. http://doi.acm.org/10.1145/3301417.3312501 (cit. on p. 158).

[187] J. Kramer, 'Is abstraction the key to computing?', *Commun. ACM*, vol. 50, no. 4, pp. 36–42, Apr. 2007, ISSN: 0001-0782. DOI: 10.1145/1232743.1232745. https://doi.org/10.1145/1232743.1232745 (cit. on p. 19).

[188] D. Krefting, M. Scheel, A. Freing, S. Specovius, F. Paul and A. Brandt, 'Reliability of quantitative neuroimage analysis using freesurfer in distributed environments', in *MICCAI Workshop on High-Performance and Distributed Computing for Medical Imaging.(Toronto, ON)*, 2011 (cit. on p. 161).

[189] S. Krieter, T. Thüm, S. Schulze, R. Schröter and G. Saake, 'Propagating configuration decisions with modal implication graphs', in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 898–909. DOI: 10.1145/3180155.3180159. http://doi.acm.org/10.1145/3180155.3180159 (cit. on pp. 56, 57).

[190] R. Krishna, V. Nair, P. Jamshidi and T. Menzies, 'Whence to learn? transferring knowledge in configurable systems using beetle', *IEEE Transactions on Software Engineering*, pp. 1–1, 2020 (cit. on pp. 147, 149).

[191] T. Krismayer, R. Rabiser and P. Grünbacher, 'Mining constraints for event-based monitoring in systems of systems', in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Press, 2017, pp. 826–831 (cit. on p. 135).

[192] W. H. Kruskal and W. A. Wallis, 'Use of ranks in one-criterion variance analysis', *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952 (cit. on p. 135).

[193] D. Kuhn, D. Wallace and A. Gallo, 'Software fault interactions and implications for software testing', *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, Jun. 2004, ISSN: 0098-5589 (cit. on p. 134).

[194] A. Kurmus, A. Sorniotti and R. Kapitza, 'Attack surface reduction for commodity os kernels: Trimmed garden plants may attract less bugs', in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11, Salzburg, Austria: ACM, 2011, 6:1–6:6, ISBN: 978-1-4503-0613-3. DOI: 10.1145/1972551.1972557. http://doi.acm.org/10.1145/1972551.1972557 (cit. on p. 158).

[195] J. Lawall and G. Muller, 'Coccinelle: 10 years of automated evolution in the linux kernel', in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 601–614 (cit. on p. 142).

[196] J. Lawall and G. Muller, 'Jmake: Dependable compilation for kernel janitors', in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, 2017, pp. 357–366. DOI: 10.1109/DSN.2017.62 (cit. on p. 142).

[197] D. Le Berre and A. Parrain, 'The sat4j library, release 2.2', *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010 (cit. on pp. 80, 134).

[198] J. Le Noir, S. Madelénat, C. Labreuche, O. Constant, G. Gailliard, M. Acher and O. Barais, 'A Decision-making Process for Exploring Architectural Variants in Systems Engineering', in *Software Product Lines Conference (SPLC)*, Beijing, China, Sep. 2016. DOI: 10.1145/1235. https://hal.inria.fr/hal-01374140 (cit. on p. 16).

[199] L. Lesoil, M. Acher, A. Blouin and J.-M. Jézéquel, 'Deep Software Variability: Towards Handling Cross-Layer Configuration', in *VaMoS 2021 - 15th International Working Conference on Variability Modelling of Software-Intensive Systems*, Krems / Virtual, Austria, Feb. 2021. https://hal.inria.fr/hal-03084276 (cit. on pp. 155, 156).

[200] L. Lesoil, M. Acher, X. Tërnava, A. Blouin and J.-M. Jézéquel, 'The Interplay of Compile-time and Run-time Options for Performance Prediction', in *SPLC '21 - 25th ACM International Systems and Software Product Line Conference - Volume A*, Leicester, United Kingdom, Sep. 2021. DOI: 10.1145/3461001.3471149. https://hal.archives-ouvertes.fr/hal-03286127 (cit. on pp. 12, 156).

[201] H. Levene, 'Robust tests for equality of variances', *Contributions to probability and statistics. Essays in honor of Harold Hotelling*, pp. 279–292, 1961 (cit. on p. 136).

[202] J. H. Liang, V. Ganesh, K. Czarnecki and V. Raman, 'Sat-based analysis of large real-world feature models is easy', in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15, Nashville, Tennessee: ACM, 2015, pp. 91–100, ISBN: 978-1-4503-3613-0. DOI: 10.1145/2791060.2791070. http://doi.acm.org/10.1145/2791060.2791070 (cit. on pp. 56, 57).

[203] M. Lillack, J. Müller and U. W. Eisenecker, 'Improved prediction of non-functional properties in software product lines with domain context', *Software Engineering 2013*, 2013 (cit. on p. 135).

[204] R. E. Lopez-Herrejon and A. Egyed, 'On the need of safe software product line architectures', in *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, ser. LNCS, vol. 6285, Springer, 2010, pp. 493–496 (cit. on p. 96).

[205] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura and A. Egyed, 'An assessment of search-based techniques for reverse engineering feature models', *Journal of Systems and Software*, 2014, ISSN: 0164-1212 (cit. on p. 85).

[206] H. B. Mann and D. R. Whitney, 'On a test of whether one of two random variables is stochastically larger than the other', *The annals of mathematical statistics*, pp. 50–60, 1947 (cit. on p. 135).

[207] M. Mannion, J. Savolainen and T. Asikainen, 'Viewpoint-oriented variability modeling', in *Proceedings of the 33rd International Computer Software and Applications Conference (COMPSAC'09)*, IEEE, 2009, pp. 67–72, ISBN: 978-0-7695-3726-9. DOI: http://dx.doi.org/10.1109/COMPSAC.2009.19 (cit. on p. 22).

[208] H. Martin, M. Acher, J. A. Pereira, L. Lesoil, J. Jézéquel and D. E. Khelladi, 'Transfer learning across variants and versions: The case of linux kernel size', *Transactions on Software Engineering (TSE)*, 2021 (cit. on pp. 12, 138, 151).

[209] H. Martin, J. A. Pereira, M. Acher and J. Jézéquel, 'A comparison of performance specialization learning for configurable systems', in *SPLC '21: 25th ACM International Systems and Software Product Line Conference*, ACM, 2021 (cit. on pp. 12, 123, 158).

[210] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller and R. Marlet, 'Specialization tools and techniques for systematic optimization of system software', *ACM Trans. Comput. Syst.*, vol. 19, no. 2, pp. 217–251, 2001. DOI: 10.1145/377769.377778. http://doi.acm.org/10.1145/377769.377778 (cit. on p. 158).

[211] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi and S. Apel, 'A comparison of 10 sampling algorithms for configurable systems', in *ICSE'16*, 2016 (cit. on pp. 44, 46, 51–54, 60, 112, 115).

[212] J. Meinicke, C.-p. Wong, C. Kästner, T. Thüm and G. Saake, 'On essential configuration complexity: measuring interactions in highly-configurable systems', in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, Singapore, Singapore: ACM Press, 2016, pp. 483–494 (cit. on pp. 44, 46).

[213] J. Melo, E. Flesborg, C. Brabrand and A. Wasowski, 'A quantitative analysis of variability warnings in linux', in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16, Salvador, Brazil: ACM, 2016, pp. 3–8, ISBN: 978-1-4503-4019-9 (cit. on p. 140).

[214] M. Mendonca and D. Cowan, 'Decision-making coordination and efficient reasoning techniques for feature-based configuration', *Science of Computer Programming*, vol. 75, no. 5, pp. 311–332, 2010 (cit. on p. 22).

[215] M. Mendonca, A. Wasowski, K. Czarnecki and D. Cowan, 'Efficient compilation techniques for large scale feature models', in *Int'l Conference on Generative programming and component engineering*, 2008, pp. 13–22 (cit. on p. 134).

[216] M. Mendonça, M. Branco and D. D. Cowan, 'S.p.l.o.t.: Software product lines online tools', in *OOPSLA Companion*, 2009, pp. 761–762 (cit. on pp. 68, 69).

[217] P. e. a. Merle, *OW2 FraSCAti Web Site*, http://frascati.ow2.org, 2008 (cit. on p. 97).

[218] A. Mesbah, A. van Deursen and S. Lenselink, 'Crawling ajax-based web applications through dynamic analysis of user interface state changes', *ACM Trans. Web*, vol. 6, no. 1, 3:1–3:30, Mar. 2012, ISSN: 1559-1131. DOI: 10.1145/2109205.2109208 (cit. on p. 91).

[219] H. Miyashita, H. Tai and S. Amano, 'Controlled modeling environment using flexibly-formatted spreadsheets', in *ICSE'14*, 2014 (cit. on p. 38).

[220] C. Molnar, *Interpretable Machine Learning*. Lulu. com, 2020 (cit. on p. 145).

[221] B. Morin, O. Barais, J. Jézéquel, F. Fleurey and A. Solberg, 'Models@ run.time to support dynamic adaptation', *IEEE Computer*, vol. 42, no. 10, pp. 44–51, 2009. DOI: 10.1109/MC.2009.327. http://dx.doi.org/10.1109/MC.2009.327 (cit. on p. 9).

[222] S. Mühlbauer, S. Apel and N. Siegmund, 'Identifying software performance changes across variants and versions', in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 611–622 (cit. on p. 142).

[223] A. Murashkin, M. Antkiewicz, D. Rayside and K. Czarnecki, 'Visualization and exploration of optimal variants in product line engineering', in *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, T. Kishi, S. Jarzabek and S. Gnesi, Eds., ACM, 2013, pp. 111–115. DOI: 10.1145/2491627.2491647. https://doi.org/10.1145/2491627.2491647 (cit. on pp. 82, 85).

[224] G. C. Murphy, D. Notkin and K. J. Sullivan, 'Software reflexion models: Bridging the gap between design and implementation', *IEEE Trans. Softw. Eng.*, vol. 27, no. 4, pp. 364–380, Apr. 2001, ISSN: 0098-5589. DOI: 10.1109/32.917525. http://dx.doi.org/10.1109/32.917525 (cit. on p. 96).

[225] I. M. Murwantara, B. Bordbar and L. L. Minku, 'Measuring energy consumption for web service product configuration', in *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, ser. iiWAS, Hanoi, Viet Nam: ACM, 2014, pp. 224–228, ISBN: 978-1-4503-3001-5 (cit. on p. 135).

[226] S. Nadi, T. Berger, C. Kästner and K. Czarnecki, 'Mining configuration constraints: Static analyses and empirical results', in *ICSE*, Hyderabad, 2014 (cit. on p. 82).

[227] V. Nair, T. Menzies, N. Siegmund and S. Apel, 'Using bad learners to find good configurations', in *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 257–267 (cit. on p. 135).

[228] V. Nair, Z. Yu, T. Menzies, N. Siegmund and S. Apel, 'Finding faster configurations using flash', *IEEE Transact. on Software Engineering*, 2018 (cit. on p. 135).

[229] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. Rubinstein, U. Saini, C. A. Sutton, J. D. Tygar and K. Xia, 'Exploiting machine learning to subvert your spam filter.', *LEET*, vol. 8, pp. 1–9, 2008 (cit. on p. 126).

[230] S. Nestorov, S. Abiteboul and R. Motwani, 'Inferring structure in semistructured data', *SIGMOD Rec.*, vol. 26, no. 4, pp. 39–43, Dec. 1997, ISSN: 0163-5808. DOI: 10.1145/271074.271084. http://doi.acm.org/10.1145/271074.271084 (cit. on p. 39).

[231] H. V. Nguyen, C. Kästner and T. N. Nguyen, 'Exploring variability-aware execution for testing plugin-based web applications', in *Proceedings of the 36th International Conference on Software Engineering - ICSE '14*, ACM, 2014, pp. 907–918 (cit. on pp. 44, 45).

[232] N. Niu and S. M. Easterbrook, 'Concept analysis for product line requirements', in *AOSD'09*, K. J. Sullivan, A. Moreira, C. Schwanninger and J. Gray, Eds., ACM, 2009, pp. 137–148, ISBN: 978-1-60558-442-3 (cit. on p. 82).

[233] OASIS, *Service Component Architecture*, http://www.oasis-opencsa.org/sca/, 2007 (cit. on p. 97).

[234] J. Oh, D. S. Batory, M. Myers and N. Siegmund, 'Finding near-optimal configurations in product lines by random sampling', in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 61–71 (cit. on p. 134).

[235] S. Oh, A. Hoogs, A. Perera, N. Cuntoor, C.-C. Chen, J. T. Lee, S. Mukherjee, J. K. Aggarwal, H. Lee, L. Davis, E. Swears, X. Wang, Q. Ji, K. Reddy, M. Shah, C. Vondrick, H. Pirsiavash, D. Ramanan, J. Yuen, A. Torralba, B. Song, A. Fong, A. Roy-Chowdhury and M. Desai, 'A large-scale benchmark dataset for event recognition in surveillance video', in *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 3153–3160, ISBN: 978-1-4577-0394-2. DOI: 10.1109/CVPR.2011.5995586. http://dx.doi.org/10.1109/CVPR.2011.5995586 (cit. on p. 66).

[236] R. Olaechea, D. Rayside, J. Guo and K. Czarnecki, 'Comparison of exact and approximate multi-objective optimization for software product lines', in *SPLC'14*, 2014, pp. 92–101 (cit. on pp. 75, 85).

[237] M. Opdenacker, *Bof: Embedded linux size*, Embedded Linux Conference North-America, 2018 (cit. on p. 138).

[238] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, J. Penhoat and L. Seinturier, 'Taming energy consumption variations in systems benchmarking', in *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, J. N. Amaral, A. Koziolek, C. Trubiani and A. Iosup, Eds., ACM, 2020, pp. 36–47. DOI: 10.1145/3358960.3379142. https://doi.org/10.1145/3358960.3379142 (cit. on p. 156).

[239] S. J. Pan and Q. Yang, 'A survey on transfer learning', *TKDE*, vol. 22, no. 10, pp. 1345–1359, 2009 (cit. on p. 146).

[240] R. R. Panko, 'Thinking is bad: Implications of human error research for spreadsheet research and practice', *CoRR*, vol. abs/0801.3114, 2008 (cit. on p. 38).

[241] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon and A. Egyed, 'Multi-objective test case prioritization in highly configurable systems: A case study', *Journal of Systems and Software*, vol. 122, pp. 287–310, 2016 (cit. on p. 54).

[242] J. R. Parker, *Algorithms for image processing and computer vision*. Wiley. com, 2010 (cit. on p. 62).

[243] C. A. Parra, A. Cleve, X. Blanc and L. Duchien, 'Feature-based composition of software architectures', in *ECSA'10*, ser. LNCS, vol. 6285, Springer, 2010, pp. 230–245 (cit. on p. 96).

[244] L. T. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba and J. Guo, 'Coevolution of variability models and related software artifacts - A fresh look at evolution patterns in the linux kernel', *Empirical Soft. Eng.*, vol. 21, no. 4, pp. 1744–1793, 2016. DOI: 10.1007/s10664-015-9364-x (cit. on p. 142).

[245] B. Pérez Lamancha and M. Polo Usaola, 'Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage', in *Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Proceedings*, A. Petrenko, A. Simão and J. C. Maldonado, Eds. Springer, 2010, pp. 111–125 (cit. on pp. 112, 115).

[246] G. Perrouin, S. Sen, J. Klein, B. Baudry and Y. l. Traon, 'Automated and scalable t-wise test case generation strategies for software product lines', in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 459–468, ISBN: 978-0-7695-3990-4 (cit. on p. 51).

[247] J. Petke, S. Yoo, M. B. Cohen and M. Harman, 'Efficiency and early fault detection with lower and higher strength combinatorial interaction testing', in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: ACM, 2013, pp. 26–36, ISBN: 978-1-4503-2237-9 (cit. on p. 51).

[248] Q. Plazar, M. Acher, S. Bardin and A. Gotlieb, 'Efficient and Complete FD-Solving for Extended Array Constraints', in *IJCAI 2017*, Melbourne, Australia, Aug. 2017. https://hal.archives-ouvertes.fr/hal-01545557 (cit. on p. 15).

[249] Q. Plazar, M. Acher, G. Perrouin, X. Devroey and M. Cordy, 'Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?', in *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*, Xian, China, Apr. 2019, pp. 1–12. https://hal.inria.fr/hal-01991857 (cit. on pp. 12, 14, 15, 44, 57, 58).

[250] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer and S. Kowalewski, 'Model-driven support for product line evolution on feature level', *Journal of Systems and Software*, vol. 85, no. 10, pp. 2261–2274, 2012 (cit. on p. 96).

[251] K. Pohl, G. Böckle and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005, ISBN: 3540243720 (cit. on pp. 6, 22, 61, 82).

[252] J. Ponce, D. Forsyth, E.-p. Willow, S. Antipolis-Méditerranée, R. d'activité-RAweb, L. Inria and I. Alumni, 'Computer vision: A modern approach', *Computer*, vol. 16, p. 11, 2011 (cit. on p. 62).

[253] A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt and B. Natarajan, 'Skoll: A process and infrastructure for distributed continuous quality assurance', *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 510–525, 2007 (cit. on p. 135).

[254] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein *et al.*, 'Measuring the impact of spectre and meltdown', in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, IEEE, 2018, pp. 1–5 (cit. on p. 143).

[255] R. Queiroz, T. Berger and K. Czarnecki, 'Towards predicting feature defects in software product lines', in *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, ACM, 2016, pp. 58–62 (cit. on p. 135).

[256] C. Quinton, D. Romero and L. Duchien, 'Cardinality-based feature models with constraints: A pragmatic approach', in *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, 2013, pp. 162–166. DOI: 10.1145/2491627.2491638. http://doi.acm.org/10.1145/2491627.2491638 (cit. on pp. 68, 69).

[257] A. Rabkin and R. Katz, 'Static extraction of program configuration options', in *ICSE'11*, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 131–140, ISBN: 978-1-4503-0445-0 (cit. on p. 96).

[258] M. Raible, *The JHipster mini-book*. C4Media, 2015 (cit. on p. 45).

[259] I. Reinhartz-Berger, 'Can domain modeling be automated? levels of automation in domain modeling', in *SPLC '14*, 2014, p. 359 (cit. on p. 82).

[260] I. Reinhartz-Berger, A. Sturm and Y. Wand, 'Comparing functionality of software systems: An ontological approach', *Data Knowl. Eng.*, vol. 87, pp. 320–338, 2013. DOI: 10.1016/j.datak.2012.09.005. http://dx.doi.org/10.1016/j.datak.2012.09.005 (cit. on p. 82).

[261] M.-O. Reiser and M. Weber, 'Multi-level feature trees: A pragmatic approach to managing highly complex product families', *Requir. Eng.*, vol. 12, no. 2, pp. 57–75, 2007 (cit. on pp. 22, 29).

[262] X. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm and D. Yuan, 'An analysis of performance evolution of linux's core operations', in *Proceedings of the 27th ACM SOSP*, 2019, pp. 554–569 (cit. on pp. 142, 143).

[263] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer and T. Berger, 'Presence-condition simplification in highly configurable systems', in *ICSE*, 2015 (cit. on p. 120).

[264] T. Rogoll and F. Piller, 'Product configuration from the customer's perspective: A comparison of configuration systems in the apparel industry', in *PETO'04*, 2004 (cit. on p. 90).

[265] M. Rosenmüller, N. Siegmund, T. Thüm and G. Saake, 'Multi-dimensional variability modeling', in *VaMoS'11*, ACM, 2011, pp. 11–20 (cit. on p. 22).

[266] M. Rosenmüller, N. Siegmund, T. Thüm and G. Saake, 'Multi-dimensional variability modeling', in *VaMoS*, 2011, pp. 11–20 (cit. on pp. 68, 69).

[267] H. Samih, H. Le Guen, R. Bogusch, M. Acher and B. Baudry, 'An Approach to Derive Usage Models Variants for Model-based Testing', Anglais, in *26th IFIP International Conference on Testing Software and Systems (ICTSS'2014)*, Madrid, Espagne: Springer, Sep. 2014. http://hal.inria.fr/hal-01025124 (cit. on p. 15).

[268] F. Samreen, Y. Elkhatib, M. Rowe and G. S. Blair, 'Daleel: Simplifying cloud instance selection using machine learning', in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2016, pp. 557–563 (cit. on p. 135).

[269] A. B. Sánchez, S. Segura, J. A. Parejo and A. Ruiz-Cortés, 'Variability testing in the wild: The drupal case study', *Software & Systems Modeling*, vol. 16, no. 1, pp. 173–194, 2017, ISSN: 1619-1374 (cit. on pp. 44, 54).

[270] A. B. Sanchez, S. Segura and A. Ruiz-Cortes, 'A Comparison of Test Case Prioritization Criteria for Software Product Lines', in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation - ICST*, IEEE, 2014, pp. 41–50 (cit. on pp. 44, 46).

[271] N. Sannier, M. Acher and B. Baudry, 'From Comparison Matrix to Variability Model: The Wikipedia Case Study', in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, Palo Alto, USA, 2013. http://hal.inria.fr/hal-00858491 (cit. on pp. 12, 13, 15, 33).

[272] N. Sannier, G. Bécan, M. Acher, S. Ben Nasr and B. Baudry, 'Comparing or Configuring Products: Are We Getting the Right Ones?', Anglais, in *8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14)*, A. Wasowski and T. Weyer, Eds., Nice, France: ACM, Jan. 2014 (cit. on p. 33).

[273] A. Sarkar, J. Guo, N. Siegmund, S. Apel and K. Czarnecki, 'Cost-efficient sampling for performance prediction of configurable systems (t)', in *ASE'15*, 2015 (cit. on pp. 112, 115, 131, 135, 138).

[274] J. Savolainen, M. Raatikainen and T. Männistö, 'Eight practical considerations in applying feature modeling for product lines', in *ICSR*, 2011, pp. 192–206 (cit. on pp. 61, 65, 66).

[275] M. Sayagh, N. Kerzazi, B. Adams and F. Petrillo, 'Software configuration engineering in practice: Interviews, survey, and systematic literature review', *IEEE Transactions on Software Engineering*, 2018 (cit. on p. 9).

[276] I. Schaefer, L. Bettini, V. Bono, F. Damiani and N. Tanzarella, 'Delta-oriented programming of software product lines', in *International Conference on Software Product Lines*, Springer, 2010, pp. 77–91 (cit. on p. 6).

[277] P.-Y. Schobbens, P. Heymans and J.-C. Trigaux, 'Feature diagrams: A survey and a formal semantics', in *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Washington, DC, USA: IEEE Computer Society, 2006, pp. 136–145, ISBN: 0-7695-2555-5. DOI: http://dx.doi.org/10.1109/RE.2006.23 (cit. on p. 68).

[278] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux and Y. Bontemps, 'Generic semantics of feature diagrams', *Comput. Netw.*, vol. 51, no. 2, pp. 456–479, 2007 (cit. on p. 22).

[279] J. Schroeter, M. Lochau and T. Winkelmann, 'Multi-perspectives on feature models', in *MoDELS'12*, ser. LNCS, vol. 7590, 2012, pp. 252–268 (cit. on p. 22).

[280] U. P. Schultz, J. L. Lawall and C. Consel, 'Automatic program specialization for java', *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 452–499, Jul. 2003, ISSN: 0164-0925. DOI: 10.1145/778559.778561. http://doi.acm.org/10.1145/778559.778561 (cit. on p. 158).

[281] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni and J.-B. Stefani, 'Reconfigurable SCA Applications with the FraSCAti Platform', in *Proceedings of the 2009 IEEE International Conference on Services Computing (SCC 2009)*, IEEE, 2009, pp. 268–275 (cit. on p. 97).

[282] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni and J.-B. Stefani, 'A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures', *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, May 2012. DOI: 10.1002/spe.1077. http://hal.inria.fr/inria-00567442 (cit. on p. 97).

[283] S. Sepúlveda, A. Cravero and C. Cachero, 'Requirements modeling languages for software product lines: A systematic literature review', *Information & Software Technology*, vol. 69, pp. 16–36, 2016. DOI: 10.1016/j.infsof.2015.08.007. https://doi.org/10.1016/j.infsof.2015.08.007 (cit. on p. 69).

[284] S. She, R. Lotufo, T. Berger, A. Wasowski and K. Czarnecki, 'Reverse engineering feature models', in *ICSE'11*, 2011 (cit. on pp. 21, 25, 77, 91, 95, 96, 102).

[285] S. She, U. Ryssel, N. Andersen, A. Wasowski and K. Czarnecki, 'Efficient synthesis of feature models', *Information and Software Technology*, vol. 56, no. 9, 2014 (cit. on pp. 77, 80).

[286] J. Siegmund, N. Siegmund and S. Apel, 'Views on internal and external validity in empirical software engineering', in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora and S. G. Elbaum, Eds., IEEE Computer Society, 2015, pp. 9–19. DOI: 10.1109/ICSE.2015.24. https://doi.org/10.1109/ICSE.2015.24 (cit. on p. 14).

[287] N. Siegmund, A. Grebhahn, C. Kästner and S. Apel, 'Performance-influence models for highly configurable systems', in *ESEC/FSE'15*, 2015 (cit. on pp. 131, 135, 138).

[288] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller and G. Saake, 'Predicting performance via automated feature-interaction detection', in *International Conference on Software Engineering (ICSE)*, 2012, pp. 167–177 (cit. on p. 134).

[289] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel and S. S. Kolesnikov, 'Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption', *Inf. Softw. Technol.*, vol. 55, no. 3, pp. 491–507, 2013. DOI: 10.1016/j.infsof.2012.07.020. https://doi.org/10.1016/j.infsof.2012.07.020 (cit. on p. 138).

[290] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel and S. S. Kolesnikov, 'Scalable prediction of non-functional properties in software product lines', in *Software Product Line Conference (SPLC), 2011 15th International*, 2011, pp. 160–169 (cit. on p. 138).

[291] J. Sincero, W. Schroder-Preikschat and O. Spinczyk, 'Approaching non-functional properties of software product lines: Learning from products', in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 2010, pp. 147–155 (cit. on p. 138).

[292] G. W. Snedecor and W. G. Cochran, 'Statistical methods, 8thedn', *Ames: Iowa State Univ. Press Iowa*, 1989 (cit. on p. 136).

[293] C. Song, A. Porter and J. S. Foster, 'Itree: Efficiently discovering high-coverage configurations using interaction trees', *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 251–265, 2013 (cit. on p. 135).

[294] S. Souto, M. D'Amorim and R. Gheyi, 'Balancing Soundness and Efficiency for Practical Testing of Configurable Systems', in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 632–642 (cit. on p. 54).

[295] M. Svahnberg, J. van Gurp and J. Bosch, 'A taxonomy of variability realization techniques: Research articles', *Softw. Pract. Exper.*, vol. 35, no. 8, pp. 705–754, 2005, ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.v35:8 (cit. on p. 6).

[296] P. Temple, M. Acher and J.-M. Jézéquel, 'Empirical Assessment of Multimorphic Testing', *IEEE Transactions on Software Engineering (TSE)*, pp. 1–21, Jul. 2019. DOI: 10.1109/TSE.2019.2926971. https://hal.inria.fr/hal-02177158 (cit. on pp. 13, 14, 123).

[297] P. Temple, M. Acher, J.-M. A. Jézéquel, L. A. Noel-Baron and J. A. Galindo, 'Learning-Based Performance Specialization of Configurable Systems', IRISA, Inria Rennes ; University of Rennes 1, Research Report, Feb. 2017. https://hal.archives-ouvertes.fr/hal-01467299 (cit. on p. 124).

[298] P. Temple, M. Acher, J.-M. Jézéquel and O. Barais, 'Learning-Contextual Variability Models', *IEEE Software*, vol. 34, no. 6, pp. 64–70, Nov. 2017. https://hal.inria.fr/hal-01659137 (cit. on pp. 9, 12, 110, 135, 158).

[299] P. Temple, J. A. Galindo Duarte, M. Acher and J.-M. Jézéquel, 'Using Machine Learning to Infer Constraints for Product Lines', in *Software Product Line Conference (SPLC'16)*, Beijing, China, Sep. 2016. DOI: 10.1145/2934466.2934472. https://hal.inria.fr/hal-01323446 (cit. on pp. 12, 110, 135).

[300] P. Temple, G. Perrouin, M. Acher, B. Biggio, J.-M. Jézéquel and F. Roli, 'Empirical Assessment of Generating Adversarial Configurations for Software Product Lines', *Empirical Software Engineering (ESE)*, Nov. 2020 (cit. on pp. 12, 124, 130).

[301] C. Thornton, F. Hutter, H. H. Hoos and K. Leyton-Brown, 'Auto-weka: Combined selection and hyperparameter optimization of classification algorithms', in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2013, pp. 847–855 (cit. on p. 135).

[302] T. Thüm, S. Apel, C. Kästner, I. Schaefer and G. Saake, 'A classification and survey of analysis strategies for software product lines', *ACM Computing Surveys*, 2014 (cit. on pp. 44, 45, 75, 134).

[303] T. Thüm, D. Batory and C. Kästner, 'Reasoning about edits to feature models', in *ICSE'09*, Vancouver, Canada: ACM, 2009, pp. 254–264 (cit. on pp. 27, 112).

[304] T. Thüm, C. Kästner, S. Erdweg and N. Siegmund, 'Abstract features in feature modeling', in *SPLC'11*, Munich: IEEE, Aug. 2011, pp. 191–200 (cit. on p. 107).

[305] L. Torvalds, 'The linux edge', *Communications of the ACM*, vol. 42, no. 4, pp. 38–38, 1999 (cit. on p. 138).

[306] A. Trentin, E. Perin and C. Forza, 'Sales configurator capabilities to prevent product variety from backfiring', in *Workshop on Configuration (ConfWS)*, Montpellier, France, 2012 (cit. on p. 90).

[307] S. Trujillo, C. Kästner and S. Apel, 'Product lines that supply other product lines: A service-oriented approach', in *SPLC Workshop: Service-Oriented Architectures and Product Lines - What is the Connection?*, Kyoto, Japan, Sep. 2007 (cit. on p. 22).

[308] E. Vacchi, W. Cazzola, B. Combemale and M. Acher, 'Automating Variability Model Inference for Component-Based Language Implementations', Anglais, in *18th International Software Product Line Conference (SPLC'14)*, P. Heymans and J. Rubin, Eds., Florence, Italie: ACM, Sep. 2014. http://hal.inria.fr/hal-01023864 (cit. on p. 108).

[309] P. Valov, J. Guo and K. Czarnecki, 'Empirical comparison of regression methods for variability-aware performance prediction', in *SPLC'15*, 2015 (cit. on p. 135).

[310] P. Valov, J. Guo and K. Czarnecki, 'Empirical comparison of regression methods for variability-aware performance prediction', in *SPLC'15* (cit. on p. 124).

[311] P. Valov, J. Petkovich, J. Guo, S. Fischmeister and K. Czarnecki, 'Transferring performance prediction models across different hardware platforms', in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, 2017, pp. 39–50. DOI: 10.1145/3030207.3030216. http://doi.acm.org/10.1145/3030207.3030216 (cit. on p. 124).

[312] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister and K. Czarnecki, 'Transferring performance prediction models across different hardware platforms', in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ACM, 2017, pp. 39–50 (cit. on pp. 135, 145, 147, 149).

[313] D. Van Landuyt, S. Op De Beeck, A. Hovsepyan, S. Michiels, W. Joosen, S. Meynckens, G. De Jong, O. Barais and M. Acher, 'Towards Managing Variability in the Safety Design of an Automotive Hall Effect Sensor', Anglais, in *18th International Software Product Line Conference (SPLC'14), industrial track*, Florence, Italie, Jul. 2014. http://hal.inria.fr/hal-01018938 (cit. on p. 16).

[314] A. Vargha and H. D. Delaney, 'A critique and improvement of the cl common language effect size statistics of mcgraw and wong', *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000 (cit. on p. 135).

[315] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi and I. Schaefer, 'A classification of product sampling for software product lines', in *Proceeedings of the 22nd International Conference on Systems and Software Product Line - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, 2018, pp. 1–13. DOI: 10.1145/3233027.3233035. http://doi.acm.org/10.1145/3233027.3233035 (cit. on pp. 60, 134).

[316] V. Weber, 'Utfm - a next generation language and tool for feature modeling', PhD thesis, Faculty of Electrical Engineering, Mathematics and Computer Science of the University of Twente, Aug. 2014. http://essay.utwente.nl/65854/ (cit. on pp. 68, 69).

[317] M. Weckesser, R. Kluge, M. Pfannemüller, M. Matthé, A. Schürr and C. Becker, 'Optimal reconfiguration of dynamic software product lines based on performance-influence models', in *International Systems and Software Product Line Conference (SPLC)*, ACM, 2018, pp. 98–109 (cit. on p. 135).

[318] W. Wei, J. Erenrich and B. Selman, 'Towards efficient sampling: Exploiting random walk strategies', in *AAAI*, vol. 4, 2004, pp. 670–676 (cit. on p. 56).

[319] K. Weiss, T. M. Khoshgoftaar and D. Wang, 'A survey of transfer learning', *Journal of Big data*, vol. 3, no. 1, p. 9, 2016 (cit. on p. 146).

[320] D. Westermann, J. Happe, R. Krebs and R. Farahbod, 'Automated inference of goal-oriented performance prediction functions', in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2012, pp. 190–199 (cit. on p. 135).

[321] N. Weston, R. Chitchyan and A. Rashid, 'A framework for constructing semantically composable feature models from natural language requirements', in *SPLC'09*, ACM, 2009, pp. 211–220 (cit. on p. 96).

[322] D. Wille, S. Holthusen, S. Schulze and I. Schaefer, 'Interface variability in family model mining', in *Proceedings of the 17th International Software Product Line Conference co-located workshops*, 2013, pp. 44–51 (cit. on p. 96).

[323] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy and R. Talwadker, 'Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software', in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 307–319. DOI: 10.1145/2786805.2786852. http://doi.acm.org/10.1145/2786805.2786852 (cit. on p. 9).

[324] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou and S. Pasupathy, 'Do not blame users for misconfigurations', in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 244–259, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522727. http://doi.acm.org/10.1145/2517349.2522727 (cit. on p. 157).

[325] C. Yilmaz, M. B. Cohen and A. A. Porter, 'Covering arrays for efficient fault characterization in complex configuration spaces', *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006 (cit. on p. 135).

[326] L. A. Zaid, F. Kleinermann and O. D. Troyer, 'Feature assembly: A new feature modeling technique', in *Conceptual Modeling (ER'10)*, 2010, pp. 233–246 (cit. on p. 29).

[327] W. Zhang, H. Yan, H. Zhao and Z. Jin, 'A bdd-based approach to verifying clone-enabled feature models'constraints and customization', *High Confidence Software Reuse in Large Systems*, pp. 186–199, 2008. http://dx.doi.org/10.1007/978-3-540-68073-4_18 (cit. on p. 70).

[328] Y. Zhang, J. Guo, E. Blais, K. Czarnecki and H. Yu, 'A mathematical model of performance-relevant feature interactions', in *International Systems and Software Product Line Conference (SPLC)*, ACM, 2016, pp. 25–34 (cit. on p. 135).

[329] W. Zheng, R. Bianchini and T. D. Nguyen, 'Automatic configuration of internet services', *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 219–229, 2007 (cit. on p. 135).

[330] T. Ziadi, L. Frias, M. A. A. da Silva and M. Ziane, 'Feature identification from the source code of product variants', in *CSMR*, T. Mens, A. Cleve and R. Ferenc, Eds., IEEE, 2012, pp. 417–422, ISBN: 978-1-4673-0984-4 (cit. on p. 91).