Variability Modeling in the Real
An Empirical Journey from Software Product Lines
to Software Ecosystems

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

D I S S E R T A T I O N

zur Erlangung des akademischen Grades
DOCTOR RERUM NATURALIUM
(Dr. rer. nat.)

im Fachgebiet
Informatik

vorgelegt

von Dipl.-Inf. Thorsten Berger
geboren am 26. Oktober 1981 in Wurzen

Die Annahme der Dissertation wurde empfohlen von:

1. Prof. Dr.-Ing. Klaus-Peter Fähnrich (Universität Leipzig)
2. Prof. Dr. Paul Grünbacher (Universität Linz)

Die Verleihung des akademischen Grades erfolgt mit Bestehen der
Verteidigung am 16.04.2013 mit dem Gesamtprädikat magna cum laude.
Berger, Thorsten:
Variability Modeling in the Real:
An Empirical Journey from Software Product Lines to Software Ecosystems
Dissertation, University of Leipzig, 2012
Abstract

Software product lines are among the most successful approaches to intra-organizational reuse of software. Product line engineering allows companies to efficiently create portfolios of systems in an application domain by leveraging commonality and managing the variability, that is the differences, among the systems. However, the platforms of large product lines have complex variability that imposes a significant challenge to their development.

Variability modeling is one of the key disciplines to cope with this complexity. Variability modeling aims at creating, evolving, and configuring variability models, which describe the common and variable characteristics, also known as features, of products in a product line. Since the introduction of feature models more than twenty years ago, many variability modeling languages and notations have been proposed both in academia and industry, followed by hundreds of publications on variability modeling techniques that have built upon these theoretical foundations.

Surprisingly, there are relatively few empirical studies that aim at understanding the actual use of such languages, leading to speculations and ad-hoc assumptions in literature. What variability modeling concepts are actually used in practice? Do variability models applied in real-world look similar to those published in literature? Or in what technical and organizational contexts are variability models applicable?

We present an empirical study on variability modeling that addresses this major gap in software product line research. Our goals are i) to verify existing theoretical research, and ii) to explore real-world variability modeling languages and models expressed in them. Therefore, we study concepts and semantics of variability modeling languages conceived by practitioners, and the usage of these concepts in real, large-scale variability models. Our aim is to support variability modeling research by providing empirical evidence for the actual use of its core modeling concepts, by identifying and characterizing further concepts that have not been addressed in the literature, and by providing realistic assumptions about scale, structure, content, and complexity of real-world variability models. We believe that our findings are of relevance to variability modeling researchers and tool designers, for example, those working on interactive product configurators or feature dependency checkers. Our extracted models provide realistic benchmarks that can be used to evaluate new techniques.

Recognizing the recent trend in software engineering to open up software platforms to facilitate inter-organizational reuse of software, we extend our empirical discourse to the emerging field of software ecosystems. As natural successors of successful software product lines, software ecosystems manage huge variability among and within their software assets, thus, represent a highly interesting class of systems to study variability modeling concepts and mechanisms. Our studied systems comprise eleven highly configurable software systems and two ecosystems with closed platforms and three software ecosystems relying on open platforms. Some of our subjects are among the largest successful systems in existence today. Results from a survey on industrial variability modeling complement these subjects.

Our overall results provide empirical evidence that the well-researched concepts of feature modeling are used in practice, but also that more advanced concepts are needed. We observe that assumptions about realistic variability models in the literature do not hold. Our study also reveals that variability models—while providing
system-wide abstractions over code—work best in centralized variability management scenarios and that they are fragile and have to be controlled by a small team. Among all subjects, we also identify a particular type of dependencies (capability-based dependencies), which are increasingly used in open platforms and help sustain the growth of ecosystems. Interestingly, while enabling distributed variability, these dependencies rely on a centralized and stable vocabulary. Finally, the studies allow us to formulate new hypotheses and research questions that provide direction for future research.
Acknowledgements

Being intrigued by the fact that I almost “made it”, I am looking back to a couple of years with very interesting, but also very intensive work on my dissertation project. This endeavor would not have been possible without the support of many important colleagues, friends, and family members, who all helped and believed in me in many different ways. First of all, I would like to express a particular gratitude to my parents for their patient support and encouragement to pursue my studies and doctoral research.

Many thanks go to my dissertation supervisor Klaus-Peter Fähnrich for giving me the opportunity to do my dissertation at his Chair of Business Information Systems, for always supporting me with advice and encouraging me in my work, for example, by showing trust in me and my abilities to follow my research objectives while being provided with freedom to follow my research agenda. I would also like to thank Stefan Kühne, team lead of our research group for his support, and all my colleagues for a very nice time and a lot of discussions that sharpened my research objectives and how to present them. In particular, I would like to thank Thomas Riechert, whom I met long time ago during my studies, who supervised my Diploma thesis and sparked my interest in software product line research. Another colleague, Steffen Dienst, has always been a close friend since high school, and a great guy to work with.

In 2009, I had the chance to visit Krzysztof Czarnecki and his GSD lab at the University of Waterloo, which was the beginning of a great collaboration, resulting in a lot of collaborative publications. I am very thankful to him for teaching me a lot about conducting research in the field of software product lines and variability modeling. His sharp and direct guidance on core publications have helped to significantly advance my dissertation project. The same holds for Andrzej Wąsowski from the ITU Copenhagen, who also invited me to his research group and always had an “open ear” when I was stuck with my research. I am also very proud that Ulrich Eisenecker has supported me with feedback and many discussions on my research since I first met him in the context of my Diploma thesis.

A special thanks goes to all my co-authors I had the pleasure to work with. I would like to thank Steven She, with whom it was great to perform a lot of collaborative research. His abilities to write efficient, clean, and very structured, both imperative and functional code, has inspired me a lot. I was happy to work with Rafael Lotufo, Rolf-Helge Pfeiffer, Reinhard Tartler, Leonardo Passos, Marko Novakovic, Yingfei Xiong, Christian Kästner (and his colleagues in Marburg), Ahmed Hassan, Bram Adams, Mei Nagappan, and Israel Mojica. I’m further very thankful to feedback on my research provided by Sven Apel, Klaus Schmid, Kyo Kang, and many more who I met at many occasions, such as the Software Product Line conference, or the VaMoS workshop.

For my current ongoing research, a study on industrial variability modeling, I am very
Acknowledgements

thankful to my Diploma student Ralf Rublack, the PhD student Divya Nair, Jo Atlee and Martin Becker for great collaboration.

This dissertation has been funded by scholarships. I am very happy to have been awarded a scholarship from the German National Academic Foundation, which supports 0.5% of Germany’s students, and provided additional non-monetary support. I also acknowledge their allowances for my research visits. I have also received a project-bound scholarship from the Institute of Applied Informatics at the University of Leipzig (by grants from the Federal Ministry of Education and Research). Furthermore, the German Academic Exchange Service provided me with a conference travel allowance.

Last, but not least, I have to acknowledge many personal skills due to the dissertation work itself. I significantly improved the way I am thinking about problems and how to tackle them. I believe that I also became a bit more open-minded to unfamiliar research fields, and that I sharpened my mind on how to present ideas, and how to bring problems to the point. Finally, I learned to become productive in many different environments, including trains, buses, and aircrafts.
Contents

Abstract iii
Acknowledgements v

1. Introduction 1
   1.1. Trends and Motivation ........................................ 2
       1.1.1. Software Product Lines ................................ 2
       1.1.2. Software Ecosystems .................................... 4
       1.1.3. Empirical Software Engineering ....................... 5
   1.2. Problem Statement ........................................... 6
       1.2.1. Empirical Data on Variability Modeling ............... 6
       1.2.2. Applicability of Variability Models ................. 7
   1.3. Research Hypothesis, Questions, and Methodology .......... 8
   1.4. Contributions ............................................... 11
   1.5. Guide for Readers ............................................ 12
   1.6. Bibliographical Notes ....................................... 13

2. Background 17
   2.1. Software Product Lines ...................................... 17
       2.1.1. Examples of Software Product Lines .................. 19
       2.1.2. Software Product Line Architectures ................. 19
   2.2. Variability Modeling ......................................... 22
       2.2.1. Variability Model Semantics ........................... 24
       2.2.2. Configuration Process ................................ 24
   2.3. Feature Models ............................................... 25
       2.3.1. The Notion of a Feature ................................ 25
       2.3.2. Feature Model Semantics ............................... 26
       2.3.3. Feature Model Extensions .............................. 31
       2.3.4. Decision Models ....................................... 32
       2.3.5. Model Repositories ..................................... 33
   2.4. Software Ecosystems .......................................... 34
       2.4.1. Closed versus Open Platforms ......................... 34
       2.4.2. Software Ecosystems in Research ...................... 36

3. State of Research 37
   3.1. Variability Modeling Languages ............................. 37
3.2. Variability Modeling in Practice .............................................. 39
  3.2.1. Experiments ................................................................. 40
  3.2.2. Industrial Experience Reports ........................................ 41
  3.2.3. Variability Model Evolution .......................................... 41
3.3. Tools and Evaluation .......................................................... 41
3.4. Variability in Open Source Projects ....................................... 42
3.5. Knowledge-Based Configuration ............................................ 43
3.6. Software Ecosystems ............................................................. 44
  3.6.1. Development Processes ................................................ 44
  3.6.2. Relationship between Variability Models and Manifests .......... 45
3.7. Conclusions ............................................................................ 45

4. Variability Modeling Languages .................................................. 47
  4.1. Methodology .......................................................................... 47
  4.2. Language Introduction ........................................................ 48
    4.2.1. eCos and the Component Definition Language ..................... 48
    4.2.2. Linux and the Kernel Configuration Language ................... 49
  4.3. Conceptual Framework .......................................................... 50
    4.3.1. Feature Kinds ............................................................... 50
    4.3.2. Feature Representation ................................................ 52
    4.3.3. Feature Hierarchy ......................................................... 52
    4.3.4. Feature Groups ............................................................ 53
    4.3.5. Feature Constraints ...................................................... 53
    4.3.6. Further Concepts ........................................................ 53
  4.4. The CDL Language ................................................................ 55
    4.4.1. Feature Kinds ............................................................... 55
    4.4.2. Feature Representation ................................................ 55
    4.4.3. Feature Hierarchy ......................................................... 56
    4.4.4. Feature Groups ............................................................ 56
    4.4.5. Feature Constraints ...................................................... 57
    4.4.6. Feature-to-Code Mapping .............................................. 57
    4.4.7. Further Concepts ........................................................ 58
    4.4.8. Formal Semantics ........................................................ 58
    4.4.9. CDL Propositional Semantics ........................................ 63
  4.5. The Kconfig Language ............................................................ 66
    4.5.1. Feature Kinds ............................................................... 66
    4.5.2. Feature Representation ................................................ 67
    4.5.3. Feature Hierarchy ......................................................... 68
    4.5.4. Feature Groups ............................................................ 69
    4.5.5. Feature Constraints ...................................................... 69
    4.5.6. Mapping to code ........................................................... 70
    4.5.7. Further Concepts ........................................................ 72
    4.5.8. Formal Semantics ........................................................ 72
## Contents

4.6. The Configurators ................................................. 74
   4.6.1. Process ..................................................... 74
   4.6.2. Reasoning and Limitations ................................ 75
4.7. Conclusions ...................................................... 76

5. Variability Models .................................................. 79
   5.1. Methodology .................................................... 79
   5.2. The Systems ................................................... 80
      5.2.1. eCos ...................................................... 81
      5.2.2. Kconfig Systems .......................................... 81
   5.3. Model Content ................................................. 84
      5.3.1. Feature Themes .......................................... 84
      5.3.2. Feature Classification .................................. 85
   5.4. Organization and Hierarchy .................................... 91
      5.4.1. Organizational Structures ............................... 91
      5.4.2. Model Hierarchies ....................................... 97
   5.5. Constraints .................................................... 99
      5.5.1. Group Constraints ....................................... 100
      5.5.2. Feature Constraints ..................................... 102
   5.6. Conclusions .................................................... 105

6. Software Ecosystems ................................................ 109
   6.1. Methodology .................................................. 109
      6.1.1. Subject Selection Criteria ............................... 110
      6.1.2. Data Sources and Analysis Infrastructure ............... 111
   6.2. Conceptual Framework ....................................... 111
      6.2.1. Software Ecosystem ..................................... 111
      6.2.2. Variability Representation .............................. 112
      6.2.3. Instance Derivation ..................................... 112
   6.3. Organization and Scale ..................................... 113
      6.3.1. Organization .............................................. 113
      6.3.2. Scale and Growth ........................................ 115
   6.4. Variability Mechanisms ..................................... 117
      6.4.1. Variability Representation .............................. 117
      6.4.2. Decisions ................................................. 118
      6.4.3. Encapsulation ............................................ 120
      6.4.4. Interactions .............................................. 120
   6.5. Dependencies ................................................ 122
      6.5.1. Specification, Semantics & Expressiveness .............. 122
      6.5.2. Dependency Structures .................................. 124
   6.6. Conclusions .................................................. 126
7. Discussion and Outlook 129
  7.1. Towards a Theory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
    7.1.1. Conceptual Framework: Putting it All Together . . . . . . . . . . 129
    7.1.2. Phenomena, Hypotheses, and Research Questions . . . . . . . . . . 132
  7.2. Guidelines for Practitioners . . . . . . . . . . . . . . . . . . . . . . . . . 134
  7.3. Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
    7.3.1. Software Product Lines . . . . . . . . . . . . . . . . . . . . . . . . 136
    7.3.2. Software Ecosystems . . . . . . . . . . . . . . . . . . . . . . . . . . 137
  7.4. Outlook: Industrial Variability Modeling . . . . . . . . . . . . . . . . . . . 138
    7.4.1. Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
    7.4.2. Preliminary Survey Results . . . . . . . . . . . . . . . . . . . . . . 139
    7.4.3. Preliminary Interview Results . . . . . . . . . . . . . . . . . . . . . 143
  8. Conclusions 145
    8.1. Summary of Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
      8.1.1. Research Question RQ1 . . . . . . . . . . . . . . . . . . . . . . . . 145
      8.1.2. Research Question RQ2 . . . . . . . . . . . . . . . . . . . . . . . . 146
      8.1.3. Research Question RQ3 . . . . . . . . . . . . . . . . . . . . . . . . 147
      8.1.4. Research Question RQ4 . . . . . . . . . . . . . . . . . . . . . . . . 147
    8.2. Research Impact . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
    8.3. Perspective . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

A. Analysis Tool Infrastructure 151
  A.1. CDLTools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
  A.2. KBuildMiner . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
  A.3. Linux Variability Analysis Tools . . . . . . . . . . . . . . . . . . . . . . 152
  A.4. Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152
  A.5. FOSD Cool Wall . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

B. Software Ecosystem Statistics 155
  B.1. Scales and Growth Rates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
    B.1.1. Current Sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
    B.1.2. Growth Rates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
  B.2. Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
  B.3. Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
    B.3.1. Raw Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
    B.3.2. Synthesized Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . 158
  B.4. Static Analysis of Android Bytecode . . . . . . . . . . . . . . . . . . . . . 159
    B.4.1. Intent Mechanism . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
    B.4.2. Dataflow Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
    B.4.3. Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
  B.5. Dependency Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
    B.5.1. Dependency Type Histograms . . . . . . . . . . . . . . . . . . . . . . 165
    B.5.2. Comparisons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168
B.6. Dependencies and Sizes of Units .................................. 169
  B.6.1. Debian ....................................................... 170
  B.6.2. Eclipse ....................................................... 172
  B.6.3. Android ....................................................... 173

C. Survey Questionnaire ................................................. 175

Bibliography ............................................................. 181

List of Figures .......................................................... 205

List of Tables .......................................................... 208
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CDL</td>
<td>Component Definition Language</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CSP</td>
<td>Constraint Satisfaction Problem</td>
</tr>
<tr>
<td>CVL</td>
<td>Common Variability Language</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>FMP</td>
<td>Feature Modeling Plugin</td>
</tr>
<tr>
<td>FOSD</td>
<td>Feature-Oriented Software Development</td>
</tr>
<tr>
<td>GP</td>
<td>Generative Programming</td>
</tr>
<tr>
<td>GSD</td>
<td>Generative Software Development</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>MSR</td>
<td>Mining Software Repositories</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean Satisfiability Problem</td>
</tr>
<tr>
<td>SPLE</td>
<td>Software Product Line Engineering</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“The scientist builds in order to study; the engineer studies in order to build.”
- Frederik P. Brooks [Bro96]

Considering software as a composition of features can arguably be seen as one of the most important shifts in thinking on our road to mass-customization [Pin93] of software. Features represent the common and variable characteristics of products in a software product line [CN01, PBVDL05, WL99, LSR07] and seem to be an appropriate abstraction to enable the automatic generation of tailor-made software products. But while the notion of a feature [KCH+90, CHE05b, vGBS01] might seem obvious to many of us, it is a huge controversy in the research field of software product lines. Features are neither components, nor classes, objects or individual files—they are abstract entities used in a multitude of contexts, such as software configuration, product marketing, or during scoping in requirements engineering.

Not too long ago, before finishing my dissertation, I gave a talk on our empirical work in the field of software product lines and variability modeling—the work that constitutes the main parts of this dissertation. In the audience was David Parnas, one of the early researchers on software product lines [Par76]. We had a conversation afterwards, where he told me that it was never clear to him what exactly a feature is, but that this concept became more graspable with the examples I provided in the talk. This interesting conversation illustrates a major problem in software product line research, where numerous abstract descriptions and definitions of the term feature exist [KCH+90, Bos00, Bat05, AK09]. Without mapping features to entities in real-world systems and studying realistic examples, it is difficult to understand the notion of a feature and its semantics—and, more importantly, to ultimately guide the development and management of configurable feature-based systems. In this light, this dissertation addresses a prime gap in product line research [HCMH10, CABA09, CB11]—empirical work that aims at building and verifying theories, and providing requirements for practitioners, such as language designers and tool builders.
1. Introduction

1.1. Trends and Motivation

The increasing complexity of software and the demand for higher quality and shorter time-to-markets has changed the way software is engineered. New and demanding application domains require complex applications developed by large numbers of developers, potentially distributed over the whole world. New variants of software should be quickly available; thus, reuse of software becomes more critical to achieve strategic advantages.

Developing configurable, tailor-made software with variability—software product lines—and creating vibrant ecosystems of software are among the most successful recent trends and paradigms to tackle these challenges. In contrast to standard software, which approach their requirements in a “one size fits all” [SC05] fashion, software product lines and software ecosystems aim at providing customized products for specific environments and use cases. However, these benefits come at the cost of increased complexity and might not pay off at all [MNJP02], or encompass high risks in organization [Coh02].

1.1.1. Software Product Lines

Lifting software engineering from single systems development to mass customization, as applied in the car industry, is at the very heart of software product line engineering (SPLE). Instead of separately developing individual software products, SPLE allows companies to efficiently create portfolios of systems in an application domain—software product lines—by leveraging commonalities and carefully managing the variabilities among the systems [CN01, WL99]. Product lines strive to solve one of the main problems of software engineering: efficient reuse of code and non-code assets.

Although product lines can be realized using different approaches, such as cloning of existing products, SPLE aims at scalable and systematic engineering. A variable platform provides the basis for the derivation of products in an automated process. Two major trends for variability have been observed [GBS01, Gur03, Bos05, JB09]: first, variability in hardware is gradually transformed to variability in software; second, binding times shift from early (build time) to late dynamic binding (execution time) in order to delay strategic decisions as long as possible. Furthermore, dynamic binding allows the re-configuration of products to flexibly change decisions, as opposed to static binding mechanisms.

Interactive configuration tools (configurators for short) are popular means to cope with complex variability in product lines. They drive the derivation process and help users making valid decisions about the target product. Configurators rely on variability models, such as feature or decision models [KCH+90, SRG11], which specify the configuration space—choices and their valid combinations. A large number of variability modeling languages and configurators have been designed both in industry, such as pure::variants [Gmb06] or Gears [Kru07], and in academia, such as the Dopler tool suite [DRGN07], FeatureIDE [KTS+09] or XFeature [RP05]. Further practical relevance can be seen in the recent adoption of the Feature Model incubation project\(^1\) under the

\(^1\)http://eclipse.org/proposals/feature-model
1.1. Trends and Motivation

Eclipse Modeling Framework Technology umbrella, and in OMG’s Common Variability Language—a standard proposal presently under review [CVL12, Obj09].

Despite the simple idea underlying software product lines, the efficient creation of scalable configurable systems is still a major challenge due to the inherent complexity of software variability. In contrast, mass-customization is common practice in other industrial domains, such as automotive, telecommunications, or financial services. Fig. 1.1 shows the web-based configurator of a BMW car, allowing customers to select valid combinations of desired equipment, and to draw conclusions about the price. Product configurators, focused on physical goods or services, have been researched in the field of Knowledge-Based Configuration [Stu97, GK99], a branch of Artificial Intelligence (AI). The BMW configurator is only one of many—a public catalog currently lists impressive 900 web-based configurators.

Figure 1.1.: Automotive example: car configurator

---

2http://www.omgwiki.org/variability
3http://www.bmw.com
4http://www.configurator-database.com
1. Introduction

1.1.2. Software Ecosystems

Complementing the advantages of product lines, we recently observe the trend in many software businesses to open up platforms to third-party contributions, striving to establish vibrant ecosystems of software. Consider the mobile-phone domain, where the recent shift from closed and centrally managed systems to open and extensible platforms, such as Android and iOS, led to some of the largest ecosystems of software in existence today. Fig. 1.2 shows the Google Play Store, Android’s main distribution channel for applications (apps). The Google Play client allows users to conveniently select apps and highly customize their mobile phones.

As natural successors of successful software product lines [Bos09, Bos10], software ecosystems are becoming increasingly popular due to their economic, strategic, and technical advantages [BA11, MS03, BWB12]. From a user’s perspective, ecosystems are a successful approach to mass customization: users select the desired functionality of their instance—a phone, an IDE, or an operating system—using proper tools. From a

---

5http://play.google.com

6This claim is further supported by publications showing successful applications of product lines spanning multiple organizations [MRM06, McG10], literature surveys [BA11], or comparative studies [Sch10].
technical perspective, software ecosystems are large systems consisting of interrelated components—apps, packages or features—built upon a common software platform; thus, supporting software reuse. From an economic perspective, ecosystems enable the sharing of a commodity burden—when many companies contribute to a shared project—or foster new business models and markets.

While software product lines are usually developed within one organization, software ecosystems enable inter-organizational reuse [Bos09] by outsourcing the realization of niche, or very domain-specific requirements to third-party developers [Rad12]. This strategy is related to the Open Innovation paradigm [Che03, McG09, WG06, Mil07]—an empirical observation about one of the most significant shifts in managing successful technology-driven innovation processes in the software industry: instead of developing products completely on their own, companies should leverage both internal and external ideas and resources. Fig. 1.3 shows an adapted version of the Open Innovation paradigm, but using our own terminology—built during our exploratory study—for software ecosystems. A platform supplier incorporates mechanisms to enable outside contributions, which either become part of the main platform, or of the less-controlled, free market around it. Consumers are provided with tools, such as the Google Play client, to use assets from the whole ecosystem.

### 1.1.3. Empirical Software Engineering

The knowledge that exists in practice certainly exceeds what we know as researchers in our field. However, that knowledge is not explicit, not systematically available, and we cannot use it directly to design new languages, tools, or processes in order to improve software engineering practices. Thus, we need to study the practice (or the “nature” as natural scientists would say) by eliciting and evaluating facts in order to conceptualize...
1. Introduction

and cross-link these concepts. Put simply, to build a theory.

Empirical work is gaining momentum in software engineering research [ESSD08], given the richer sets of data we can elicit from many different sources, such as companies, individual developers, users, and especially large source code archives now being available with the success of open source software. One of the recent hot topics in software engineering is Mining Software Repositories (MSR) [Has08], which applies data mining and analysis techniques on heterogeneous information available in source code repositories, issue and bug trackers, or mailing lists. Its ultimate goal is to learn from the history of software engineering projects to guide and improve future developments. MSR is a prime example of a software engineering subfield that rose successively over the last ten years, starting with a workshop co-located with the International Conference on Software Engineering and now having established the Working Conference on Mining Software Repositories with accompanying summer schools for its growing community comprising researchers, students, and large companies that use MSR knowledge and tools to improve software quality and development efficiency.

1.2. Problem Statement

The field of SPLE rose similarly to MSR, but is slightly older with significant research starting around twenty years ago. It started with industry-funded research projects and small workshops that provided the basis for establishing its main venue—the Software Product Line Conference—facing its 16th edition this year. The focus of the SPLE community was more on engineering research to develop solutions for coping with the complexity that comes from variability in software artifacts, and to scale these solutions to industry-level product lines. There is so far little empirical work in SPLE that analyzes industrial practices and verifies whether the well-researched and designed techniques for SPLE processes, variability management, or variability modeling are used in practice, and how they are applied. For example, there is a wealth of different variability model analysis and reasoning techniques that is based on one of the most dominant formalizations of variability in the field: feature models [KCH+90, Kan09, CE00]. However, it is largely speculation whether all these techniques can be applied in real industrial and open source projects. In fact, although as many as 91 approaches [CB11] to variability management have been introduced, only very little work explores their application in practice. The majority of techniques is evaluated on small toy examples created by researchers.

1.2.1. Empirical Data on Variability Modeling

The lack of analyzable artifacts, such as industrial modeling languages and models, is symptomatic for the field. While reports indicate the existence of large variability models with thousands of features [SPK06, STB+04, LP07], these models are highly protected and not available to researchers, since they contain core strategic knowledge about current and future products of a company. Although public model repositories, such as S.P.L.O.T. [MBC09], have been established, they contain rather small and mostly
1.2. Problem Statement

academic examples. Furthermore, many academic languages—such as the one used in S.P.L.O.T.—have limited modeling capabilities, which generally questions their practical applicability.

Literature mainly speculates about the occurrence and frequency of certain variability modeling concepts. Although a large number of different techniques for configuration, analysis, benchmarking or consistency checking of variability models have been conceived both in academia and industry, evaluations of these techniques often rely on generated models. Their generators rely on assumptions about realistic models (such as in [TBK09, MWC09]). To the best of our knowledge, no work provides proper assumptions based on empirical evidence. In particular, none of the major books on SPL [CN01, WL99, PBVDL05], even the more practice-oriented [KSP09, LSR07], contains any quantitative data on variability models, although these constitute the central artifact of a product line.

The lack of empirical data on variability modeling is also recognized in literature studies [HCMH10, CABA09], which identified only few papers reporting practical experience. Even those that report on experience provide only little details on the models used. Thus, research effort about real variability modeling languages and their instances is necessary to provide requirements for tools, such as configurators and reasoners; to gain realistic model assumptions for evaluations of variability management techniques; to facilitate model transformations; and to guide future research.

1.2.2. Applicability of Variability Models

Studying the applicability and limitations of variability models is another research challenge in SPL. It is, so far, rather speculation in which organizational contexts variability models are applicable. Do they support distributed development or variability management? Or can we even use variability models to specify the variability in open platforms? What is their influence on dependency structures?

Closed versus open platforms. As natural extensions of successful software product lines, ecosystems also aim at mass-customization—by maintaining huge variability among and within their software assets. However, while product lines target intra-organizational reuse, ecosystems facilitate inter-organizational reuse by opening up the platform to third-party contributions—and add a new dimension of complexity that has to be tamed. The distinction between closed and open platforms is important to the remainder of our work. We consider a platform open when there is explicit technical support for consumers to use third-party assets in an instance, and closed when outside contributions need to be integrated into the platform with a controlled process.

While variability mechanisms in software product lines and closed platforms are reasonably well understood, that is not the case for software ecosystems with their open platforms. Research has addressed ecosystems, but focused on economic, strategic, and organizational aspects [BWB12, vGPB10]—largely sidestepping technology. While ecosystems are clearly driven by business and strategic forces, it largely remains speculation what and how mechanisms sustain their success and growth. What are their
characteristics and how do ecosystems with closed and open platforms differ? How is a mechanism related to an ecosystem’s organization or to dependency structures?

Thus, to study the applicability and limitations of variability models and their concepts, we need to explore variability mechanisms in software ecosystems. While closed platforms with complex variability, such as the Linux kernel, rely on variability models, open platforms, such as the Debian package manager, or the Android platform, rely on distributed manifest files to express variability information. A clear difference certainly is that variability models are designed under a closed-world assumption, where variability in ecosystems is characterized by an open-world assumption. Understanding the applicability of variability models, and their relationship to manifests [CZ10, GBS10, Sch10], is a main research issue in SPLE7.

Domain impact. “Many people have the feeling that the concept of software product lines fits better for embedded systems” [BCL12]. This statement emphasizes that fine-grained variability with expressive variability mechanisms might only be necessary for complex (C/C++-oriented) embedded systems with their specific requirements, such as static configuration; while these mechanisms might be too complicated for end-user-oriented software. This challenges proponents of configurators that i) should be directly usable by less technically skilled end users, and ii) should be used within product lines written in modern programming languages, such as Java, C# or Scala. We believe that this highly controversial speculation about the domain impact for variability modeling requires to study a broad range of configurable systems—a challenge we partly address.

1.3. Research Hypothesis, Questions, and Methodology

Our empirical dissertation is based on the following research hypothesis:

By building a theory that is grounded in empirical data about real-world variability modeling concepts and their use in real large-scale systems, we will be able to: i) explain how a certain concept sustains the success of a platform, ii) make informed decisions when developing closed and open platforms, iii) build better modeling, reasoning, and analysis tools, and iv) guide future research.

This research hypothesis is reflected in our mix of exploratory and descriptive research questions. We define four high-level questions that are further refined into sub-questions. We also briefly sketch the methodologies of the study behind each research question; these will be described in more detail later. The first two questions represent the main focus of our work: variability models in closed configurable platforms, such as software product lines. Fig. 1.4 illustrates the main focus, and the direction into which we extend our work to set variability modeling into a broader context.

---

7 The applicability of variability models for distributed development and variability management is a contested issue in SPLE research; albeit it was shown that a variability model can be converted to a set of manifest files and back [CZ10, GBS10, Sch10].
1.3. Research Hypothesis, Questions, and Methodology

Figure 1.4.: Our empirical journey from software product lines to software ecosystems. The figure reflects dominant variability representations in each paradigm and type of domain: variability models in product lines and system-related domains; manifest files in ecosystems and rather user-oriented domains. Our main focus is on variability modeling in software product lines.

**RQ1** What variability modeling concepts are used in real-world languages?

We study syntax and reverse-engineer formal semantics for two variability modeling languages. Analyzing the semantics allows us to map identified language concepts to each other and to feature modeling concepts, further to recognize subtle semantic differences not obvious from syntax. Studying the full language design space provides basis for identifying concepts and their characteristics beyond feature modeling. This research classifies into four sub-questions:

**RQ1.1** Are the well-researched concepts of feature models used in real-world languages, and if so, how?

**RQ1.2** What is the full design space of real-world languages?

**RQ1.3** What are the semantics of concepts beyond feature models?

**RQ1.4** What tool support is available for our languages and what kind of reasoning about models is supported?

Investigating RQ1 provides qualitative empirical evidence for the relevance of the well-researched concepts of feature modeling, and identifies further concepts used by system designers, but neglected by researchers.
1. Introduction

To prioritize these concepts—to verify whether and how frequently they occur in real instances, we need to quantitatively study variability models. To characterize the content, and to determine design practices (such as feature grouping patterns) that were used to structure the models, we also need to qualitatively analyze the models. Both RQ1 and RQ2-related results aim at providing requirements for language designers, tool builders, and to provide guidelines for modelers and developers. This objective is reflected in our second major research question:

**RQ2** What are characteristics of real variability models?

We perform case studies on models of freely available open source projects from the systems software domain. We develop an analysis infrastructure to analyze models, define model metrics, and interpret and provide statistics derived from extracted and elicited models. RQ2 further classifies into:

- **RQ2.1** What is the content of the models?
- **RQ2.2** How to characterize the structure of the models?
- **RQ2.3** How to characterize constraints and dependency structures in the models?
- **RQ2.4** Do assumptions in literature about real models hold for our subjects?

To broaden our perspective on variability modeling, we extend our study on two dimensions: i) we investigate variability in open platforms, and ii) we investigate organizational structures—development and variability management—of our subjects. To this end, we study open platforms and their induced ecosystems in order to compare identified concepts, organizational structures, and dependencies with the variability-model-dominated mechanisms in closed platforms.

We aim to assess the organizational structures in which variability models are applicable, in particular, whether they are applicable in open platforms. We also study their relationship to distributed variability specification techniques—manifest files. We formulate the following exploratory research question:

**RQ3** What are key variability modeling concepts and variability mechanisms in open platforms?

We qualitatively and quantitatively study three of the most successful and largest open platforms and ecosystems in existence today, using large extracted datasets for the quantitative part. We also extend the previous analysis of our closed platforms to organizational structures. These results provide the basis for comparing variability modeling concepts in closed and open platforms on three levels: organizational structure (organization of development and variability management), variability mechanisms, and dependencies. Thus, we refine RQ3 into:

- **RQ3.1** What are organizational structures in our subjects?
- **RQ3.2** What are variability modeling concepts and variability mechanisms in open platforms?
- **RQ3.3** What are the concepts to declare dependencies in open platforms?
1.4. Contributions

RQ3.4 What are the corresponding dependency structures?

Broadening our perspective to open platforms and organizational structures aims at setting our variability modeling languages and models into a wider context—to study their limitations, and their applicability. It also helps us to identify concepts in the languages we have not found before. In fact, when identifying the concept of capability-based dependencies (Section 6.5.1.1), we were able to retroactively identify this concept in our variability modeling languages from the closed platforms; with less frequent occurrence in the variability models, however.

Finally, towards building a theory, we identify correlations, and strive to find causalities among the observations from our studies. Although the selection of our case studies aims at high representativeness of our results, it is inappropriate to generalize from a limited set of subjects. Thus, the goal of exploratory empirical work is to develop testable hypotheses, which have to be confirmed or refuted by follow-up studies of further subjects, or using other research tools, such as interviews or survey questionnaires. Our last research question targets theory building:

RQ4 How are the discovered modeling concepts related, and what causalities exist?

We investigate strong correlations, and develop hypotheses about qualitatively identified causalities. For correlations that need further research to be investigated, we formulate research questions.

On a final note, we follow the definitions of empirical research methods as given and described by Easterbrook et al. in [ESSD08], who elaborate on: theory building versus theory verification, qualitative versus quantitative analyses, and on conducting case studies. Case studies have been successfully used before in a similar context, to study open source software development (such as [MFH00]). We also emphasize that generating hypotheses by analysis of case studies is a highly qualitative and interpretive process by its nature. Finally, an interesting discussion on obvious and non-obvious results is provided by Tichy [Tic00].

1.4. Contributions

Our PhD work results in the following empirical contributions that advance our understanding of variability modeling techniques and mechanisms in real-world software product lines and ecosystems. Our engineering contributions comprise static analysis techniques for model and build system analysis.

C1 A conceptual framework explaining key variability modeling concepts.

Our conceptual framework is based on detailed empirical data extracted and synthesized from a wide range of software systems. The framework also relates modeling concepts to organizational structures of development and variability management.

We further contribute to the understanding of semantics of variability modeling
1. Introduction

languages. For example, we show that denotational semantics are a concise form of modeling variability model semantics. This style has not been used in variability modeling before.

Finally, we contribute a standard set of metrics to characterize variability models.

C2 An instantiation of the framework with empirical data.

Our qualitative analyses of the case studies results in a characterization of the concrete variability concepts (and their semantics) supported in each subject. We also interpret and provide statistics derived from extracted datasets: 13 variability models from twelve projects of the systems software domain, and large parts of three large-scale ecosystems.

These results further contribute to the initial discussion about the “nature” of a feature, by providing one more perspective in the domain of systems software, and relating the feature concept to entities in software ecosystems.

C3 Phenomena, hypotheses, and new research questions emerging from empirical data.

We formulate hypotheses as proposed explanations for our observed phenomena. These have to be confirmed or refuted by follow-up work, and might eventually lead to a more grounded theory behind variability modeling.

C4 Static analysis tools and large extracted datasets.

We make all our extracted models, research artifacts, datasets, and tools publicly available for transparency and follow-up studies. They also provide realistic benchmarks for analysis and reasoning tools.

1.5. Guide for Readers

Our dissertation is divided into three parts, followed by an Appendix with auxiliary information and datasets.

Essentials and state of research. In Chapter 2, we provide background information on software product lines, variability modeling, and software ecosystems to introduce underpinnings and common terminology for the remainder. In particular, for readers not familiar with these fields, the chapter gives an introduction into the basics of our research.

Chapter 3 reviews existing research that is related to our work, such as empirical studies, experience reports, and literature reviews on variability modeling.

Studies and results. Chapters 4 and 5 represent the main focus of our work: variability modeling in the context of software product lines. Chapter 4 presents our study on two real-world variability modeling languages, more precisely, on their concepts, semantics, and tool support. Chapter 5 comprises our study on the use of these languages in large-scale models from the systems software domain. It characterizes the model contents, their organization, and their constraints with respect to the set of metrics we introduced.
Chapter 6 then extends our discourse to variability in software ecosystems. We study organizational structures, variability mechanisms, and dependencies by performing case studies of two closed and three open platforms, which all successfully established an ecosystem of third-party contributions.

**Results analysis, discussion, and outlook.** In Chapter 7, we further analyze and discuss our results. We combine all parts of our iteratively developed conceptual framework, which was so far used to classify and compare all of our product line and ecosystem case study subjects, into a single whole. We provide a graphical overview in Fig. 7.1 and a tabular one in Table 7.1, the latter with references to details on concepts described in the corresponding parts of Chapters 4–6. We also compile a set of derived guidelines for language and tool designers. Most notably, Section 7.4 presents our study on industrial variability modeling and preliminary results, complementing the results from our open source subjects.

Finally, Chapter 8 summarizes our results with respect to each research question, discusses the impact of our research, and presents our perspective for future work.

**Appendix.** Appendix A describes our tool infrastructure and provides our extracted variability models. Appendix B expands on our extracted ecosystem datasets, supporting statistics, and the tools we developed, in particular, the static analysis of Android bytecode. Finally, Appendix C contains our survey questionnaire as part of our industrial variability modeling study.

### 1.6. Bibliographical Notes

Major parts of this dissertation are published by the thesis author, or currently under review. This dissertation extends and reuses their content. The following list contains the most-relevant publications, with references to the main chapters of their appearance. Smaller chunks of these publications are also used in Introduction (Chapter 1), Background (Chapter 2), Discussion and Outlook (Chapter 7), and Conclusions (Chapter 8).


2. Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability modeling in the systems software domain. Under review for IEEE Transactions on Software Engineering (TSE). (Chapters 3, 5)
1. Introduction

3. Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. Variability mechanisms in software ecosystems: Open versus closed platforms. Under review. (Chapters 3, 6)


5. Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature-to-Code mapping in two large product lines. Technical report. Institute of Computer Science, University of Leipzig, 2010. (Section 4.5.6, Appendix A.2)


The results of our research gave rise to further work, in which the thesis author was involved, such as:


Chapter 2

Background

This chapter provides background information on software product lines, variability modeling, and software ecosystems, to set the stage and introduce common terminology for our study. We provide an insight into architectures of open source systems software to put the discussed variability modeling concepts into an architectural context and elaborate on variability modeling semantics.

2.1. Software Product Lines

The first idea of product lines, albeit termed program families, dates back to 1968 at the NATO conference on software engineering, where McIlroy et al. [MBNR68] introduced the idea of software mass customization by reusing variability-enabled components built by dedicated suppliers as a solution to the software crisis back then. Later in 1976, Parnas [Par76] discusses methods of developing program families, using the assumption that it pays off to perceive a program family upfront, instead of developing the products sequentially. This assumption is reflected in his definition—he considers “a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members”.

This idea is central to software product lines and led to the explicit separation of domain engineering (development for reuse) and application engineering (development with reuse) in major SPLM methodologies [WL99]. Fig. 2.1 shows the two processes and their interactions. Domain engineering comprises domain analysis (scoping and eliciting domain knowledge, defining reusable requirements), domain design (modeling variability and designing a product line architecture), and domain implementation (developing the platform, reusable assets, and a reuse infrastructure, such as generators). Application engineering aims at deriving concrete products from the product line and comprises requirements analysis (eliciting product requirements), product configuration (configuring the platform based on product requirements, that is, configuring an existing variability model), and integration and test¹ (deriving the product, either by manual construction or automatic generation). Although product line development starts with domain

¹We will refer to this activity as product derivation, and assume an automated process using generators or build systems in the remainder of our work.
engineering, the two processes interfere and run in parallel to some extent; for example, to integrate new product-specific requirements into the product line or propagate bug fixes to products.

While there used to be a clear distinction between the terms software product line and product family, both are now used nearly synonymously [PBVDL05]. Earlier, product lines referred to an economic construct of products that were somehow related, but not necessarily using a common technological foundation; while product families always referred to a range of products based on a common platform. Both terminologies grew independently due to separated efforts made in both Europe and North America. For details and mappings between the terminologies, we refer to [CE00, CN01]. In the remainder, we use the nowadays more established term software product line in conjunction with assets and platform.

For an overview of concrete SPLE methodologies, we refer to our preceding Master’s thesis (Diplomarbeit) [Ber07], which describes the SEI framework [CN01], FAST [Wei95], PuLSE [BFK+99], KobrA [ABM00], FeatuRSEB [GFA98], and the framework of Pohl et al. [PBVDL05]. Czarnecki et al. [CE00] provide a further survey and a genealogy of domain engineering methods—in particular comprising Feature-Oriented Domain Analysis (FODA) [KCH+90], which is central to our work due to its invention of feature models as a variability modeling technique.

Despite the simple idea, proclaimed more than 40 years ago, software product lines still turn out to be a highly complex endeavor for software businesses; in particular due to their manifold implications to different concerns, mainly: business, architecture, process, and organization. These became known under the acronym BAPO [vdL02, OMA+00]; and their mutual interferences are illustrated in Fig. 2.2. In our work, we mainly target the architecture and organization concerns with respect to variability modeling; very
briefly touch the process (how variability models are built); but sidestep the business
concern (such as when introducing variability modeling pays off, like discussed for SPLE
itself in [MNJP02]).

![Diagram of BAPO: concerns affecting SPLE (according to [vdL02])]

2.1. Examples of Software Product Lines
Books on SPLE, primarily [CN01, LSR07], list many success stories of software product
lines in industry, including cases where the application of systematic reuse was a prime
necessity to survive in a market segment. The product line community maintains a
Hall of Fame\(^2\) of successful and documented industrial product line projects, comprising
companies such as Boeing (avionics software [Sha00]), Bosch (engine control software [STB\(^+\)04, TP00]), Nokia (networking [LSR07] and mobile phone firmwares [Jaa02]),
or Hewlett Packard (printer firmware [Ref09, PO97]).

However, to become a member of the Hall of Fame, the product line has to fulfill strict
requirements. Most importantly, it has to be conceived as a product line with planned
reuse from the beginning, and developed according to a formal product line process, like
the one in Fig. 2.1. However, preliminary results of our current industrial variability
modeling study (see Section 7.4) indicate that most software product lines are refactored
from an initial product or a set of independently developed products.

**Note on terminology.** Many systems, such as our open source subjects in Chapter 5,
have characteristics of product lines, but were likely not developed according to an
SPLE process. Since these are highly configurable systems, support automated product
derivation, and contain variability models, we refer to them as software product lines. In
fact, Sincero et al. [SSPS07] discussed this issue for the Linux kernel and confirm this
view.

2.1.2. Software Product Line Architectures
While SPLE aims at systematic and planned reuse, product lines can also be realized
using ad-hoc techniques, such as code cloning. In fact, our so far unpublished empirical
study on code cloning practices in industry [DRB\(^+\)12] confirms that cloning—if performed
under strict governance—is a common development technique, even applied for product

\(^2\)http://splc.net/fame.html
lines with up to 20 products. However, for larger variant spaces, configurator-based approaches using variability modeling are inevitable.

Figure 2.3.: Problem and solution space (adapted from [COV02])

The notion of problem and solution space [CE00] provides a useful abstraction to describe the architecture of configurator-based approaches. Fig. 2.3 shows an illustration, where the parts focused in this thesis are emphasized in gray. Variability models belong to the problem space, representing requirements and capabilities of the system. A mapping connects the problem to the solution space, which provides realizations (code, resources) for the abstractions in the former. Variability in the solution space is realized using dedicated variability mechanisms, which classify into compositional and annotative approaches [KAK08]. Examples of the former are plugin mechanisms, generative programming [CE00] or component-oriented techniques, such as OSGi [OSG09]; and of the latter the C/C++ preprocessor [PO97], which is dominantly used in embedded software product lines.

We briefly illustrate the principal architecture of configuration-based product lines by describing the problem space, mapping, and solution space of three configurable open source systems—the eCos operating system, and the kernels of Linux and FreeBSD, which we investigated during our variability studies. We refer to their configuration options as features, given the mapping we will later establish (Table 4.1 in Section 4.3). Fig. 2.4 summarizes common (solid line) and variable (dashed line) parts among the high-level architectures of the three systems. The problem space in Linux and eCos comprises variability models expressed in the CDL and Kconfig language, with graphical configurators on top (explained and analyzed in Chapter 4). The variability models describe features, their possible values, and dependencies among them. In contrast, FreeBSD has no variability model, but a list of features with textually documented dependencies (such as \texttt{IPI\_PREEMPTION} \rightarrow \texttt{PREEMPTION}), as shown in Fig. 2.5 (left). Interestingly, all three systems distinguish between features configuring the target hardware architecture, and non-hardware (mostly functional) features.

The mapping between features and solution space (code) is realized within the build system. A mapping declares which files have to be included (compiled) for the final product under a certain configuration. It is explicitly specified within the variability model in eCos (see Section 4.4.6), hidden in imperative build logic in Linux (see Section 4.5.6), and explicitly declared as file presence conditions—logical expressions over features—in a
2.1. Software Product Lines

mapping file in FreeBSD, as shown in Fig. 2.5 (right). In contrast to the imperative build scripts in Linux, both eCos and FreeBSD rely on a Makefile generator, which evaluates the mapping for a specific configuration and creates build specifications for the C/C++ files to compile.

All three systems generate header files that define activated features as preprocessor symbols, to be used with C/C++ #IF and #IFDEF preprocessor directives. These enable fine-grained parametrization of source files. These directives can even cut C/C++ language statements or conditional expressions. In fact, undisciplined annotations are commonly used [KGR+11]. For an analysis of preprocessor use in open source product lines, see for example [LAL+10].

Further Reading. The architectures we just described rely on variability mechanisms known as annotative approaches [KAK08]—a preprocessor cuts out irrelevant parts of the platform during product derivation. From experience, this style represents the architecture of many open source configurable systems, but is also—reportedly—used in commercial product lines3, such as Hewlett Packard’s printer firmware “Owen” [Ref09]. Compositional approaches, such as our analyzed software ecosystems (see Section 2.4 and Chapter 6), have faced more attention in the research community, due to their cleaner design and separation of concerns, as opposed to the maintenance-intensive C/C++ preprocessor [SC92]—sometimes even called the “IFDEF hell”. However, the work of

3“I am not surprised that parameterization by preprocessor directives has been a sturdy meme which has spread in legacy code despite of its numerous disadvantages. This could only have happened because parameterization is very useful—it is the key to reuse, one of the great goals of software engineering.” [Sim99]
2. Background

# IPI_PREEMPTION instructs the kernel to preempt threads running on other CPUs if needed. Relies on the PREEMPTION option

## Mandatory:

- Device
  - apic

- I/O
  - apic

## Optional:

- Device
  - options
    - MPTABLE_FORCE_HTT
    - Enable HTT CPUs

- Device
  - options
    - IPI_PREEMPTION
    - Watchdog routines.
    - MP_WATCHDOG

- Device
  - options
    - COUNT_XINVLTLB_HITS
    - Counters for TLB events
    - COUNT_IPIS
    - Per-CPU IPI interrupt counters

# CPU OPTIONS

- You must specify at least one CPU (the one you intend to run on); deleting the specification for CPUs you don’t need to use may make parts of the system run faster.

## Device

- cpu
  - 4466_CPU
  - 1866_CPU

**Figure 2.5.** Feature specification without variability model (left) and feature-to-code mapping (right) in FreeBSD

Kästner [Käs10] shows that the advantages of both approaches can be combined by providing proper tool support for preprocessor-based variability.

Further implementation techniques are described by Czarnecki et al. [CE00] (generic programming, C++ template metaprogramming, aspect-oriented programming, and code generators), Pohl et al. [PBVDL05] (component frameworks), Völter et al. [VV11] (DSL-based transformation). Svahnberg et al. [SvGB05] provide a general taxonomy on variability mechanisms, while both van Gurp et al. [vGBS01] and Völter [Völ09] identify common patterns of variability mechanisms. As a realization of the intentional programming paradigm [Sim95], the IDE JetBrains MPS\(^4\) allows switching on and off features mapped to nodes in the abstract syntax tree (AST) of the underlying programming language [Völ10, Völ11].

Finally, for further mapping techniques, we refer to Heidenreich et al. [HW07], who present feature-to-model mappings with their tool FeatureMapper\(^5\), and Czarnecki et al., who describe the concept of a presence condition [CA05].

### 2.2. Variability Modeling

Variability models are central artifacts in product lines with complex variability, which needs to be explicitly declared to remain manageable [DSB05]. Variability models describe the product line’s variability, meta information, and dependencies; and are input to configurators. An overview on software variability modeling approaches is provided by Sinnema et al. [SD07], who classify five academic—CBFM [CHE05a], COVAMOF [SDNB04], VSL [Bec03], ConIPF [HWK+06], Koalish [ASM04]; and one commercial variability

\(^4\)Meta Programming System, \url{http://www.jetbrains.com/mps}

\(^5\)http://www.featuremapper.org
modeling language—pure::variants [Gmb06]. Each of which represents a different style of modeling: CBFM, ConIPF, and pure::variants represent variability in terms of features; COVAMOD and VSL in terms of variation points; and Koalish is embedded into the architecture description language (ADL) Koala [vOvdLKM00].

A range of automated tools supports the activities around variability modeling: analyzers verify model consistency or detect dead features (see Section 2.3.2.1), graphical configuration tools (configurators) support intelligent choice propagation and model completion [DRGN07, WSB+08, JBGS10]. These tools are usually optimized for models with certain properties, such as a specific size and specific density of constraints, due to the computational hardness of configuration problems.

Most significant research on variability modeling was accomplished within the last 22 years, with one of the most influential contributions being feature models [KCH+90]. In fact, of 91 variability management approaches introduced [CB11], 33 use feature models to specify variability information. Many graphical and textual variability modeling languages with corresponding configurators have been developed. In the following, we provide a list of tools that have been considered in academic publications. Academic feature-based languages (mostly open source) comprise, for example:

- **TVL**, the Textual Variability Language [CBH11, HBH+11]
- **ClaFeR**, textual Class-Feature-Relationship language [BCW11], a combination of class and feature modeling
- **Dopler** tool suite [DRGN07, DHR10], a graphical language and configurator based on decision models, which are similar to feature models, see Section 3.1
- **FeatureIDE** [KTS+09], an integrated development environment supporting graphical feature modeling and preprocessor-based variability in Java (graphical)
- **AHEAD Tool Suite** [Bat04], an algebraic, textual feature specification language, and a feature-based program synthesis tool suite based on mixins [FKF98] and mixin-layers [SB02]; that is, feature modules realized using the Java derivative JAK [BSR04]
- **KumbangModeler** [MRM07, KRNM07], a graphical configurator based on the Kumbang domain ontology [AMS07]
- Smaller graphical tools, such as the feature modeling plugin (FMP) [AC04], Captain-Feature [BEL04] or AmiEddi [Mar04]

Commercial languages comprise, for example:

- Gears from BigLever Software, Inc. [Kru07, Kru02]
- Pure::variants from pure::systems GmbH [Gmb06, Beu04, Beu03]
- Product Modeler from Configit, Inc. [Con05]
- XFeature from P&P Software [RP05]

We are so far not aware of any exhaustive survey on variability modeling tools for software. However, the overview of Sinnema et al. [SD07], a tool survey of Munir et al. [MS10], and our survey on industrial variability modeling (Section 7.4) should give a relatively complete picture. Recall that we consider configurators for software, not for tangible products (or services), as focused in the knowledge-based configuration domain—where a good overview of tools is provided by Gronau et al. [GS05].
2. Background

2.2.1. Variability Model Semantics

Variability modeling languages can have complex semantics. In fact, we usually do not know the full semantics\(^6\) of a language with regard to specific use cases, such as the interpretation in alternative configurators and build systems, the behavior of default values, or reasoning including choice propagation and conflict resolution. Thus, certain analyses rely on specific abstractions of the full semantics. The particular use case determines the portion of the full semantics needed and what abstraction is required.

The primary meaning of a variability model is the set of valid configurations that adhere to all the constraints in the model, known as the **configuration space semantics**. We will provide this semantic abstraction for feature models (Section 2.3.2) and our two subject languages (Chapter 4) in the study. A common approach to provide semantics is to map models to formulas in propositional or first order logic—so-called translational semantics.

Other useful abstractions are the behavior of the configurator with regard to the configuration process, referred to as **configurator semantics**, or the **ontological semantics** describing the concept hierarchy of features. In fact, the tree structure of a feature model is fully neglected in the configuration space semantics.

2.2.2. Configuration Process

To derive a product from the product line, configurators aim at interactive, stepwise creation of a valid configuration within the configuration space of the variability model. Various processes exist, primarily:

- **Complete configuration** of the model in one self-contained process, making decisions for all features. The state of each feature is changed from undecided to decided. This process often uses **valid domain computation** [HSJ+04, HA07], which calculates the remaining possible decisions left after each step, and automatically propagates decisions to required or excluding features. See Section 4.6.1 for more details.
- **In re-configuration**, the complete model is always decided, based on a default configuration—either with explicitly declared defaults per feature, or “default” defaults per feature type. Re-configuration is used in our studied languages, as we will see in Section 4.6.1.
- **Staged configuration** [CHE05b, CHE05a] refers to stepwise refinement of a model, where decisions lead to syntactic transformations of the model that narrow the configuration space. For example, after selecting an optional feature, it becomes mandatory in the derived model after the step. This is useful in a supply chain, where subsequent users should be provided with partly preconfigured models, but without being able to change previous decisions.

\(^6\)Since the reality of language use is very complex given the different expectations of users and tools, it might be philosophical to say that there is one single (inconsistent or paraconsistent) semantics, instead of many consistent semantics. Thus, taking the perspective of a single consistent semantics is a platonic stance, but a good approach to develop abstractions on top, for example, to utilize classical logic.
2.3. Feature Models

Feature models were introduced as part of the FODA domain engineering methodology [KCH+90] and gained popularity due to their simple and intuitive notation. The notation was later refined in the generative programming book [CE00], which now has become the predominantly used notation. Feature models are tree-like menus of configuration options—features—with constraints among them. The constraints either reside in the graphical notation (hierarchy, optional/mandatory features, and groups), or are declared as additional cross-tree constraints.

Fig. 2.6 presents a feature model in the generative programming notation. It shows the variability of the Journaling Flash File System—one of the numerous file systems supported in Linux and eCos, our major study objects (Section 4.2). We created it in order to use it as a running example later. The boxes represent features. The hierarchy represents dependencies; for instance, the Default Compression feature allows a further choice of sub-features that refine it: None, Priority or Size. Filled dots mark mandatory features (like Debug Level), which must be selected if the parent is. Hollow dots represent optional features, which do not have this constraint. Further, several features can be related by a group constraint: the sub-features of Default Compression are connected by an arc denoting the xor group constraint—exactly one of the three choices has to be selected. Finally, textual cross-tree constraints are listed to the right.

Note that we used features that can take integer and string values. Such types were envisioned for additional attributes of features in the original FODA report [KCH+90], later referred to as features with attribute [BSRC10], or attributed feature models. We allow non-Boolean values for features directly, and will simply call them data features throughout our work.

2.3.1. The Notion of a Feature

In the formal semantics we provide shortly, features are labels that can be assigned values in a configuration. Setting them into context, for instance relating them to the architecture, requirements or to code, gives features a meaning. Since many definitions for a feature are provided in literature, we repeat the most important ones and then point out our view. Among others, a feature is:

- Kang et al. [KCH+90]: “A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.” (originating from [Mor85])
- Batory [Bat05] and Zave [Zav04]: “An increment in program functionality.”
2. Background

Figure 2.6.: Feature model of the JFFS2 filesystem (excerpt)

- Apel et al. [AK09]: “A unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option.”
- Czarnecki et al. [CE00]: “A distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept.”
- Bosch et al. [Bos00]: “A logical unit of behaviour specified by a set of functional and non-functional requirements.”
- Classen et al. [CHS08]: “A triplet, $f = (R, W, S)$, where $R$ represents the requirements the feature satisfies, $W$ the assumptions the feature takes about its environment and $S$ its specification.”

This list is inspired by [CHS08] and [AK09], which both reflect on even more definitions. We refrain from developing yet another one based on our empirical studies; however, we found that the definition by Apel et al. [AK09] from the FOSD (Feature-Oriented Software Development) paradigm fits best for the features we found in our subject variability languages and models. Our features represent user- and non-user-selectable (derived) increments in functionality, but also parametrization options for this functionality. For example, the features in our operating system kernels are, among others, drivers, protocols, file systems or multimedia devices, and parameters to configure them. These features are mapped to software artifacts in the solution space, as explained previously.

2.3.2. Feature Model Semantics

Many works have studied and defined formal semantics of feature models to avoid ambiguities in language use and to guide tool builders, aiming at correct interpretation of language concepts. Given the expressiveness of FODA feature models, translational semantics to propositional logic are the most popular approach. Such semantics have been defined, among others, by Batory [Bat05], Bontemps et al. [BHST05] or Wei Zhang.

7To be precise: the entities of the languages we mapped to features of feature modeling languages.
2.3. Feature Models

Table 2.1.: Intuitive translational semantics for propositional feature models (from [Käs10], inspired by [Bat05, Men09])

<table>
<thead>
<tr>
<th>Feature Model Edge</th>
<th>Propositional Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>( f \leftrightarrow p )</td>
</tr>
<tr>
<td>( f \rightarrow p )</td>
<td></td>
</tr>
<tr>
<td>( f \rightarrow p )</td>
<td></td>
</tr>
<tr>
<td>( (f_1 \lor \ldots \lor f_n \leftrightarrow p) \land \bigwedge_{i&lt;j} \neg(f_i \land f_j) )</td>
<td></td>
</tr>
<tr>
<td>( f_1 \lor \ldots \lor f_n \leftrightarrow p )</td>
<td></td>
</tr>
</tbody>
</table>

et al. [ZZM04]. Furthermore, a survey on the semantics of seven variants (expanded on shortly) of FODA feature models is provided by Schobbens et al. [SHTB07, BHST04]. Since the translation to a formula looses the feature hierarchy, the reverse-transformation is a major challenge, as pointed out in [CW07], and later addressed in our\(^8\) semi-automatic approach on reverse-engineering feature models from formulas [SLB+11]; or related approaches, such as [ACC+11]. Beyond propositional logics, further semantics based on higher-order [JK07], probabilistic [CSW08], and description logics [WLS+05], or constraint logic programming [Apt03] (such as Forfamel [AMS06], which uses the WCRL [NSS99, SNS02] language) have been presented in the literature.

To provide a basic understanding of feature model semantics for the remainder, we start with an intuitive form of translational feature model semantics, and then define semantics more formally in a denotational style.

2.3.2.1. Propositional Translational Semantics

The most intuitive form of defining semantics—for propositional feature models—is given in Table 2.1, relating the hierarchy edge types—optional and mandatory features, or groups—to a conjunctive element in a corresponding propositional formula. Any

---

\(^8\)More precisely, Steven She’s PhD thesis, which is related to ours in the sense that it leverages our results for reverse-engineering feature models, as published in [SLB+11].
2. Background

Table 2.2.: Common reasoning tasks on feature models (adapted from [Men09])

<table>
<thead>
<tr>
<th>activity</th>
<th>task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging</td>
<td>• Checking satisfiability of models</td>
</tr>
<tr>
<td></td>
<td>• Detecting if a given feature is “dead”</td>
</tr>
<tr>
<td></td>
<td>• Detecting “dead” features</td>
</tr>
<tr>
<td>Refactoring</td>
<td>• Checking equivalence of feature models</td>
</tr>
<tr>
<td></td>
<td>• Checking extension of feature models</td>
</tr>
<tr>
<td>Configuration</td>
<td>• Checking specialization of feature models</td>
</tr>
<tr>
<td></td>
<td>• Validating partial or full configurations</td>
</tr>
<tr>
<td></td>
<td>• Calculating valid domains</td>
</tr>
<tr>
<td></td>
<td>• Enumerating one or more valid configurations</td>
</tr>
<tr>
<td></td>
<td>• Resolving conflicts in collaborative configuration</td>
</tr>
<tr>
<td>Measuring</td>
<td>• Counting valid configurations</td>
</tr>
<tr>
<td></td>
<td>• Computing variability factor</td>
</tr>
<tr>
<td></td>
<td>• Computing commonality of a model</td>
</tr>
</tbody>
</table>

additional cross-tree constraints are simply conjoined.

A Boolean variable is created for each feature, corresponding to the feature state—selected or unselected—in a configuration. All model constraints are represented in the formula. Thus, the configuration space is isomorphic to the set of all “models” (satisfying assignments) of the propositional formula.

Reasoning. Such a formula provides the basis for reasoning about models for debugging, refactoring, configuration and measurement [Men09]. Table 2.2 summarizes the most common reasoning tasks on feature models. A further overview of analysis techniques is provided in [BSRC10].

However, we will later see that all reasoners developed on top of propositional feature models cannot handle our real subject models. Thus, we develop propositional abstractions on those rich languages to provide approximations for such reasoning tasks (Section 4.4.9).

2.3.2.2. Denotational Semantics

We now formally define configuration space semantics in a denotational style [Sch87, Win93]. We define⁹ abstract syntax, semantic domain, and semantic function (or “semantic mapping”) according to [HR04]. Although translational semantics, such as the one above, or provided in [SLB+11], can be more concise, we use this style to prepare for the formal semantics we define for our formal language analysis later, with more expressive languages. For those, denotational semantics turned out to be most concise.

⁹Yet another formal semantics of feature models. However, we have not seen this style for defining feature model semantics before, though.
Furthermore, the definition provided here facilitates comparing the languages to feature models.

In contrast to the translational semantics above, we not restrict cross-tree constraints to propositional logic, since our sample model in Fig.2.6 already uses arithmetic operators. Consequently, we allow arbitrary values for features in our definition, but use $\bot$ for non-selected features. FODA only supported requires and excludes relationships (called composition rules [KCH+90]) between features. This restriction was seldomly relaxed in later variants of feature models, primarily to retain the feasibility of SAT-based model analyses.

**Abstract Syntax.** We first define an abstract syntax underlying the concrete syntax of feature models, as shown in Fig.2.6.

**Definition 1 (Feature Model)** Let $\text{Id}$ be a finite set of feature names (string labels), $\text{Group} = \{\text{or, xor}\}$ the types of feature groups supported in feature models, and $\text{Exp}(\text{Id})$ all possible expressions over $\text{Id}$. When $\text{FM}$ is the set of all possible feature models, each model $m \in \text{FM}$ is a set of features, so $\text{FM} = \mathcal{P}(\mathcal{F})$, where\(^{10}\):

$$\mathcal{F} = \text{Id} \times [\text{Id}] \times \mathbb{B} \times [\text{Group}] \times [\text{Exp}(\text{Id})]$$

(2.1)

If $(n,p,o,g,e) \in \mathcal{F}$, then $n$ is the name, $p$ is the parent of the feature ($p = \top$ for the root feature), $o$ determines whether the feature is optional (true) or mandatory (false), $g$ whether the feature represents a group constraint among its children, and $e$ is an additional cross-tree constraint per feature. Two well-formedness constraints hold: The parent relationship $p$ forms a rooted tree, and a feature representing a group ($g \neq \bot$) has at least one child. Note that for brevity, we disallow multiple groups under one parent, as shown in Fig.2.7.

![Figure 2.7.: Multiple groups under one feature, not allowed in our syntax](image)

For readers more familiar with metamodeling, Fig.2.8 shows a metamodel that also covers the abstract syntax of the previous feature model (Fig.2.6). However, it omits cross-tree constraints and has a different structure than our set-based definition above. Note that we and the metamodel avoid defining types of feature values.

\(^{10}\)Hint: $[X] = X \cup \{\top\}$ and $[X] = X \cup \{\bot\}$
2. **Background**

![Metamodel of feature models](image)

**Semantic Domain.** The semantic domain is the set of all possible configurations.

**Definition 2 (Configuration)** A configuration of a feature model is an assignment of values to features. Let \( \text{Conf}s \) be the set of all possible configurations, and \( \text{Val} \) all possible values (Boolean, string, number):

\[
\text{Conf}s = \text{Id} \rightarrow |\text{Val}|
\]

(2.2)

**Semantic Function.** The semantic function maps abstract syntax into the semantic domain, that is, a feature model to a set of valid configurations.

\[
[\cdot]_{\text{fm}} : \text{FM} \rightarrow \mathcal{P}(\text{Conf}s)
\]

(2.3)

**Definition 3 (Feature Model Semantics)** We define the semantic function as an intersection of denotations (valuation functions); however, we need to set the values of features not part of the model to \( \perp \).

\[
[\cdot]_{\text{fm}} = \bigcap_{f \in m} [\cdot]_{\text{tree}}(f) \cap \bigcap_{f \in m} [\cdot]_{\text{opt}}(n) \cap \bigcap_{f \in m} [\cdot]_{\text{group}}(n) \cap \bigcap_{f \in m} [\cdot]_{\text{constraints}}(n)
\]

\[\cap \{ \sigma \in \text{Conf}s | \sigma(f) = \perp \text{ for all } f \in \text{Id} \setminus \text{Id}(m) \} \]

(2.4)

We define the valuations as follows. For brevity, we introduce \( \sigma'(n) := \sigma(n) \neq \perp \), which determines whether a feature is selected. Our first valuation assures child-to-parent implications:

\[
[(n,p,\_\_\_,\_\_\_)]_{\text{tree}} = \{ \sigma \in \text{Conf}s | \sigma'(n) \rightarrow \sigma'(p) \}
\]

(2.5)

Mandatory features are also implied by their parent:

\[
[(n,p,\text{false},\_\_\_,\_\_\_)]_{\text{opt}} = \{ \sigma \in \text{Conf}s | \sigma'(p) \rightarrow \sigma'(n) \}
\]

(2.6)

For OR groups, at least one of the children should be selected:

\[
[(n,p,\_,\text{OR},\_\_\_)]_{\text{group}} = \left\{ \sigma \in \text{Conf}s \bigg| \bigvee_{(x=\_\_\_,\_\_\_,\_\_\_)} \sigma'(x) \right\}
\]

(2.7)
2.3. Feature Models

XOR groups are like OR groups, but allow at most one child selected:

\[
\llbracket (n, p, \_, \text{XOR}, \_) \rrbracket_{\text{group}} = \llbracket (n, p, \_, \text{OR}, \_) \rrbracket_{\text{group}} \cap
\left\{ \sigma \in \text{Conf} \mid \bigwedge_{(x, y = (\_, n, \_, \_, \_)) \in F, x \neq y} \neg (\sigma'(x) \land \sigma'(y)) \right\}
\] (2.8)

Finally, constraints of selected features have to hold (assuming an evaluation function under a configuration \( eval : \text{Exp}(\text{Id}) \times \text{Conf} \rightarrow \text{Val} \)):

\[
\llbracket (n, \_, \_, \_, e) \rrbracket_{\text{constraints}} = \{ \sigma \in \text{Conf} \mid \sigma'(n) \rightarrow eval(e, \sigma) \neq \perp \}
\] (2.9)

2.3.3. Feature Model Extensions

The original feature models of FODA have been extended in many ways. Kang [Kan09] provides a genealogy of successors introduced in the literature, shown in Fig. 2.9. Although many more variants exist, the major extensions from this genealogy comprise:

- **FORM feature models** [KKL+98] were introduced as part of the feature-oriented reuse method (FORM) and sub-divided models into four layers, from abstract on top to very concrete implementation-oriented features at the bottom. The high-level features are connected to low-level ones via a specific relationship type “implemented-by”. Thus, FORM represents problem space, solution space, and mapping in one layered model.
2. Background

- **FeatuRSEB feature models** [GFA98] were introduced with the FeatuRSEB methodology, aiming at integration with use case and similar models. They are mostly equivalent to FODA models, except that feature groups are referred to as variation points, and their children as variants. A new notation for OR and XOR groups is introduced, with the interpretation that they determine the binding time of a variation point: XOR variants are bound at build (reuse) time, and OR variants at run (use) time.

- **Hein et al. feature models** [HSVM00] introduce typed relationships (roles) and explicit binding times for features, based on industrial experience that FODA “does not provide the necessary expressiveness to represent the different types of crosslinks” in their application domain. These roles give rise to alternative hierarchical structures in one model. Consequently, the diagram is a directed acyclic graph (DAG), not a tree anymore.

- **Generative Programming feature models** [CE00] introduced the current notation, and OR groups. This notation was later extended with typed attributes and feature cardinalities in [COV02]. Furthermore, Riebisch et al. [RBSP02] introduced arbitrary group cardinalities and constraint notations. The most significant extension were feature cardinalities, however (also cf. [CK05, CHE05a]). Features (and their whole subtrees) can have more than one instance in a configuration, which has considerable impact on tools and reasoners. Only the tool FMP supports this kind of feature models.

In addition to these extensions—diagram shapes, layers, binding modes, expressive constraints, cardinalities, and typed edges—we find some further concepts acknowledged in literature. Among these are—although rarely—defaults [CE00, SBKM09] or visibility conditions [DHR10]. The latter are usually part of decision modeling languages, which share many commonalities with feature models [CGR+12].

2.3.4. Decision Models

Decision Modeling [SRG11, DG08] is another prominent variability modeling technique that became popular with the Synthesis method [HER93] for software reuse—introduced only three years after FODA. It models the commonality and variability of a software product line in terms of decisions that users need to make during product derivation, with dependencies among decisions. More precisely, a decision model is “a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products” [HER93].

In contrast to feature models, decision models lack a standardized graphical notation and are usually declared in a tabular (see for instance [SJ04, DGR11]) or a textual notation, as in the original Synthesis method. However, as speculated earlier, recent research has shown that feature and decision modeling are almost isomorphic and share most of their concepts—“cool” features map nicely to “tough” decisions [CGR+12].
2.3. Feature Models

Figure 2.10.: Feature model repository S.P.L.O.T.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Features</th>
<th>CTCs</th>
<th>Clause Density</th>
<th>Creator</th>
<th>Creation Date</th>
<th>Details and Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Car PL</td>
<td>9</td>
<td></td>
<td>44%</td>
<td>Jules White et. al</td>
<td>Aug 2009</td>
<td>Click</td>
</tr>
<tr>
<td>2</td>
<td>rttr</td>
<td>10</td>
<td>0%</td>
<td>N/A</td>
<td>tt</td>
<td></td>
<td>Click</td>
</tr>
<tr>
<td>3</td>
<td>Phone sdp</td>
<td>10</td>
<td>0%</td>
<td>N/A</td>
<td>cho</td>
<td></td>
<td>Click</td>
</tr>
<tr>
<td>4</td>
<td>e-shop</td>
<td>10</td>
<td>30%</td>
<td>0.7</td>
<td>Samuel Sepulveda, adapted from Wikipedia</td>
<td>July 2011</td>
<td>Click</td>
</tr>
<tr>
<td>5</td>
<td>Phone</td>
<td>10</td>
<td>50%</td>
<td>0.6</td>
<td>m</td>
<td></td>
<td>Click</td>
</tr>
<tr>
<td>6</td>
<td>AvionixFeatures</td>
<td>10</td>
<td>0%</td>
<td>N/A</td>
<td>ISD</td>
<td></td>
<td>Click</td>
</tr>
</tbody>
</table>

---

2.3.5. Model Repositories

Recognizing the lack of feature models available to researchers, a few repositories have been created to collect freely available models. Among them are S.P.L.O.T.\(^{11}\) [MBC09], the repository of the Generative Software Development (GSD) lab\(^{12}\), and the recently introduced repository SPL2Go\(^{13}\), which contains both models and the accompanying product lines’ source code.

Unfortunately, a closer look shows that the majority of models in these repositories are small and created by researchers. Fig. 2.10 shows a (truncated) screenshot of the S.P.L.O.T. repository. Currently, half of the 203 models have \(\leq 20\) features (average 28 features). The largest model comprises 290 features; however, it was created by researchers in the context of a Master’s thesis [Lau06]. Furthermore, the propositional language used for all the models in the repository is equivalent to generative programming feature models (without attributes and cardinalities), which at least threatens the practical relevance of the models. Similarly, SPL2Go contains 27 academic models with a maximum of 144 features (median 14, average 20).

Real industrial models have been reported, such as the one of HP’s Owen product line with over 2000 features [Ref09], or Bosch engine control software with even 5200 features [STB\(^{+}\)04]; but since they contain core strategic knowledge, it is almost impossible for research to study such models. Thus, the (also industrially used) open source models from the system software domain that we study in Chapter 5 are among the largest freely available variability models in existence today.

\(^{11}\)http://www.splot-research.org

\(^{12}\)http://fm.gsdlab.org

\(^{13}\)http://spl2go.cs.ovgu.de
2. Background

2.4. Software Ecosystems

Software product lines and software ecosystems are two clearly distinguished software engineering paradigms in the literature. Both aim at mass-customization, but approach variability—the diversity of systems they offer—in different ways. While SPLE fosters intra-organizational reuse and avoids variability that has no clear business advantage [CN01] by strict scoping, software ecosystems enable inter-organizational reuse by opening up the platform to third-party contributions and extensions [BBS10].

Software ecosystems are an emergent field of research and have been addressed from various perspectives. Unfortunately, research has not agreed on a crisp definition of ecosystems from the perspective of technology yet, although they are often seen as technical constructs [BA11]—arguably with fluid boundaries to related paradigms, such as distributed systems or componentware. We take the view of ecosystems being extensions of software product lines of substantial size [Bos09, JFB09, Rad12, VGP08]. We consider an ecosystem a large system composed of interrelated assets developed by communities of developers upon a common technological platform. Like in product lines, consumers derive products by making decisions in an automated, tool-supported process. We give a precise definition based on empirical data in Section 6.2.

However, since spatially distributed development, even for software product lines, and contributions from different companies (such as libraries or COTS components [Voa98, CL00]) are common practice, the real novel challenge of software ecosystems lies in the design, development, and establishment of an open platform. As pointed out earlier, we consider a platform open when consumers can utilize third-party contributions from a free market directly, having explicit technical tool support. In contrast, closed platforms require the integration of outside contributions into the platform first, usually with a controlled process.

2.4.1. Closed versus Open Platforms

Consider, for example, the Linux kernel, which is in fact more than just a traditional software product line. Although it uses mechanisms from SPLE and tightly manages variability, a free market of contributions (mostly drivers) has emerged around the main platform. The kernel development is spread over a broad community of internal and external developers, who build on each other’s solutions. Over the last six years, more than 6000 individuals from over 600 companies [CKHM10] have helped to more than double the Linux kernel code base from 3.5M to 7.9M lines of code (LOC). However, the Linux platform is predominantly closed, since additions need to be applied to the source tree, for example, as Git [Loe09] branches or patch sets. This “out-of-tree” development is actively discouraged [Cor04], and deriving such an instance is not supported by the configurator. Exceptions are some loadable kernel modules from commercial vendors.

In contrast, consider the Android application platform for mobile devices, which also manages huge variability, but in a more compositional and open way. Users derive a
2.4. Software Ecosystems

Figure 2.11.: Excerpt of a Debian manifest file

concrete system (on a mobile device) by selecting apps from an app store using an installer tool—in effect, composing their system from third-party components (apps). Apps run in a highly dynamic virtual machine and can interact via service-oriented facilities. In contrast to Linux, Android has no centralized variability model, but represents variability information decentrally, within the manifest file of each app. Android is also an ecosystem, spanning an industrial consortium developing the main platform, device providers, and a vast and vibrant market of third-party apps. In contrast to Linux Kernel’s respectable, yet controlled growth, the Android ecosystem has literally exploded with tremendous growth rates—similar to other mobile application platforms, such as iOS. Created just four years ago, the Android ecosystem boasts over 400,000 apps today.

Linux and Android are two prime examples of closed and open platforms, which are both highly successful, but approach variability differently—in terms of variability mechanisms as well as the organizational structures and business strategies. An apparent difference is that open platforms rely on manifests to express variability information, as shown in the excerpt of a Debian manifest in Fig. 2.11. The example contains meta data about the GNU/awk interpreter package, such as package name (line 1), version (line 2), dependencies (lines 4–5), or categorization information (lines 6–7).

So far, we can only speculate about further differences, given the current state of research on variability mechanisms in software ecosystems. We conjecture differences in the following three aspects:

- Asset packaging is a prerequisite for open platforms to support coarse-grained variability. We expect differences in packaging, encapsulation, and parameterization support; also in facilities for interactions.
- We conjecture that processes of making decisions, and how they are reflected in tools, differ. Not all platforms might support derivation of a whole product instance due to complexity reasons (Android handsets always come with pre-installed/pre-configured apps). Reconfiguration of an initial instance, on the other hand, requires special binding mechanisms, with implications to dependencies and tools.
- Interactions between assets introduce dependencies that are declared in variability models or manifests using constraint languages. Dependencies complicate development and maintenance, but also challenge derivation and reconfiguration tools. To
2. Background

understand how ecosystems cope with complexity, it is crucial to understand the dependency structures that tools and consumers manage. We expect the presence of variability models and the different granularities of variability to influence dependency structures.

2.4.2. Software Ecosystems in Research

Some of the currently largest software ecosystems have been reviewed, studied, or at least mentioned in research. All of the following software ecosystems aim at supporting mass-customization by leveraging technical variability mechanisms.

• The Eclipse IDE—a foundation for highly customizable development tools—appears to be one of the most recognized software ecosystems in research [MM12, vGPB10, Sch10, WG06, LBR09, LG05]. It was explicitly conceived as an ecosystem [MM12] and advertised as such by the Eclipse Foundation [Mil07]. The primary goal is to encourage contributions—to increase the variability available to customers—technically supported by Eclipse’s plugin mechanisms.

• Linux distributions maintain some among the largest software collections known [CZ10], using package management systems. Various works have studied Debian, such as from a software evolution [GBRM+09], a development organization [vGPB10], or a dependency management perspective [ADCBZ09, TLO10]. Others investigate Debian manifests and provide transformations to (subsets of) feature models and back [CZ10, GBS10]—unfortunately, without an experimental proof that such transformations can be beneficial to ecosystem practices.

• Studying ecosystems of mobile apps is a hot topic in the MSR community, but according to recent discussions in the MSR Vision 2020 summer school15 not sufficiently addressed in research yet—despite significant work on app security [EOM09, OMEM09, EOMC11]. However, Android is recognized as a highly relevant study object in general work on software ecosystems [BWB12]. Interestingly, Android not explicitly declares dependencies between apps, which means that apps are either self-contained or handle interactions fully dynamically. Analyzing Android’s variability mechanisms helps to understand how one of the largest and fastest-growing ecosystems today copes with complexity.

• Further ecosystems comprise, for example, Ruby [KJ11], MySQL [War12], or iPhone and Windows applications [Bos09, JFB09].

15http://msrcanada.org/msrvision2020/
This chapter shares content with the technical report “Variability Modeling in the Systems Software Domain” [BSL+12] and the paper “Variability Mechanisms in Software Ecosystems: Closed versus Open Platforms” [BPT+].

We provide an overview on work related to our empirical study of variability modeling. First, we describe existing studies of variability modeling languages; second, depict work on variability modeling in practice, such as experience reports; and third, report on variability modeling and analysis techniques that benefit from empirical work—such as ours—that provides assumptions about languages and models to evaluate techniques. Finally, we describe related work on software ecosystems, with a focus on comparative and exploratory work that investigates technical mechanisms, the relationship to software product lines, and the applicability of variability models. We will carefully anticipate some of our study results to explain the relationship of our dissertation to the discussed publications.

In general, our work is based on the observation that little empirical research on variability modeling exists. This subjective impression is confirmed by literature studies. A recent survey on the use of feature models [HCMH10] identified only five papers (2%) reporting practical experience. References 14, 16, and 17 in that survey are experiences from researchers applying feature modeling to sample problems from industry. References 31 and 37 therein are self-reported industry experiences: the first on using a feature modeling tool prototype on automotive control software, and the second one on managing avionic control software with feature models, but with few details on the languages and tools used. Another systematic literature review on variability management [CABA09] concludes that “there is only little, if any, experimental or detailed comparative analysis to show the relative advantages or disadvantages of different variability management approaches”. The authors argue that all approaches share similar concepts, and that a reference model would be needed for model transformations, tools, and further research.

3.1. Variability Modeling Languages

Concepts and semantics of variability modeling languages have been studied before.

Sinnema et al. [SD07] investigate six variability modeling languages (five academic, one commercial language, see Section 2.2) and provide a classification according to their
modeling capabilities and tool support. With regard to modeling, their conceptual
framework comprises the representation of choices (e.g. as features, decisions or variation
points) and of the product model (a configuration); abstraction support (e.g. with a
hierarchy or model layers); constraint languages; quality attributes; and support for
incomplete knowledge and imprecision. With regard to tooling, the authors investigate
whether the tool supports views, active specification (choice propagation to prevent
invalid future choices during configuration), configuration guidance (providing a workflow
through the choices), an inference engine, and effectuation (product derivation). The
authors evaluate and categorize each technique using a small sample product line.

The study shows that a broad range of techniques have been created in academia and
industry; however, a central conclusion is that:

"Most publications on the variability modeling techniques do claim that they
have been tested on one or more cases. These case studies, however, all seem to
involve very small configurable product families. The scalability and suitability
of the techniques with respect to other types of product families therefore
remains questionable until more extensive case studies are performed."

Furthermore, the authors point out the lack of defined modeling processes, in particular
to extract and evolve variability; and that the degree of formalization as required by
their variability modeling languages, is often hard to achieve in industrial settings, such
as for quality attributes.

Schobbens et al. [SHTB07]—as an extension to previous work [SHT06, BHST05,
BHST04]—survey seven feature modeling languages, each of which is a variant of FODA.
Arguing that most of them are not formally enough defined to avoid ambiguities, they
develop a common abstract syntax (Free Feature Diagrams) and define individual formal
semantics. They also introduce a new language (Varied Feature Diagrams) that is as
expressive, but more succinct than the others. Interestingly, the authors conclude that
many of the existing variants are expressively complete, thus, further extensions could
not be justified by expressiveness.

Czarnecki et al. [CGR+12] systematically compare feature modeling with decision
modeling [DG08] (cf. Section 2.3.4). The authors compare both techniques on ten
dimensions, which were inspired by our work (from [BSL+10]), and that of Schmid et
al. [SRG11] on decision modeling). They also establish a mapping of the two techniques
to Kconfig, CDL, and a previous proposal (initial submission of [CVL12]) of CVL [Ohl09].
The study concludes that there are no major conceptual differences between feature and
decision modeling—except for the support of modeling commonality (via mandatory
features) in feature models, as decision models focus purely on variability.

Our work complements these three—purely qualitative—studies mainly in two respects:
we analyze languages originating from practice in their full richness, and we quantitatively
analyze their real, large-scale instances. In consequence, we provide empirical evidence
for the occurrence and frequency of variability modeling concepts in practice. Compared
to Sinnema et al. [SD07], we study language concepts on a more fine-grained level and
also reverse-engineer and analyze their formal semantics. Compared to Schobbens et
al. [SHTB07], our quantitative analysis shows that more advanced concepts than found in the FODA variants are commonly used, which challenges their conclusion about expressiveness. With Czarnecki et al.’s work [CGR+12], we can also confirm the use of decision modeling concepts in practice. Studying real decision models would be valuable future work, however.

Finally, many smaller qualitative comparative studies exist, such as Haugen et al. [HMPO05], who provide a high-level framework to compare variability modeling approaches; Günther et al. [GCJ12], who elaborate on the design space of variability modeling languages from a DSL perspective; Trigaux et al. [TH03], who compare feature models, class diagrams, and use cases to specify requirements with variability; or Istoan et al [IKPJ11], who discuss syntactic differences of variability modeling approaches based on a literature review. Schmid et al. [SRG11] study commonalities and differences of several decision modeling languages with regard to decision representation (data types), constraints, code mapping, and further concepts, such as modularization. They conclude that, even though all techniques share a common set of concepts, small deviations exist.

3.2. Variability Modeling in Practice

Although detailed experimental work or experience reports on variability modeling are sparse, notable exceptions exist.

Grünbacher et al. [GRDL09] report on the industrial use of their Dopler tool suite for variability modeling and product derivation. It has been used by Siemens VAI\(^1\) to automate component-based software development since 2007 and to manage Eclipse-based tools. While the language and its semantics are formally defined [DHR10], unfortunately, neither the models nor further empirical data is available. In line with our later findings, the authors emphasize the need for domain-specific adaptations of tools and languages in various papers [DRGN07, DGR07a], with [DGR07b] focusing directly on this topic.

Riebisch et al. [RSP04] report results of a workshop on the industrial applicability of feature models. Relevant to our work is their discussion of the role of feature models, their classification of features, their elaboration on reasoning support in tools, and reported industrial experiences. The authors emphasize that feature models should be used orthogonally to other artifacts, such as models and code, and that feature models should be used by consumers (e.g. product managers, customers, merchants) and by technically skilled suppliers (e.g. developers, architects). Furthermore, they classify features into functional features (expressing behavior), interface features (product conformance to external standard or subsystem), and parameter features (“enumerable, listable environmental or non-functional properties”). We identify similar kinds of features, but distinguish between kinds of features in the language, and themes of features in the models, not indicated by syntax. We will also see that many features have unbounded value domains, beyond enumerations, which challenges reasoners. Finally, the report briefly describes two real product lines: a car periphery system with around

\(^1\)http://www.siemens-vai.com
3. State of Research

200 features, and a yard inventory system for steel manufacturing comprising 1500 features. Unfortunately, no further details on the models are given. Interestingly, the report estimated that the actual benefit of variability modeling rather lies in reduced time-to-market, less in cost reductions.

Reiser et al. [RTW07] report industrial experiences on variability modeling from the automotive domain. They sketch a framework based on FODA and define seven requirements for highly flexible variability modeling: (1) principal feature modeling concepts with some extensions, (2) feature meta information with typed attributes, (3) determined order of features (for wide and shallow trees), (4) domain-independence without project-specific cases, (5) formal foundation, (6) open reference implementation and mapping to XML, and (7) compliance constraints (restrict modeling to a subset of concepts). Our own study confirms requirements 1, 3, and partly 6; refutes 4; and shows absence of 2, 5, partly 6 (no XML mapping), and 7 in our languages.

Gillan et al. [GKS+07] report on application challenges of feature modeling in the telecommunications domain. They conclude that there are many ways to express a feature model for a telecommunications system, which calls for research on methodologies for variability modeling. We confirm the absence of documented methodologies for our languages.

Czarnecki et al. [CBUE02] provide an experience report on the applicability of generative programming for embedded systems. Therefore, they introduce an extended notation of feature models (feature cardinalities, typed attributes, and attribute references), discuss configurator tool support for feature models, and the role of static configuration in embedded systems. They conclude that generative programming is applicable for the embedded domain; however, provide no further information on the feature model.

3.2.1. Experiments

Recognizing the lack of experience reports, researchers themselves have performed case studies on variability modeling.

Hubaux et al. [HHB08] present a case study on reverse-engineering variability models. They migrate the heterogeneous configuration mechanisms of PloneGov [DMH+07] to a feature-oriented approach, unifying its configurability into a feature model. The authors report challenges, such as modeling binding times, large numbers (>50) of direct children, or the need to introduce intermediate derived features to optimize dependencies. Unfortunately, neither the size nor further statistics about the model are available.

Kästner et al. [KAB07] refactor the Berkeley database into a configurable product line, concluding that very fine-grained variability mechanisms are necessary (even to split expressions in IF statements), beyond capabilities of aspect-oriented frameworks. The relatively small model (38 features) is freely available. Unfortunately, both case studies are performed by researchers and neither product line went into production.
3.2.2. Industrial Experience Reports

Experience reports written by practitioners instead of researchers provide valuable direct insights into industrial practices.

Jepsen et al. [JB09] from Danfoss Drives\(^2\) describe their experience of introducing a product line of frequency converters iteratively from a clone-and-own approach to a configurable platform. They provide very detailed information on the process, on obstacles and their solutions, both from a technical and organizational perspective. Unfortunately, no details on the feature model are reported.

Lee et al. [LKK] report experiences from developing an elevator control software product line. Their feature model comprises 490 features and is, according to FORM, divided into the four layers: capability, operating environment, domain technology, and implementation technique. However, only a small part of the whole model is shown in the report, which was created by eight domain experts, two methodologists, and one moderator within a period of three months. The authors emphasize that strict scoping and a standardized domain terminology are important to prevent “wasteful discussions” and a complicated, redundant feature model.

3.2.3. Variability Model Evolution

In [LSB\(^+\)10], we study the evolution of the Linux model. Specifically, we investigate how the statistics from our workshop paper [SLB\(^+\)10] have evolved over the last five years, and we classify the types of edits applied to the model. The analysis shows that the number of dependencies has grown proportionally to the number of features over the last five years.

Passos et al. [PCW12]—building on results of our work—study the co-evolution of the Linux variability model, the build system, and the code, in order to identify common patterns of variability evolution. They conclude that evolution of the variability model only gives a very narrow picture of the real evolution. To provide guidance and tool support, this information has to be complemented with insights from build system and code evolution.

3.3. Tools and Evaluation

Chen et al. [CABA09] conduct a systematic literature review on reported evaluations of variability management techniques. They conclude that a majority of the techniques lacks a robust evaluation: 80% of the inspected publications were evaluated in terms of an experience report or a discussion, which both lack scientific standards. Furthermore, 96% were evaluated in a single study, and 71% never faced an industrial setting.

**Tools.** Requirements for tools have been proposed in research. The tool survey by Djebbi et al. [DSF07] investigates four variability modeling tools and compares their\(^2\)http://www.danfoss.com/businessareas/drivesolutions
3. State of Research

capabilities with claimed expectations from industry. The catalog of 34 expectations comprises, among others, modeling requirements such as attribute support and the usage of FODA-like concepts, comprising mandatory and optional features, feature decomposition (hierarchy), cardinalities, and cross-tree dependencies. Unfortunately, the source of the large number of requirements remains unclear, for example, whether they were systematically elicited using a questionnaire or interviews.

In another recent expert survey on requirements for product derivation [RGD10], interactive support for resolving variability was ranked highest. This support requires adequate, scalable model reasoners. Finally, a study on configuration challenges in Linux and eCos [HXC12]—performed by surveying actual users—emphasizes the lack of guidance for making choices and the low quality of advice offered by the configurators.

**Benchmarks.** Using benchmarks or generating variability models are common approaches to evaluate new variability modeling tools or techniques. For both, realistic assumptions about structure and constraints of real models are crucial. The works of Thüm et al. [TBK09] and Mendonça et al. [MWC09] are two such examples. Both present reasoning techniques for feature models and rely on model assumptions for evaluation, which we will challenge later.

Furthermore, Segura et al. [SGB+12] introduce a framework for testing and benchmarking feature modeling analysis tools, after recognizing the lack of such [BRCTS06, SRC09]. Unfortunately, the feature model generator requires parameters as input, which the user has to provide.

**Scalability.** In another systematic literature review, Chen et al. [CAB09] review the scalability of variability modeling techniques. They aim to i) find evidence that scalability is important, ii) identify the mechanisms that were proposed to increase scalability, and iii) determine evaluation approaches to variability modeling techniques. With respect to i), the authors conclude that the majority of publications neglects discussing scalability of their techniques. However, they identify six publications that emphasize scalability. Interestingly, one even discusses downward scaling—the applicability to very small projects. With respect to ii), the study identifies ten techniques or principles to achieve scalability, such as modularization, hierarchy, views or automated tool support. Finally, for iii), the authors conclude that a majority of variability modeling approaches that claim scalability have not been sufficiently evaluated in that respect. Therefore, they demand that “variability modeling approaches should be preferably tested on large product lines in real industrial settings instead of ‘toy’ systems.”

3.4. Variability in Open Source Projects

Research community has recognized the appeal of studying configurable open source software due to large source code archives available.

In particular the Linux Kernel has been a frequent study object; several variability-related aspects have been addressed. Sincero et al. [SSPS07] are the first to discuss
3.5. Knowledge-Based Configuration

Recognizing many overlaps between software configuration, and the older AI-related field of knowledge-based configuration, recent research has started to investigate their relationships, including work on leveraging product configurators and AI techniques for software product lines [AMS04, HWK+06, ASM04]. Although the relationships between software and product configuration are blurred and part of ongoing research, the following works are closely related to ours.

Hubaux et al. [HJD+12] present a research agenda on unifying software and product configuration. Their comparison of both fields concludes that software configuration can benefit from existing techniques in product configuration, such as in the expressiveness of modeling languages and reasoning support (e.g. to optimize configurations according to certain criteria). Notably, they emphasize that both fields lack research on evolution of models.

Very recently, Abbasi et al. [AHA+12] empirically study 111 web configurators. Their conceptual framework to classify the investigated cases comprises the visualization of the configuration options, the handling of constraints, and the type of configuration process supported. They develop a JavaScript-based tool infrastructure to semi-automatically extract datasets (e.g. configuration options and attributes) from the web-based configurators. Among others, the authors confirm that hierarchical organization and grouping of configuration options is commonly used. XOR groups are the most frequent kind of grouping with constraints. Furthermore, cross-tree constraints exist. The authors also identify limitations in the reasoning procedures, with regard to reliability and runtime efficiency.

Rabiser et al. [RGL12] study user guidance support in product configurators. They identify seven core capabilities from the literature, implement these in the DOPLER tools suite (see Section 2.2), and evaluate each of which in a user study with industrial
3. State of Research

participants. Among others, capabilities such as visibility control (hide and show options), views and filters, or freedom in navigation are very important, while immediate feedback turns out to be hard to comprehend for users. Reset and undo functionality is essential to experiment with choices and their impact.

3.6. Software Ecosystems

General work on software ecosystems targets business, strategic, and organizational aspects [BWB12]. Barbosa et al. [BA11] review publications on software ecosystems using a systematic mapping study. They confirm that ecosystem are technical constructs, related to open source software and SPLE; however, none of their identified publications covers technical mechanisms to support variability.

Bosch [Bos09] presents a taxonomy of software ecosystems, applicable to all of our subjects. He takes the perspective of economical incentives of open platforms for software development. Main characteristics include value and attractiveness offered to existing and new users, collaborations with partners, and the practical scalability of ecosystems. Later, Bosch et al. [BBS10] study companies relying on software product lines and on software ecosystems, and the problems resulting from these approaches. The authors describe software ecosystems as the logical destiny of successful product lines. While both leverage a platform to build products, a major property of software ecosystems is that developers extending the platform can stem from other organizations.

Kabbedijk et al. [KJ11] study defining characteristics of open-source ecosystems using the Ruby ecosystem, which also relies on manifests to represent variability. They focus on the role of developers and basic units (gems). We study similar subjects, and our conceptual framework (summarized in Section 7.1.1) includes both of these concepts and.

Messerschmitt et al. [MS03] characterize software ecosystems according to their context, which, notably, also comprises technical aspects. They differentiate between organizational, business, and technical aspects of software manufacturing; and identify stakeholders and their interests and views on software ecosystems. Our work complements theirs with an empirical study of existing software ecosystems.

Jansen et al. [JFB09] present a research agenda for software ecosystems. They propose to study ecosystems such as the MySQL/PHP, Microsoft Windows, and iPhone apps. We deliver on this agenda by investigating similar ecosystems. They announce the characterization and modeling of software ecosystems as a main challenge—which we address with our conceptual framework in Section 6.2 and Section 7.1.1.

3.6.1. Development Processes

Development processes in software ecosystems are studied by van Gurp et al. [vGPB10], who analyze the Eclipse and Debian ecosystems. They show that indeed large-scale open software development can be performed successfully using practices not seen in SPLE. Furthermore, they conclude that in open compositional development, requirements of components are often developed independently by separate teams, which leads to
increased integration and testing effort, however. We extend this research by analyzing a larger variety of ecosystems with a closer look on their organizational structures, and relating them to technical aspects.

3.6.2. Relationship between Variability Models and Manifests

The relationship between manifest files and variability models has, to some extent, been investigated in the literature. Schmid [Sch10] explores modeling concepts in Debian package manifests and Eclipse bundle manifests with regard to distributed development, and relates these to feature modeling concepts. Cosmo et al. [CZ10] and Galindo et al. [GBS10] show that subsets of variability models can easily be converted into a Debian package structure with manifest files and back. More precisely, Cosmo et al. encode a subset of Free Feature Diagrams (cf. Section 3.1), arguing that this transformation enables the use of Debian package manifest reasoners to analyze feature models. Galindo et al. propose DebianVML—a graphical language to express the variability of the Debian package ecosystem and to enable their analysis using propositional logics. The extracted models are suitable for consistency checking and benchmarking feature modeling tools. Unfortunately, both works lack an experimental evaluation on the benefit of these transformations.

3.7. Conclusions

Our literature overview shows that variability modeling is a rich field with many techniques, formalizations, and tools. However, one of the major research issues in variability modeling remains: verifying the feasibility of applying academic languages and techniques in industrial settings, and exploring the knowledge that exists in practice, to systematize and make it available for research. Many of the surveys described in this chapter emphasize that most techniques were solely evaluated on hand-crafted toy examples, or generated models based on assumptions.

The shift from product lines to ecosystems of software has been recognized in the literature. Organizational aspects, that is, processes and business strategies, are reasonably well studied. However, we observe a lack of knowledge regarding the technical aspects of such ecosystems. In particular, we are not aware of literature researching organizational and business aspects of software ecosystems trying to reveal correlations between them, and investigating causalities of technical mechanisms.
Chapter 4

Variability Modeling Languages

This chapter shares content with the ASE’10 paper “Variability Modeling in the Real: A Perspective from the Operating Systems Domain” [BSL+10], the technical note “Formal Semantics of the CDL language” [BS10], and the technical report “Feature-to-Code Mapping in Two Large Product Lines” [BSCW10a].

We qualitatively analyze the variability modeling concepts and the tool support of two open source languages: CDL and Kconfig. Both are textual domain-specific languages that were conceived to describe the valid configurations of their host projects—the eCos operating system and the Linux kernel. After describing our methodology, we introduce the languages, explain our conceptual framework, and report the results of our language analysis, backed up by reverse-engineered formal semantics. We also inspect the configurators of both languages with respect to their capabilities and limitations to drive the product derivation process. This chapter addresses our first research question RQ1.

4.1. Methodology

To directly compare the two languages, we reverse-engineer specifications of the syntax and the configuration space semantics of CDL and Kconfig. Therefore, we analyze user manuals, extensively test the tools both on actual models and on manually created examples, and inspect the tool implementations, which are available as open source. This process allows us to understand the languages in depth and discover many subtle differences and connections. On top of the full semantics, we develop propositional abstractions as a prerequisite for SAT-based analyses. Fig. 4.1 illustrates this approach.

To address research question RQ1.1, we use the concepts of FODA feature models as a reference and identify semantically corresponding concepts in CDL and Kconfig, to establish a mapping between the three languages. Following RQ1.2 and RQ1.3, we identify concepts beyond FODA in the languages and characterize their semantics. For RQ1.4, we inspect the configurators and their source code with respect to configuration process, user assistance, and reasoning support—in particular, the facilities to propagate choices and to resolve conflicts (unsatisfied constraints) in a configuration.

To increase understandability for the reader, we will describe the formal semantics after reporting the results of the language analysis in the remainder of this chapter.
4. Variability Modeling Languages

4.2. Language Introduction

We first introduce the CDL and Kconfig language and briefly describe the open source projects they were designed for. We use the feature model from Section 2.3 (Fig. 2.6) as our running example. Fig. 4.3 shows this model expressed in Kconfig (to the left) and CDL (to the right). Both snippets are extracted from the original Linux and eCos models. They define the features of the Journalling Flash File System version 2 (JFFS2), supported by both systems. In fact, eCos’ JFFS2 implementation was ported from Linux. JFFS2 is one of the very few of such ports, but it makes an ideal example to illustrate the similarities and differences between Kconfig and CDL. To give a realistic impression of both languages, we keep the examples close to the originals; in particular, we retain the original identifiers, which differ somewhat from the names in Fig. 4.3. The few lines introduced purely for the purpose of the example are underlined and we leave out some unnecessary parts of the corresponding sources to avoid clutter.

Fig. 4.2 gives a glimpse of the accompanying GUI-based configurators of CDL (configtool, to the left) and Kconfig (xconfig, to the right) that support users in creating a legal configuration of a given model.

4.2.1. eCos and the Component Definition Language

The CDL language was specifically developed as part of eCos, a free real-time operating system for deeply embedded applications. Requirements of this domain comprise, for example, small code image sizes, low resource usage, and a high degree of portability. eCos is used, among others, in consumer electronics, networking, automotive devices, and even satellite and space-based devices, with a reported\(^1\) market share of 5–6% in the embedded operating systems market.

We study version 3.0 of eCos, which supports 116 hardware architectures—called targets—and comprises almost a million LOC. The code base is divided into 500

\(^1\)http://ecoscentric.com/ecos/examples.shtml
4.2. Language Introduction

(a) configtool

(b) xconfig

Figure 4.2.: Configurators of CDL (left) and Kconfig (right)

packages, each containing the source code and a set of CDL files declaring the variability of the package. Each target defines a set of packages specific to the architecture. So-called templates aggregate packages with architecture-independent functionality. In the configurator, a user first selects one of the 116 targets and then one of nine predefined templates (e.g. default, min, all). Finally, the user may decide to load additional packages. The configurator then aggregates all partial variability models of the loaded packages into a single configuration tree.

The CDL language that is used to define the partial models associated with packages, is an internal DSL embedded in Tcl—a dynamic and highly extensible scripting language. CDL inherits characteristics from Tcl, such as syntactic nesting of blocks, the ability to embed Tcl control structures (conditional statements, for loops), dynamic typing of values, and a rich set of operators in constraint expressions. CDL’s configurator configtool incorporates an inference engine to support choice propagation and interactive conflict resolution.

4.2.2. Linux and the Kernel Configuration Language

Kconfig is a standalone domain-specific language that is used to specify build-time configurations of the Linux kernel since 2002. The graphical configurator xconfig reads the Kconfig model and allows users to select features in a user interface closely resembling the CDL configurator of Fig. 4.2a. It outputs a set of feature symbol and value mappings that are referenced in Makefiles and in the source code, as described earlier in Section 2.1.2.

The studied Linux version 2.6.32 supports 23 hardware architectures. The code base spans 1880 directories and 701 Kconfig files. Kconfig models are distributed over multiple files, organized according to the source code hierarchy. Each Kconfig specification is placed alongside the related code. An architecture-specific Kconfig file is used as a
starting point for the specification, loading other files with a simple inclusion mechanism.

4. Variability Modeling Languages

4.3. Conceptual Framework

To compare the languages, we first describe the conceptual framework that emerged from our qualitative analysis of the languages. It is a refinement of our summarized framework in Section 7.1.1, and represented by the left-most columns of Table 4.1. We will use it to explain the key similarities and differences between the languages.

4.3.1. Feature Kinds

In our two subject languages, features are labels with metadata organized in a hierarchy, as known from most variability modeling languages. The features have different characteristics according to i) their purpose for the hierarchical organization and ii) the role they play for configuring the project, for example, whether they can be mapped to source code, and how they may be referenced therein. Thus, we introduce two orthogonal classifications for these different kinds of features (first row in Table 4.1): i) we distinguish between grouping and individual features; and ii) between various roles that features can take. In both languages, these different kinds of features are defined using specialized keywords, resembling the project-specific terminology of eCos and Linux.

Grouping and individual features. Grouping features are used to structure models by gathering a set of features as their children. Nevertheless, grouping features can also provide configuration options. An example is the “Journalling Flash File System” in Fig. 4.3. Some grouping features further impose cardinality constraints on their children (see Section 4.3.4), such as the exclusive choice “Default Compression” in Fig. 4.3, which has exactly one selectable child at a time. In contrast, individual features have no children; they are leaves in the hierarchy and are purely used for providing configuration options.

Roles of features. Features that represent configuration options can take one or more of the following roles:

1. User feature: a configuration option that can be set by the user in a configurator, like all active (not grayed-out) features shown in Fig. 4.2a;
2. Implementation feature: a configuration option accessed by the build system or a generator, like those referenced with #IF and #IFDEF preprocessor directives in the Linux code excerpts in Fig. 4.4;
3. Derived feature: a configuration option automatically computed via constraints, such as the feature "JFFS2 FS tests” with a grayed-out value in Fig. 4.2a.
4. Capability: an abstraction of functionality that can be provided by several features interchangeably. For example, the Linux’ HAVE_IDE feature represents hardware IDE support. Other features can depend on this capability instead of on a concrete IDE controller, which reduces coupling.
Figure 4.3.: Model excerpts expressed in CDL (left) and Kconfig (right), largely resembling the feature model above. Corresponding definitions are aligned.
4. Variability Modeling Languages

```c
#if CONFIG_JFFS2_FS_DEBUG > 0
  /* Enable "paranoia" checks and dumps */
#define JFFS2_DBG_PARANOIA_CHECKS
#define JFFS2_DBG_DUMPS
...
#endif

#ifdef CONFIG_JFFS2_ZLIB
  jffs2_zlib_init();
#endif
```

Figure 4.4: Feature symbols referenced in code (JFFS2 code excerpt, occurring in eCos and Linux)

### 4.3.2. Feature Representation

With feature representation, we refer to the way values of features are expressed in a configuration, that is, whether a feature is abstract and has no value (*none* features), just a truth value (*switch* features), or whether data values are supported (*data* features), and how both values interact. Recall that in feature modeling, features are often only considered to be of type *switch*, while features with a data value are referred to as features with attributes (cf. Section 2.3).

Recognizing that our two subject languages differ with respect to feature representation, we characterize the values that features can take in a configuration. The second row of Table 4.1 lists feature types supported in each language. We also describe the semantic domain of each language, since feature value ranges and their possible combinations depend on the configuration space of a model—the set of valid configurations, which can be empty if the model is overconstrained.

### 4.3.3. Feature Hierarchy

All major variability modeling languages stemming from academia admit a single feature hierarchy in the model, which is then reused in the respective configuration tools. In the FODA example in Fig.2.6 and Fig.4.3, the diagrammatic tree represents both the intended configuration hierarchy and the syntactic nesting.

With feature hierarchy (Table 4.1, row 3), we characterize the realization of the feature tree in our subject languages. Interestingly, the hierarchies displayed in the CDL and Kconfig configurators deviate from the syntactic structure of the models. Thus, we distinguish between the syntactic model hierarchy and the configurator hierarchy. The former is given by the syntactic nesting of features in the model, while the latter is shown to the user in the configurator, as in Fig.4.2.

A main property of the feature hierarchy in FODA-like languages is that the presence of a child feature implies the presence of its parent, that is, for each edge from child \( c \) to parent \( p \), we have that for a configuration \( \sigma: \sigma(c) \rightarrow \sigma(p) \). We will see that Kconfig partly violates this rule.

Finally, most hierarchies of variability modeling languages are trees instead of just directed acyclic graphs (DAGs). We also classify languages according to this aspect,
in particular, whether explicit root nodes for the feature trees are declared, or are synthetically introduced by the configurator, which enables working with diagrams that are forests and not trees like in FODA.

4.3.4. Feature Groups

Feature groups are a core modeling concept of feature models. They restrict the number of selectable sibling features if their parent is selected (Table 4.1, row 4): exactly one child for XOR, at least one for OR, and at most one for MUTEX. Alternatively, the constraint can be given as an interval in some extensions to feature models (cf. Section 2.3.3). We will observe that CDL also supports feature groups that cross-cut the hierarchy, which we have not seen in any other variability modeling language.

4.3.5. Feature Constraints

In addition to the constraints stemming from hierarchy and feature groups, most variability modeling languages support cross-tree constraints, which restrict the configuration space in addition to the feature hierarchy. We identified three types of constraints (Table 4.1, row 5) in our subject languages:

1. **Configuration constraints** restrict the legal combinations and values of features. In feature modeling, they include child-to-parent implications (biimplications for mandatory features), group cardinalities, and cross-tree constraints.
2. **Defaults** provide default values for features, possibly depending on other features (computed defaults). They can be overridden by the user.
3. **Visibility conditions** control the visibility of features in the configurator GUI. Features whose visibility condition is false are not shown or otherwise disabled for user input.

Computed defaults and visibility conditions have not been widely considered in feature modeling. Unlike configuration constraints, defaults and visibility conditions have no direct impact on the configuration semantics. However, they interact with each other in complex ways in our subject languages and may impact configuration semantics. Furthermore, certain configuration constraints exist that allow restring the domains of feature data values using string and arithmetic constraints, and to control the binding mode of features.

4.3.6. Further Concepts

We identified further concepts in our subject languages, such as: **textual content** to provide natural language descriptions for features (Table 4.1, row 6) and to help users elicit a configuration decision; **modularization** concepts to divide the models into parts, ranging from static source inclusion in Kconfig to more complex mechanisms for dynamic loading of packages during configuration in CDL; and **code mapping** techniques to control the inclusion of particular source files in an instance when certain features are selected, ranging from imperative build logic to declarative specifications.
<table>
<thead>
<tr>
<th>Concept</th>
<th>Kconfig</th>
<th>CDL</th>
<th>Feature Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feature Kinds</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grouping</td>
<td>menu, menuconfig, choice</td>
<td>package, component, interface feature</td>
<td>feature</td>
</tr>
<tr>
<td>Individual</td>
<td>config option</td>
<td>feature</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>package, component, option</td>
<td>feature</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>config, menuconfig</td>
<td>N/A</td>
<td>all except features with no_default feature</td>
</tr>
<tr>
<td>Derived</td>
<td>no or conditional prompt</td>
<td>calculated, interface</td>
<td>N/A</td>
</tr>
<tr>
<td>Capability</td>
<td>N/A</td>
<td>TBD</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Feature Representation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composition</td>
<td>single value</td>
<td>bool. value with opt. attribute</td>
<td>feature</td>
</tr>
<tr>
<td></td>
<td>n/a</td>
<td>bool. value with opt. attribute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Feature Constraints</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>select requires, active_if</td>
<td>cross-tree constraint</td>
<td></td>
</tr>
<tr>
<td>Value restrictions</td>
<td>range legal_values</td>
<td>cross-tree constraint</td>
<td></td>
</tr>
<tr>
<td>Derived features</td>
<td>non-prompt default, default</td>
<td>calculated, interface</td>
<td>rare [DHR10]</td>
</tr>
<tr>
<td>Defaults</td>
<td>prompt default, default_value</td>
<td>default_value</td>
<td>rare [CE00, SBKM09]</td>
</tr>
<tr>
<td>Visibility</td>
<td>prompt condition</td>
<td>active_if</td>
<td>rare [DHR10]</td>
</tr>
<tr>
<td>Expression operators</td>
<td>&amp;&amp;,</td>
<td></td>
<td>,</td>
</tr>
<tr>
<td>Binding modes</td>
<td>three-value logic, tristate choice, boolean choice</td>
<td>interface constraint, int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Textual Content</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prompt, help display, description</td>
<td>description</td>
<td></td>
</tr>
<tr>
<td><strong>Modularization</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>textual inclusion, dynamic loading/unloading</td>
<td>dynamic loading/unloading</td>
<td>rare [BCFH10]</td>
</tr>
<tr>
<td>Build symbols</td>
<td>no, uses KBuild (m:n)</td>
<td>one-to-many, one-to-one</td>
<td>and build specifications</td>
</tr>
<tr>
<td>Code mappings</td>
<td>no, uses KBuild (m:n)</td>
<td>one-to-many</td>
<td>and build specifications</td>
</tr>
</tbody>
</table>

Table 4.1: Mapping of concepts between Kconfig, CDL, and feature modeling.
4.4. The CDL Language

In our study, we identified the following characteristics of CDL. We describe the language according to our conceptual framework and later provide formal configuration space semantics. To illustrate its language concepts, we will refer to specific lines in Fig. 4.3.

4.4.1. Feature Kinds

In CDL, feature kinds reflect types of implementation artifacts they map to: packages are top-level containers for features, mapping to eCos packages. Components are nested features grouping other features. Options are atomic configuration options appearing as leaves that are nested under packages or components. Several—possibly exclusive—features can provide equivalent functionality required elsewhere. Interfaces represent such capabilities. In our example, Line c-9 states that CYGPKG_FS_JFFS2 implements the interface CYGINT_IO_FILEIO (not shown). The value of an interface is the number of features currently in the configuration and implementing it (with the implements keyword). Declaring constraints over this value allows to impose cardinality constraints on the implementing features.

Packages and components represent both grouping and individual features; options and interfaces are always individual and cannot group features. Options are always leaves in the tree. Although interfaces can have children, they themselves are never visible in the configurator.

By default, all features can be implementation features, unless they explicitly do not define a symbol with the keyword no_define. Being a user or derived feature is determined by the declared constraints, except for interfaces, which are are always derived and not shown to the user. Interfaces explicitly represent capabilities.

4.4.2. Feature Representation

In CDL, every feature is composed of two values: an enabled value and a data value. The enabled value is a Boolean and encodes the presence or absence of the feature. The data value is dynamically typed and used to store numbers and strings. Thus, a configuration maps feature names \( f \in \text{Id} \) to value pairs, as explained in more detail later:\(^2\):

\[
\sigma: \text{Id} \mapsto \{0,1\} \times \text{Data}
\]

and if \( \sigma(f) = (e,d) \), then \( d \in \text{type-of}(f) \)

The CDL terminology for a feature type is flavor. Flavors map to FODA features as follows:

\(^2\)In the formal semantics in Section 4.4.8, we will see that the configuration is slightly more complicated than given in this simplified form, since we have to distinguish between a feature being selected (enabled value), and whether it is actually active (influencing product derivation) given its constraints (enabled state).
4. Variability Modeling Languages

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Mandatory with no attribute</td>
</tr>
<tr>
<td>bool</td>
<td>Optional with no attribute</td>
</tr>
<tr>
<td>data</td>
<td>Mandatory with attribute</td>
</tr>
<tr>
<td>booldata</td>
<td>Optional with attribute</td>
</tr>
</tbody>
</table>

The flavor instructs the configurator to show a checkbox for bool, a textfield for data, both for booldata, and just a label for none features. Interestingly, features with the flavors none and data can be made optional by using configuration constraints (explained shortly), which differs from feature modeling, where cross-tree constraints of a mandatory feature also restrict the parent feature.

The dynamic typing of the data value has the following consequences. In the configurator, if the user inputs a signed long literal written in decimal, octal or hexadecimal, it is interpreted as an integer. If the number contains a radix point, it is interpreted as float. Other input is considered as a string. Booleans are denoted by integers: 0 means false, and 1 means true. These types are dynamically converted when needed. For example, an addition of the empty string to the number 2 results in 2, since the empty string is implicitly converted to 0.

The example model in Fig.4.3 includes features of various flavors. CYGOPT_FS_JFFS2_DEBUG (c-13) of flavor data takes numeric values. CYGOPT_JFFS2_NAND (c-22) takes Boolean values (flavor bool), and the data feature CYGOPT_FS_JFFS2_COMPRESS_CMODE (c-38) assumes string values.

4.4.3. Feature Hierarchy

The syntactic model hierarchy is given by the nesting of options and components under other components and packages in CDL. The configurator hierarchy follows the syntactic nesting of features, unless declared otherwise. Re-parenting is a mechanism to explicitly specify a different parent for a feature than its syntactic scope in the model (see Line c-10). It allows adjusting the developer-oriented structure of the model, which is primarily driven by eCos’ packaging mechanism, to a slightly more user-oriented view, before it is shown in the configurator.

CDL’s configurator hierarchy rules are similar to feature modeling: for each edge from child c to parent p, \( \sigma(c) \rightarrow \sigma(p) \). However, this property does not hold for the syntactic model hierarchy, since re-parented features not imply there syntactic parent anymore.

Finally, the CDL configurator shows a synthetic root—a fresh root node not specified in the model, which enables to mount sub-models into the tree when loading eCos packages in the configurator (explained shortly in Section 4.4.7).

4.4.4. Feature Groups

CDL supports feature groups with arbitrary cardinalities—not via explicit group keywords, but with its interface mechanisms. Interfaces are a more expressive construct for restricting the cardinality of a set of features beyond OR, XOR, and MUTEX. The value of an interface counts the number of its selected implementations (features with implements keyword). Restricting this value introduces a cardinality constraint (= 1 for XOR, ≥ 1 for OR, and
4.4. The CDL Language

< 1 for mutex). If the configurator detects an XOR constraint, it replaces the checkboxes with radio buttons. Fig. 4.5 shows two scheduler types forming an XOR group in the eCos model.

In contrast to FODA-like languages, CDL does not require that all implementing features are siblings—the feature activating the group constraint needs not be a parent of the constrained features, which even allows to create groups that cross-cut the hierarchy.

Fig. 4.5: XOR group in ConfigTool

4.4.5. Feature Constraints

Configuration constraints are expressed using requires or active_if in CDL. For instance, the dependency Support ZLIB→ZLIB Inflate of Fig. 4.3 is expressed as a requires in c-34. The property takes a condition, say p, and denotes the configuration constraint \( f \rightarrow p \), where \( f \) is the feature in which it is defined. Notably, \( p \) can be an arbitrary expression\(^3\) for requires, possibly accessing multiple features via logical, arithmetic, and string operators.

CDL’s active_if has the same syntactic form and configuration semantics as requires, except that it also enforces a visibility condition. While the visibility of a child in CDL is inherited from its parent in the configuration hierarchy, an explicit visibility condition allows non-parent features to control the visibility too. For example, the visibility of CYGPKG_FS_JFFS2 is controlled by the parent (c-10) and another feature, MTD (c-11).

CDL allows to declare both ranges and enumerations with legal_values. Range restrictions on integer values are specified using legal_values in CDL (c-17). The latter can also be used to specify enumerations of values (numbers, strings, or both), such as in c-42. Enumerations are easier to handle for reasoners (such as SAT or CSP solvers) than ranges, which tend to have larger domain sizes.

Default values are introduced using the keyword default_value (c-16). If no default value is specified, CDL assumes 0 for Boolean and data values, which is dynamically cast to an empty string if needed.

4.4.6. Feature-to-Code Mapping

The feature-to-code mapping together with build specifications is declared directly in the variability model per feature using the compile keyword. This realizes a one-to-many relationship between features and source files. Thus, a feature with code binding is

\(^3\)Called goal expression, see also Section 4.4.8.1.
always an increment in code size. Negative increments would have to be emulated by introducing auxiliary features.

In the source code, feature names do not always correspond directly to preprocessor symbols; instead, flexible control over symbols is supported, such as suppressing symbols (keyword `no_define`), defining additional symbols, or changing their formatting. Line c-18 shows an example of a feature defining a build symbol (`CONFIG_JFFS2_FS_DEBUG`), which actually appears within a preprocessor directive in the code ported from Linux to eCos.

### 4.4.7. Further Concepts

**Textual content.** CDL features can declare a short text using `display` (such as in Line c-7), shown in the configurator’s feature hierarchy, and longer descriptions that explain the feature in detail using `description` (such as in Line c-19), shown in the feature properties pane (Fig. 4.2a).

**Modularization.** CDL was designed for eCos’ packaging system. The functionality of eCos is modularized in eCos packages—archives that contain source code, resource files, and partial variability models that describe the configurability of that package. A feature of kind `package` represents the root of the partial model. When loading/unloading packages in the configurator, the partial model is mounted/unmounted under the synthetic root node if no re-parenting is specified. The CDL models in a package can be further divided into smaller chunks using a simple inclusion mechanism and the `script` keyword, applicable within all grouping features.

### 4.4.8. Formal Semantics

Given our description of CDL’s modeling concepts, we can now present a precise configuration space semantics, which provides the basis for the implementation of our analysis tool infrastructure. As explained earlier (Section 2.3.2.2), we chose a denotational style, as it allows a concise notation and is structurally not far from our actual implementation in CDLTools (see Appendix A.1).

#### 4.4.8.1. Abstract Syntax

**Features, types and constraints.** Let `Id` be a finite set of features, let `Kinds = {package, component, option, interface}` be the set of feature kinds, and let `Flavors = {none, bool, booldata, data}` be the set of types that define a feature’s possible values. CDL allows two types of expressions: `Goal expressions`, used in constraints, and `list expressions`, used for ranges and enumerations in `legal_values`. For goal expressions, let `CExp(Id)` be the set of all possible expressions over `Id`, generated by the following grammar:

\[
e := id \mid \text{const} \mid e \otimes e \mid !e \mid \neg e \mid e \oplus e \mid e \oslash e \mid \text{Func}(e, e, ...) \mid e?e : e
\]
4.4. The CDL Language

with \( \otimes \in \{||, \&\&\}, \oplus \in \{+,-,*,/,%<,>,\leq,\geq,\}\), \( \odot \in \{==
!,!=,<,>,<,>,=\} \), Func \( \in \{\text{get\_data, is\_active, is\_enabled, is\_loaded, is\_substr, is\_xsubstr, version\_cmp}\} \), id \( \in \text{Id} \), and const \( \in \text{Data} \). Data is a set of untyped data, say all character strings.

List expressions represent an enumeration of values or ranges, which can itself be goal expressions. We define \( LExp(Id) \) as the set of all possible list expressions over goal expressions, generated by the following grammar (\( e \in CExp(Id) \), \( l \in LExp(Id) \)):

\[
l ::= (e | e \text{ to } e) \\
\]

\( l \in LExp(Id) \) represents a list expression.

CDL models. Cdl is the set of all possible models in CDL. Each CDL model \( m \in \text{Cdl} \) is a set of nodes\(^4\), so \( \text{Cdl} = \mathcal{P}(\text{Nodes}) \), where:

\[
\text{Nodes} = \text{Id} \times [\text{Id}] \times \text{Flavors} \times \mathcal{P}(CExp(Id)) \times \mathcal{P}(CExp(Id)) \\
\times [CExp(Id)] \times [LExp(Id)] \times \text{Kinds} \times \mathcal{P}(\text{Id})
\]

If \( (n,p,fl,ai,req,cl,lv,knd,imp) \in \text{Nodes} \), then \( n \) is the name, \( p \) is the parent of the node \( (p = \top \) for nodes at the top level), \( ai \) is a set of active_if visibility goal expressions, and \( req \) is a set of requires goal expressions. Further, \( cl \) denotes a calculated goal expression that prescribes the feature’s values and \( lv \) is a legal_values list expression restricting its values. Finally, \( knd \) specifies the node’s domain-specific kind and \( imp \) specifies whether the node implements one or more interfaces. There is no further restriction on both values, that is, an interface can even implement other interfaces. We write \( \text{Id}(m) \) to denote names of nodes in the model \( m \), so \( \text{Id}(m) = \{n \mid (n,\_\_\_\_,\_\_\_\_,\_\_\_\_) \in m\} \).

Well-formedness. CDL introduces some more constraints on the syntax of the model. If \( (n,p,fl,ai,req,cl,lv,knd,imp) \in \text{Nodes} \), it has to fulfill the following invariants:

- \( fl = \text{none} \rightarrow cl = \bot \) (calculated has no effect if flavor is none)
- \( cl \neq \bot \rightarrow lv = \bot \) (calculated and legal_values exclude each other)
- \( fl \in \{\text{bool}\} \rightarrow lv = \bot \) (legal_values applies to nodes with non-fixed data value only, see Eq.4.5)
- \( knd = \text{interface} \rightarrow (fl \neq \text{none} \land cl = \bot) \) (Interfaces must neither have the none flavor nor a calculated property)
- The parent relationship \( p \) should define a tree, with the virtual \( \top \) as the root.

Furthermore, nodes of kind option must not be parents of other nodes.

Mapping to concrete syntax. Our abstract syntax still somewhat differs from the concrete syntax, which is specified as a grammar in the source code of the configurator\(^5\). To keep it simple and concise, our abstract syntax relies on the following preprocessing steps when converting from the concrete syntax:

\(^4\)We use the term node for CDL features in order to distinguish from definitions in Chapter 2.

\(^5\)http://hg-pub.ecoscentric.com/ecos-v3_0-branch/
4. Variability Modeling Languages

1. We introduce a synthetic root element $\top$ as the parent of all top-level packages.
2. In case no flavor is specified for a node, we set the flavor ($fl$) property (according to the CDL documentation [VD]) to booldata for packages, to bool for components and options, and to data for interfaces.
3. The requires, active_if and calculated properties can contain an enumeration of goal expressions separated by whitespace. We convert such enumerations to a disjunction of their goal expressions.

4.4.8.2. Semantic Domain

A configuration is an assignment of triples of values to nodes. The set of all possible configurations in CDL is:

$$\text{Confs} = [\text{Id}] \rightarrow (\{0, 1\} \times \{0, 1\} \times \text{Data})$$  \hfill (4.5)

If $\sigma \in \text{Confs}$ and $x \in \text{Id}$, we write $\sigma(x)_1$ for the first component of the valuation (the enabled state), $\sigma(x)_2$ for the second one (the enabled value), and $\sigma(x)_3$ for the third component of the valuation (the data value). The first component specifies whether the node is actually in the configuration, that is, whether it influences the build of eCos in some sense. The latter two components refer to values the user can give to a node. We predefine the valuation of the $\top$ element as follows: $\sigma(\top)_1 = 1, \sigma(\top)_2 = 1, \sigma(\top)_3 = 1$.

The semantics of a CDL model is given in terms of sets of configurations. Thus, $\mathcal{P}(\text{Confs})$ is our semantic domain, and the semantic function has the signature:

$$[\cdot]_{\text{cdl}} : \text{Cdl} \rightarrow \mathcal{P}(\text{Confs})$$  \hfill (4.6)

4.4.8.3. Semantics

We define the semantic function from abstract syntax to semantic domain using the following helper functions.

**Helper functions.** Let $\text{access} : \text{Id} \times \text{Confs} \rightarrow \text{Data}$ denote a function that returns the value of a feature under a certain configuration while taking its enabled state into account.

$$\text{access}(x, \sigma) = \begin{cases} 0 & \text{iff } \sigma(x)_1 = 0 \\
\sigma(x)_3 & \text{iff } \sigma(x)_1 = 1 \end{cases}$$  \hfill (4.7)

Since arbitrary values can be returned for a feature’s occurrence in an expression and since they can be direct input to boolean operators (e.g. 'feature A requires B &\& C' and C could have flavor data or booldata), we define a cast of arbitrary values to boolean values in the TCL/TK style. More precisely, $\text{bool} : \text{Data} \rightarrow \{0, 1\}$. Please note that $\text{bool}$ is also defined for plain boolean values ($\{0, 1\} \subset \text{Data}$), which are the return type if nodes
are inactive, disabled or bool.

\[
\text{bool}(v) = \begin{cases} 
0 & \text{iff } v = 0 + \forall v = **+ \\
1 & \text{otherwise}
\end{cases}
\]  

(4.8)

For the evaluation of goal expressions, we define the function \( \text{eval} : CExp(\text{Id}) \times \text{Confs} \rightarrow \text{Data} \) recursively as follows, with \( x \in \text{Id}, e_1, e_2, e_3 \in CExp(\text{Id}) \) and \( \text{const} \in \text{Data} \):

\[
\begin{align*}
\text{eval}(x, \sigma) &= \text{access}(x, \sigma) \\
\text{eval}(\text{const}, \sigma) &= \text{const} \\
\text{eval}(!e_1, \sigma) &= \text{non bool}(\text{eval}(e_1, \sigma)) \\
\text{eval}(e_1 \otimes e_2, \sigma) &= \phi_0(\text{bool}(\text{eval}(e_1, \sigma)), \text{bool}(\text{eval}(e_2, \sigma))) \\
& \quad \text{with } \phi_0 = \text{vel, et, seq, eq, aut} \\
\text{eval}(e_1 \oplus e_2, \sigma) &= \phi_1(\text{eval}(e_1, \sigma), \text{eval}(e_2, \sigma)) \\
& \quad \text{with } \phi_1 \text{ TCL's arithmetic for } \oplus = ||, \&\&, \implies, \equiv, \xor \\
\text{eval}(e_1 \oslash e_2, \sigma) &= \phi_2(\text{eval}(e_1, \sigma), \text{eval}(e_2, \sigma)) \\
& \quad \text{with } \phi_2 \text{ TCL's comparison operators for } \oslash = ===, !=, <, >, <\;, >= \\
\text{eval}(e_1 ? e_2 : e_3, \sigma) &= \begin{cases} 
\text{eval}(e_2, \sigma) & \text{iff } \text{bool}(\text{eval}(e_1, \sigma)) \\
\text{eval}(e_3, \sigma) & \text{otherwise}
\end{cases}
\]  

(4.9)

We left out CDL's built-in functions (see \text{Func} in 4.4.8.1) in the definition of \text{eval}, and refer to the CDL online documentation instead.

For the evaluation of values against the \text{legal_values} property, we introduce a satisfaction relation \( \models : \text{Data} \times \text{Confs} \times \text{LExp}(\text{Id}) \rightarrow \{0, 1\} \). For \( d \in \text{Data}; \sigma \in \text{Confs}; e_1, e_2 \in CExp(\text{Id}), l_1, l_2 \in \text{LExp}(\text{Id}) \), we define the relation:

\[
\begin{align*}
d \models_\sigma e_1 & \quad \text{iff } d = \text{eval}(e_1, \sigma) \\
d \models_\sigma e_1 \text{ to } e_2 & \quad \text{iff } \text{bool}(\text{eval}(e_1 <= d \&\& d <= e_2, \sigma)) \\
d \models_\sigma l_1 \sqcup l_2 & \quad \text{iff } (d, \sigma) \models l_1 \lor (d, \sigma) \models l_2
\end{align*}
\]  

(4.10)

\textbf{Semantic function.} The semantics of a model can now be defined as the intersection of denotations of all the nodes, except that we need to ensure that all the unloaded packages (i.e. their nodes) are enforced to be false. Furthermore, we adhere to the semantics of the nodes’ \text{flavor} as well as we take their \text{calculated/legal_values} property and the
interface concept into account.

\[
[m]_{\text{cdl}} = \left( \bigcap_{n \in m} [n]_{\text{node}} \right) \cap \left( \bigcap_{n \in m} [n]_{\text{flavor}} \right) \cap \left( \bigcap_{n \in m} [n]_{\text{calculated}} \right) \cap \left( \bigcap_{n \in m} [n]_{\text{legal_values}} \right)
\cap \left( \bigcap_{n \in m} [n]_{\text{interface}} \right) \cap \{ \sigma \in \text{Confs} | \sigma(x) = 0 \text{ for all } x \in \text{Id} \setminus \text{Id}(m) \} \]

(4.11)

The semantics of a node is a set of all configurations that (1) contain the node’s parent, (2) have the node’s enabled value set, and (3) can satisfy the node’s constraints. Notably, we ignore the unloaded packages here, as we treated them all globally above (here ‘unloaded’ = ‘not mentioned in the model’). For brevity, we also introduce a macro for cross-tree constraints: \( \text{CTC} = \forall e \in ai \cup \text{req.} \, \text{bool} (\text{eval}(e, \sigma)). \)

\[
[(n, p, \_, ai, \text{req}, \_, \_, \_) ]_{\text{node}} = \{ \sigma \in \text{Confs} | \sigma(n)_{1} \leftrightarrow (\sigma(p)_{1} \land \sigma(n)_{2} \land \text{CTC}) \}
\]

(4.12)

Next, we continue with the denotation of nodes according to their flavor property. The values \text{none} and \text{data} are the equivalent to mandatory features in Feature Modeling (FM) with one big difference: In CDL, such nodes can be made optional with cross-tree constraints, whereas in FM, cross-tree constraints of mandatory features also constrain the parent. In the flavor denotation, we set the enabled value for \text{none} and \text{data} features. With regard to Equation (4.12), such mandatory nodes still require the parent node present and cross-tree constraints satisfied.

\[
[(n, p, (\text{none}|\text{data}), ai, \text{req}, \_, \_, \_, \_)]_{\text{flavor}} = \{ \sigma \in \text{Confs} | \sigma(n)_{2} \}
\]

\[
[(\_, \_, (\text{bool} | \text{booldata}), \_, \_, \_, \_, \_, \_)]_{\text{flavor}} = \text{Confs}
\]

(4.13)

The \text{calculated} property forces a node’s \text{data} and enabled value depending on the flavor. Please note that CDL excludes the \text{none} flavor for calculated nodes (see Section 4.4.8.1, well-formedness). For \( cl \neq \bot \), we define:

\[
[(n, \_, \text{bool}, \_, \_, cl, \_, \_, \_, \_)]_{\text{calculated}} = \{ \sigma \in \text{Confs} | \sigma(n)_{2} = \text{bool} (\text{eval}(cl, \sigma)) \}
\]

\[
[(n, \_, \text{booldata}, \_, \_, cl, \_, \_, \_, \_)]_{\text{calculated}} = \{ \sigma \in \text{Confs} | \sigma(n)_{3} = \text{eval}(cl, \sigma) \land \sigma(n)_{2} = \text{bool} (\sigma(n)_{3}) \}
\]

\[
[(n, \_, \text{data}, \_, \_, cl, \_, \_, \_, \_)]_{\text{calculated}} = \{ \sigma \in \text{Confs} | \sigma(n)_{3} = \text{eval}(cl, \sigma) \}
\]

(4.14)

The \text{legal_values} property restricts the data value of a node with a list expression. We define its denotation by using our satisfaction relation from Equation (4.10). Interestingly, the \text{legal_values} property only excludes the flavor \text{bool} by well-formedness rules (Section 4.4.8.1), whereas it does not have any effect on \text{none}-flavored nodes. For
4.4. The CDL Language

$lv \neq \bot$, we define:

\[
\begin{align*}
\llbracket (n,_,(bool|data),_,_,_,_,_,lv,_,_\rrbracket \text{legal\_values} &= \{\sigma \in \text{Confs} \mid \sigma(n)_3 \models \sigma lv\} \\
\llbracket (n,_,none,_,_,_,_,_,lv,_,_\rrbracket \text{legal\_values} &= \text{Confs}
\end{align*}
\]

(4.15)

Finally, we specify the denotation of interfaces, which represent derived features in CDL. Note that the none flavor is excluded for interfaces by well-formedness rules (Section 4.4.8.1).

\[
\begin{align*}
\llbracket (n,_,booldata,_,_,_,_,_,_,interface,_,_\rrbracket \text{interface} &= \{\sigma \in \text{Confs} \mid \sigma(n)_3 \models \text{impls}(n,\sigma) \mid \land \\
&\sigma(n)_2 = \text{bool}(\sigma(n)_3)\} \\
\llbracket (n,_,data,_,_,_,_,_,_,_,_,interface,_,_\rrbracket \text{interface} &= \{\sigma \in \text{Confs} \mid \sigma(n)_3 \models \text{impls}(n,\sigma) \} \\
\llbracket (n,_,bool,_,_,_,_,_,_,_,_,interface,_,_\rrbracket \text{interface} &= \{\sigma \in \text{Confs} \mid \sigma(n)_2 = \text{bool}(\text{impls}(n,\sigma))\}
\end{align*}
\]

(4.16)

where $\text{impls} : \text{Id} \times \text{Confs} \to \mathcal{P}(\text{Nodes})$ is defined as follows:

\[
\text{impls}(n,\sigma) = \{x \in \text{Nodes} \mid n \in x_{\text{impl}} \land \sigma(x)_1 = 1\}
\]

(4.17)

4.4.9. CDL Propositional Semantics

To enable SAT-based analyses, such as satisfiability and dead feature checks, or for building implication graphs, we develop a propositional semantics for CDL. Due to the expressiveness of the language, it only approximates the full configuration space semantics. While it is relatively easy to translate the hierarchy and flavor constraints, this becomes more complicated for cross-tree constraints and CDL’s interface concept. In the following, we define a semantic function that translates from the abstract syntax to propositional logic, more precisely, to a set of Boolean variable assignments. We tailor the original valuations (denotations) down to ones expressible in propositional logic. These provided basis for implementing the transformation in CDLTools (see Appendix A.1).

4.4.9.1. Propositional Semantic Domain

We define a configuration as a set of variable assignments:

\[
\text{Confs}_p = [\text{Id}] \to \{0,1\}
\]

(4.18)

If $\sigma_p \in \text{Confs}_p$ and $x \in \text{Id}$, we write $\sigma_p(x)$ for the valuation of the node under a configuration. We also redefine $\sigma_p(\top) = 1$ for the $\top$ element. Similar to the full semantics, the propositional semantics of a CDL model is given in terms of sets of configurations, so $\mathcal{P}(\text{Confs}_p)$ is our semantic domain. Our semantic function has the signature:

\[
\llbracket \cdot \rrbracket_{\text{pCdl}} : \text{Cdl} \to \mathcal{P}(\text{Confs}_p)
\]

(4.19)
4. Variability Modeling Languages

To understand the relationship between the semantics, in particular how to interpret the value of a boolean variable with regard to the full semantics, we define the following invariants between the semantic domains \( \text{Confs} \) and \( \text{Confs}_p \). These invariants depend on the flavor of a node and are shown in Table 4.2 below.

<table>
<thead>
<tr>
<th>flavor</th>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>( \sigma_p(n) = \sigma(n)_1 )</td>
</tr>
<tr>
<td>none</td>
<td>( \sigma_p(n) = \sigma(n)_1 )</td>
</tr>
<tr>
<td>bool data</td>
<td>( \sigma_p(n) = \sigma(n)_1 \land \sigma(n)_3 \neq 0 )</td>
</tr>
<tr>
<td>data</td>
<td>( \sigma_p(n) = \sigma(n)_1 \land \sigma(n)_3 \neq 0 )</td>
</tr>
</tbody>
</table>

Table 4.2: Invariants between configuration spaces

4.4.9.2. Propositional Semantics

Helper functions. A function like \( \text{access}_p : \text{Id} \times \text{Confs}_p \rightarrow \{0, 1\} \) is not necessary, since a node has only one value left (\( \text{access}_p(\text{id}, \sigma) = \sigma_p(\text{id}) \)). To deal with interfaces, we define the two helper functions \( \text{choose} : \mathcal{P}(\text{Id}) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B} \mathbb{E}(\text{Id}) \) and \( \text{impls}' : \text{Id} \times \text{Cdl} \rightarrow \mathcal{P}(\text{Id}) \). \( \text{choose}(\text{ids}, \text{min}, \text{max}) \) converts a set of \( \text{ids} \in \text{Id} \) into a boolean expression, where at least \( \text{min} \) and at most \( \text{max} \) \( \text{ids} \) can be true.

\[
\text{impls}'(i, m) = \{ n \in m \mid i \in n_{\text{impl}} \} \tag{4.20}
\]

Boolean expressions. We only consider goal expressions, since list expressions only appear in \texttt{legal_values} constraints and cannot easily be approximated\(^6\). Let \( \mathbb{B} \mathbb{E}(\text{Id}) \subseteq \mathbb{C} \text{Exp}(\text{Id}) \) be the subset of boolean expressions over \( \text{Id} \) that are defined by the following grammar, with \( \otimes = \{||, \&\& , \text{implies}, \text{eqv} \} \) and \( \text{const} \in \{0, 1\} \):

\[
e ::= \text{id} \mid \text{const} \mid e \otimes e \mid !\text{id} \tag{4.21}\]

Boolean expression evaluation. The evaluation of \( \mathbb{B} \mathbb{E}(\text{Id}) \) follows ordinary propositional semantics; thus, we define \( \text{eval}_p : \mathbb{B} \mathbb{E}(\text{Id}) \times \text{Confs}_p \rightarrow \{0, 1\} \) as follows, with \( x \in \text{Id}, e_1, e_2 \in \mathbb{B} \mathbb{E}(\text{Id}) \) and \( \text{const} \in \{0, 1\} \):

\[
\begin{align*}
\text{eval}_p(x, \sigma_p) &= \sigma_p(x) \\
\text{eval}_p(\text{const}, \sigma_p) &= \text{const} \\
\text{eval}_p(\text{!}e_1, \sigma_p) &= \text{non eval}_p(e_1, \sigma_p) \\
\text{eval}_p(e_1 \otimes e_2, \sigma_p) &= \phi_0(\text{eval}_p(e_1, \sigma_p), \text{eval}_p(e_2, \sigma_p)) \\
&\quad \text{with } \phi_0 = \text{vel, et, seq, eq} \text{ for } \otimes = ||, \&\&, \text{implies, eqv}
\end{align*} \tag{4.22}
\]

\(^6\)It is possible to approximate special cases, such as \texttt{legal_values} 0 and so on. However, we hardly have seen such enumerations in the real models.
4.4. The CDL Language

Expression rewriting rules. We define a partial function \( \text{rewrite}: C\text{Exp}(\text{id}) \times \text{Cdl} \rightarrow \text{BE}(\text{id}) \) that translates CDL goal expressions to Boolean expressions. For \( x \in \text{id} \), \( m \in \text{Cdl} \), and \( e_1, e_2, e_3 \in C\text{Exp}(\text{id}) \):

\[
\text{rewrite}(x, m) = \begin{cases} 
  x & \text{iff } x \in \text{id}(m) \\
  0 & \text{otherwise}
\end{cases}
\]

(4.23)

\[
\text{rewrite}(!x, m) = \neg \text{rewrite}(x, m)
\]

\[
\text{rewrite}(\text{const}, m) = \text{bool}(\text{const})
\]

\[
\text{rewrite}(x = \text{const}, m) = \begin{cases} 
  \text{rewrite}(x, m) & \text{iff } \text{bool}(\text{const}) \neq 0 \\
  \text{rewrite}(\neg x, m) & \text{otherwise}
\end{cases}
\]

\[
\text{rewrite}(x > \text{const}, m) = \begin{cases} 
  \text{rewrite}(x, m) & \text{iff } \text{const} \in \text{INT} \land \text{const} \geq 0 \\
  1 & \text{otherwise (drop it)}
\end{cases}
\]

\[
\text{rewrite}(\text{is_substr}(x, \text{const}), m) = \text{rewrite}(x, m)
\]

\[
\text{rewrite}(e_1 \otimes e_2, m) = \text{rewrite}(e_1, m) \otimes \text{rewrite}(e_2, m)
\]

\[
\text{rewrite}(e_1 ? e_2 : e_3, m) = (\text{rewrite}(e_1, m) \rightarrow \text{rewrite}(e_2, m)) \\
\land (\neg \text{rewrite}(e_1, m) \rightarrow \text{rewrite}(e_3, m))
\]

For \( x \in \text{id}(m) \) and if \( x \) denotes an interface, we continue the definition of \( \text{rewrite} \) as follows:

\[
\text{rewrite}(x = 0, m) = \neg x \land \bigwedge_{i \in \text{impls}'(x, m)} \neg i
\]

\[
\text{rewrite}(x > 0, m) = x \land \bigvee_{i \in \text{impls}'(x, m)} i
\]

(4.24)

\[
\text{rewrite}(x = 1, m) = x \land \text{XOR} \bigoplus_{i \in \text{impls}'(x, m)} i
\]

\[
\text{rewrite}(x \geq \text{const}, m) = x \land \text{choose}(\text{impls}'(x, m), \text{const}, | \text{impls}'(x, m) |)
\]

\[
\text{rewrite}(x > \text{const}, m) = x \land \text{choose}(\text{impls}'(x, m), \text{const} + 1, | \text{impls}'(x, m) |)
\]

Semantic function The propositional semantics of a model \( m \in \text{Cdl} \) is the intersection of the propositional denotations, similar to the full semantics. However, we omit legal_values as explained previously, and provide the current model \( m \in \text{Cdl} \) as a parameter to the valuation functions (needed for \( \text{rewrite} \) above).

65
4. Variability Modeling Languages

\[
[m]_{\text{pCDL}} = \left( \bigcap_{n \in m} \left[ n, m \right]_{\text{pNode}} \right) \cap \left( \bigcap_{n \in m} \left[ n, m \right]_{\text{pFlavor}} \right) \cap \left( \bigcap_{n \in m} \left[ n, m \right]_{\text{pCalculated}} \right) \cap \left( \bigcap_{n \in m} \left[ n, m \right]_{\text{pInterface}} \right) \cap \{ \sigma_p \in \text{Confs}_p \mid \sigma_p(x) = 0 \text{ for all } x \in \text{ld} \setminus \text{ld}(m) \} \tag{4.25}
\]

The semantics of a node is the set of all configurations that can satisfy its constraints. Similar to the full semantics, we introduce the macro

\[
C_{\text{TC}} = \forall e \in a_i \cup \text{req} (e \in \text{dom}(\text{rewrite}) \rightarrow \text{eval}(\text{rewrite}(e, m), \sigma_p)).
\]

The denotations can now be defined as follows.

\[
[(n, p, \_ , ai, \text{req}, \_ , \_ , \_ , \_ ) , m]_{\text{pNode}} = \{ \sigma_p \in \text{Confs}_p \mid \sigma_p(n) \rightarrow \sigma_p(p) \land C_{\text{TC}} \} \tag{4.26}
\]

We continue with the denotation of nodes according to their flavor property:

\[
[(n, p, (\text{none}|\text{data}), \_ , \_ , \_ , \_ , \_ , \_ ) , m]_{\text{pFlavor}} = \{ \sigma_p \in \text{Confs}_p \mid \sigma_p(p) \land C_{\text{TC}} \rightarrow \sigma_p(n) \}
\]

\[
[(n, p, (\text{bool}|\text{booldata}), \_ , \_ , \_ , \_ , \_ , \_ ) , m]_{\text{pFlavor}} = \text{Confs}_p \tag{4.27}
\]

Similarly, we define the denotation of the calculated property. For \( cl \neq \bot \):

\[
[(n, p, \_ , \_ , \_ , \_ , cl, \_ , \_ , \_ ) , m]_{\text{pCalculated}} = \{ \sigma_p \in \text{Confs} \mid \sigma_p(p) \land C_{\text{TC}} \rightarrow \sigma_p(n) = \text{eval}(\text{rewrite}(cl, m), \sigma_p) \} \tag{4.28}
\]

Finally, the propositional denotation of interfaces can be declared as follows:

\[
[(n, \_ , \_ , \_ , \_ , \_ , \_ , \_ , \_ , \text{interface}, \_ ) , m]_{\text{pInterface}} = \{ \sigma_p \in \text{Confs}_p \mid \sigma_p(p) \land C_{\text{TC}} \rightarrow \sigma_p(n) = \text{eval}(\bigvee_{i \in \text{impls}(x, m)} i, \sigma_p) \} \tag{4.29}
\]

4.5. The Kconfig Language

We now describe the characteristics of the Kconfig language we identified in our study, using the same conceptual framework as for CDL and also referring to line numbers in Fig. 4.3 for illustration.

4.5.1. Feature Kinds

In Kconfig, feature kinds reflect their appearance in the configurator UI. Menus are pure grouping features. Menuconfigs are grouping features that also represent configuration...
options—looking like menus that can be enabled and disabled by clicking. Choices are like menus or menuconfigs except that they also impose cardinality constraints on their children.Configs are individual features representing configuration options; however, some are rendered as grouping features with children in the configurator as we will see later in Section 4.5.3.

Kconfig has no syntax to indicate the role of a feature that represents a configuration option. Every config or menuconfig can be an implementation feature, that is, their names can always be referenced in build scripts or code. User and derived features are distinguished by their prompt clause—a label shown to the user and declared right after the type of the feature, such as in Lines k-7 or k-14. Derived features have no prompt, their value is always restricted by constraints and cannot be changed directly by the user. Finally, capabilities are modeled by constraints that other features declare on them; more precisely, if a features provides a capability, it declares a constraint that automatically selects the capability feature.

In Fig. 4.3, the menuconfig MISC_FILESYSTEMS (Line k-1) corresponds to the root node in the feature model above the language excerpts. It contains a choice (k-38) corresponding to the parent feature of the xor-group, Default Compression, and eight configs corresponding to the remaining features of the feature model—all enclosed by a pair of matching if (k-4) and endif (k-49) keywords. Among all individual features, JFFS2_FS_WRITEBUFFER (k-22) is a derived feature that is not visible in the configurator, because it has no prompt clause (k-23). Its value is calculated as equal to the value of the HAS_IOMEM capability (referenced in Line k-25, but defined elsewhere). All other individual features are both user and implementation features.

4.5.2. Feature Representation

In Kconfig, a configuration assigns a single value to each feature. For the set of all possible features (Id), and the set of all possible feature values (Val) in Kconfig, a particular configuration σ maps feature names (f ∈ Id) to values:

\[
\sigma : \text{Id} \mapsto \text{Val}
\]

and if \(\sigma(f) = v\), then \(v \in \text{type-of}(f)\) \hspace{1cm} (4.30)

Switch features comprise the types bool and tristate. They appear with checkboxes in the configurator. Bool allows two values, y and n, internally represented by 2 and 0. The latter, 0, denotes feature absence, while 2 means that the feature’s implementation should be compiled statically into the kernel. Tristate resembles bool, except for the additional value m, internally represented by 1. It indicates that the implementation should be compiled as a dynamically loadable module—Linux’ mechanism to load drivers at runtime. For example, for the tristate feature JFFS2_FS (k-6), the user can choose to deselect it, to create a dynamically linked module, or to link it statically. Of course, depending on the mapping to code, not every feature represents a separate module, but often belongs to one. For example, JFFS2_FS’s descendant JFFS2_ZLIB (k-32) of type bool can only be (de-)activated; but when selected, its implementation is always linked.
4. Variability Modeling Languages

statically into the JFFS2 compilation unit, without creating a separate module.

Data features comprise integer and string types, for which the configurator offers a
text box to enter values. The language supports two integer types: \texttt{int} (decimal) and
\texttt{hex} (hexadecimal). Both types also allow an empty value, which is used to encode
the absence of an integer feature. The type \texttt{string} is ambiguous in this respect: a string
feature with the empty value can be seen as a present feature with that value or an
absent feature; the two cases are indistinguishable.

\texttt{Kconfig}'s menus have no type, which corresponds to features of type \texttt{none} in CDL,
and mandatory with no attribute in feature models. In Equation (4.30), we assume that
the type \texttt{none} contains a single uninterpreted element representing no value.

4.5.3. Feature Hierarchy

The syntactic model hierarchy is given by the nesting of configs under menus or choices
in \texttt{Kconfig}. This nesting is reflected in the configurator hierarchy. However, configs can
also appear as children of other configs in the configurator, even though they cannot be
nested syntactically in the model. The configurator has an algorithm to additionally nest
syntactic sibling configs based on their declared dependencies. For example, a group of
consecutive configs declaring dependency on the same parent (such as in lines k-13–25)
is placed under this parent (\texttt{JFFS2_FS}). Finally, like in CDL, the \texttt{Kconfig} configurator
introduces a synthetic root feature that is not explicitly specified in the model.

Recall that in feature modeling, all children imply their parent feature ($\sigma(c) \rightarrow \sigma(p)$).
In contrast, the configurator hierarchy in \texttt{Kconfig} only enforces visibility between a child
and its parent—a feature is visible when its parent is visible. An interesting phenomenon
is that, in some cases, a feature can still be selected (automatically via constraints) when
the parent is not selected, possibly even excluding the parent. Such a configuration is
still valid in \texttt{Kconfig}, unlike in any other feature modeling language known to us. This
mechanism of \texttt{Kconfig} enables features to conditionally play the roles of both derived and
implementation features at the same time. Fig.4.6 shows the feature “Encoders/decoders
and other helper chips” in \texttt{xconfig}, which, together with its descendants, excludes its
parent “Autoselect pertinent encoders/decoders and other helper chips”. 
4.5. Feature Groups

In Kconfig, the choice keyword groups a set of features and imposes a group constraint on them—either XOR or MUTEX. A choice is either bool or tristate with a mandatory or optional modifier flag; Table 4.1 summarizes the combinations of types and flags. If not specified otherwise, a choice is mandatory and bool, which semantically represents an XOR group, such as the choice in line k-38⁷. If the choice is optional and bool, it realizes a MUTEX group. Tristate choices behave differently and cannot be interpreted as feature modeling groups. Mandatory tristate choices either admit exactly one feature set to y (all others to n), or any number of features set to m. This behavior is useful if various drivers exist for one hardware device where only one can be compiled into the kernel, but all can be built as modules. This realizes an XOR group at runtime, as only one driver can be loaded per device. Interestingly, this use case is—to the best of our knowledge—only supported in the FeatuRSEB method, with the interpretation of XOR and OR groups (see Section 2.3.3). Finally, optional tristate choices—surprisingly—do not impose any cardinality constraint.

4.5.5. Feature Constraints

Configuration constraints are expressed using the imperative select statement. In contrast to CDL, the Kconfig configurator uses it to immediately do choice propagation on selection of a feature, regardless of constraints restricting the target feature; see Section 4.6 for details. For instance, the dependency Support ZLIB → ZLIB Inflate of our sample feature model in Fig.4.3 is expressed as a select in line k-35. Select only allows to specify a feature identifier, instead of arbitrary expressions like CDL’s requires. Finally, Kconfig allows to restrict the domains of its data values. It supports ranges for numbers (int and hex) with the range statement (k-17).

In Kconfig, the visibility of a feature is controlled by a prompt condition. A prompt is a string that follows a type declaration (k-7). It is shown to the user when the feature is visible (the condition is satisfied). The condition is specified after the prompt: here MTD in line k-7. Note that the select statement in line k-8 is also conditioned on the same condition as the prompt. This pattern of guarding other constraints by the prompt condition is frequent in Kconfig; thus, the language provides a syntactic sugar for it. The depends on keyword adds a condition to the prompt and all other constraints of a feature. For example, the prompt, default, and range specifications of JFFS2_FS_DEBUG are only active if JFFS2_FS is selected, as specified in line k-15. Constraint expressions in Kconfig can use logical operators and equality tests over bool, tristate, integers and strings.

Default values are introduced using default keyword (k-16). If none is specified, Kconfig assumes n (0) for bool and tristate, and the empty string for string, int, and hex.

⁷Note that eCos developers decided to model this group differently (c-38): with a data-flavored option holding one of three string values encoding the three compression modes.
An observed phenomenon is that visibility conditions, defaults, and configuration constraints interact in intricate ways. If the visibility condition of a feature is false, its default value specification becomes a configuration constraint because the feature cannot be accessed by the user to modify the default value. Such invisible features with calculated values are *derived features*, as defined previously in Section 4.5.1. \texttt{JFFS\_FS\_WRITEBUFFER} in line k-22 is derived, since it has no prompt declared, thus, its visibility condition is false and its default determines the value. Notice that this feature was not shown in the feature model of Fig. 4.3, as the FODA notation does not include syntax for invisible, derived features.

An example of a *conditionally derived* feature is \texttt{JFFS2\_ZLIB}, with a stronger visibility condition (\texttt{prompt} and \texttt{depends on}) than its default condition (just \texttt{depends on}). Thus, when the feature is not visible, its value is derived using its default. This happens even if its parent \texttt{JFFS2\_COMPRESS} is not selected. Consequently, \texttt{JFFS2\_ZLIB} does not establish a child-parent implication, as in feature modeling notations.

A unique feature of Kconfig is its first-class support for a three-valued logic. Its main operators are defined as follows:

\[
\begin{align*}
\text{eval}(\neg e) &= 2 - \text{eval}(e) \\
\text{eval}(e_1 \&\& e_2) &= \min(\text{eval}(e_1), \text{eval}(e_2)) \\
\text{eval}(e_1 \mid\mid e_2) &= \max(\text{eval}(e_1), \text{eval}(e_2))
\end{align*}
\]

The semantics of expressions follows the logic of Kleene [Kle38], where \texttt{m} corresponds to the unknown state. The equality and inequality test is only defined between features and constants (i.e. \texttt{tristate}, \texttt{int}, \texttt{hex} and \texttt{string}). It evaluates to \texttt{y} (2) if the values match, and to \texttt{n} (0) otherwise.

### 4.5.6. Mapping to code

All configs and menuconfigs correspond directly to symbols used within the build system and in the source code as preprocessor directives of the same name (see Fig. 4.4). Kconfig lacks the ability to decouple features and implementation-facing symbols, as seen in CDL.

The mapping between features and source files resides in imperative build logic inside the KBuild [Cc] system—a mechanism built on top of Makefiles to control the inclusion of files based on a configuration. The logic is spread over more than 600 KBuild Makefiles in the entire source tree. Although the mapping is not declarative, KBuild is more flexible than CDL, allowing a many-to-many mapping where a feature is not necessarily an increment in code size, since enabled features can lead to excluding source files.

#### 4.5.6.1. The KBuild System

In our study, we analyzed the design of KBuild and identified the following patterns of controlling file inclusion. In the following, we describe these patterns and give their semantics in terms of a presence condition (cf. Section 2.1.2).
4.5. The Kconfig Language

The top-level KBuild Makefile declares lists collecting files for compilation in different modes: to be linked statically (`obj-y`), to be linked dynamically as modules (`obj-m`), or to be included in a library (`lib-y`). It then descends into the source tree and conditionally invokes other makefiles, which add files to the lists. In the simplest case, files are added unconditionally. In the example below, two files are added to the `obj-y` list together with the directory `partitions/`, which means that the Makefile located there should be included in further traversal. Note that names of object files are used, not source files, which are linked to object files by implicit compilation rules of Make:

\[
\text{obj-y += open.o jffs2.o partitions/}
\]

\[
\text{jffs2-y := compr.o dir.o file.o ioctl.o nodelist.o malloc.o}
\]

The second line creates a list indicating files that should be used to build `jffs2.o`. A compilation rule specified elsewhere declares a dependency between object files and lists of this kind. In the example, the presence conditions for all files are simply `true`. However, the complete presence condition of a file may be different, due to inheritance of conditions from enclosing makefiles.

Files are added conditionally either by using control-flow statements of Make or by constructing the name of a list conditionally. We illustrate the latter first:

\[
\text{obj-$(JFFS2_FS) += jffs2.o}
\]

\[
\text{jffs2-$(JFFS2_FS_WRITEBUFFER) += wbuf.o}
\]

Here, \( $(JFFS2_FS) \) denotes a value of feature `JFFS2_FS` in the configuration (the string `y`, `m` or `n` in this case), which is concatenated to create a list name. Note that `JFFS2_FS_WRITEBUFFER` can only be `y` or `n`. This example results in the following presence condition for `wbuf.c`:

\[
\text{JFFS2_FS_WRITEBUFFER} = y \land (\text{JFFS2_FS} = y \lor \text{JFFS2_FS} = m)
\]

Below, we show a conditional Make command that induces the following presence condition for `xfs_qm_stats.c`:

\[
\text{(XFS_FS} = y \lor \text{XFS_FS} = m) \land \text{PROC_FS} = y \land \text{XFS_QUOTA} = y.
\]

\[
\text{obj-$(XFS_FS) += xfs.o}
\]

\[
\text{ifeq ($(XFS_QUOTA),y)}
\]

\[
\text{xfs-$(PROC_FS) += quota/xfs_qm_stats.o}
\]

\[
\text{endif}
\]

Some dependencies are expressed in complex manners. The next example includes `dccp_ipv6` as a module if either the feature `IPv6` or `DCCP` equals `m`:

\[
\text{obj-$(subst y,$(CONFIG_IP_DCCP),$(CONFIG_IPV6)) += dccp_ipv6.o ,}
\]

where `subst` is a substring substitution function. We obtain the following condition for this example: 

\[
\text{IPV6} = m \lor (\text{IPV6} = y \land (\text{IP_DCCP} = m \lor \text{IP_DCCP} = y))
\]
4. Variability Modeling Languages

4.5.6.2. Static Analysis

To analyze the feature-to-code mapping in Linux, we implemented a static analysis technique that transforms the imperative build logic of KBuild into individual file presence conditions. The tool KBuildMiner (see Appendix A.2, a technical report [BSCW10a], and our poster [BSCW10b, Ber10]) are freely available. The tools builds upon a fuzzy parser recognizing all of the documented variability specification patterns, but also some undocumented ones we discovered.

The traversal of the build tree starts with the main hardware architecture Makefile and descends into the referenced ones. The resulting AST contains nodes representing Makefiles, conditional statements, lists of compound objects, variable references, and source files as leaves, some annotated with local presence conditions. Computing the full presence condition of a source file involves finding all paths from the file to the root, while taking variable resolution rules into account and then conjoining all expressions in a path and making a disjunction over the path conditions. However, for complex cases like Equation (4.35), we had to create the presence condition manually.

4.5.7. Further Concepts

Textual content. Kconfig features can have a short text shown in the feature tree in the configurator using prompt (such as in Line k-7), and a longer description using help (such as in Line k-19).

Modularization. The Kconfig models in the Linux kernel consist of smaller Kconfig files that are hierarchically organized in the codebase, with individual root files for each of the 22 hardware architectures. The root file includes others with a source statement, which in turn include further descendants. Although a hierarchy among the Kconfig files exist, the inclusion is purely a macro expansion mechanism and the content is linearized without introducing hierarchy—except if the source statement appears in one of the structural elements (IF/ENDIF or grouping features). On a final note, it is possible that separately declared features (also across files) have the same name and, thus, define the same preprocessor symbol. Since all these features share the same state in the configurator, doing so can lead to intricate interactions of dependencies.

4.5.8. Formal Semantics

We refrain from providing the full formal semantics of Kconfig here, but refer to a technical note [SB10] written by Steven She and guided by us. In a nutshell, the Kconfig semantics differ from CDL in the following aspects:

4.5.8.1. Abstract Syntax

Allowed feature values are $\text{Val} = \text{Tri} \cup \text{String} \cup \text{Hex} \cup \text{Int}$, where Tri, String, Hex, and Int are disjoint. $\text{Tri} = 0_t, 1_t, 2_t$ is an ordered set with $0_t < 1_t < 2_t$ that represents assignable values to tristate and bool (only $0_t$ and $2_t$) features.
Expressions are less expressive than in CDL, only few operators are supported. These expressions can only be used in visibility conditions, and as conditions of defaults and range restrictions, but not with the `select` statement. For illustration, we define $KExp(id)$ with the following grammar, where $e \in KExp(id)$, $iv \in id \cup \text{Val}$, $\otimes \in \{\text{or, and}\}$, $\ominus \in \{=, \neq\}$:

$$e ::= e \otimes e \mid !e \mid iv \ominus iv \mid iv$$  \hspace{1cm} (4.36)

The set of all possible Kconfig models can be defined as a tuple of sets of configs and choices: $Kconfig = \mathcal{P}(\text{Configs}) \times \mathcal{P}(\text{Choices})$. For brevity, we unify menuconfigs as configs, and omit menus, which do not define feature symbols. But since menus can have constraints, we impose these on its nested configs in a pre-processing step [SB10], to keep the semantics concise. The features of Kconfig models—configs—are tuples with the following components:

$$\text{Configs} = id \times \text{Type} \times KExp(id) \times \mathcal{P}(\text{Default}) \times KExp(id) \times \mathcal{P}(\text{Range}),$$  \hspace{1cm} (4.37)

where the first component represents the feature symbol, the second its value type ($\text{Type} = \{\text{boolean, tristate, int, hex, string}\}$), the third its visibility condition, the forth its (conditional) default values ($\text{Default} = KExp(id) \times KExp(id)$), the sixth its reverse dependencies (an expression representing other features that can `select` this feature), and the seventh its (conditional) range restrictions of int and hex features ($\text{Range} = (\text{Int} \cup \text{Hex} \cup id) \times (\text{Int} \cup \text{Hex} \cup id) \times KExp(id)$). Notably, we do not have to define a parent pointer, since hierarchy among configs is not supported in the Kconfig syntax, thus, not reflected in our abstract syntax, see configurator hierarchy in Section 4.5.3. The definition of a choice is similar to a config, except it only has the type declaration (bool or tristate), the mandatory or optional flag, its visibility condition, and the set of its children [SB10].

4.5.8.2. Semantic Domain and Semantic Function

As explained in Section 4.5.2, a Kconfig configuration only assigns one value per feature, which simplifies the semantics. All possible configurations can be defined as $\text{Confs} = id \rightarrow \text{Val}$. Thus, the semantic function has the signature $\llbracket \cdot \rrbracket_{kconfig} : Kconfig \rightarrow \mathcal{P}(\text{Confs})$ and the definition (with $m \in Kconfig$, and $m_1$ referring to the first component of $m$, i.e. all configs):

$$\llbracket m \rrbracket_{kconfig} = \left( \bigcap_{n \in m_1} \llbracket n \rrbracket_{\text{type}} \right) \cap \left( \bigcap_{n \in m_1} \llbracket n \rrbracket_{\text{bounds}} \right) \cap \left( \bigcap_{n \in m_1} \llbracket n \rrbracket_{\text{default}} \right) \cap \left( \bigcap_{n \in m_1} \llbracket n \rrbracket_{\text{range}} \right) \cap \left( \bigcap_{n \in m_\text{choice}} \llbracket n \rrbracket_{\text{choice}} \right) \cap \llbracket m \rrbracket_{\text{module}} \cap \llbracket m \rrbracket_{\text{undeclared}}$$  \hspace{1cm} (4.38)

\[\text{Above, we unified menuconfigs to configs; however, menuconfigs introduce syntactic hierarchy. In the formal semantics in [SB10], this issue is solved with a preprocessing step that imposes menuconfigs' constraints onto the children.}\]
The technical note [SB10] provides definitions of all the valuation functions used, and their helper functions. The following issues complicate the semantics of Kconfig in contrast to CDL:

- Kconfig uses three-state logic, which impacts the definition of expression evaluation, but also the interactions between visibility conditions and reverse dependencies. The latter behavior is modeled in the valuation function \([\cdot]_{\text{bounds}}\): For tristate features, reverse dependencies set a lower bound, and the visibility condition an upper bound for the value. This behavior has practical relevance. For example, given a model where tristate feature A requires tristate feature B. If feature A is assigned the m value, feature B at least has to be compiled as a module, but could also be built into the kernel. Now, if A is set to y, feature B also must be in the kernel, since A’s implementation could not rely on B’s implementation being loaded.
- Although Kconfig features have a type, data types are cast to tristate or bool if needed. For example, in constraints, a string feature with an empty value is not distinguishable from the case when it is not declared in the model.
- Nested features inherit constraints in non-trivial ways. To keep the abstract syntax concise, we rely on transformations from the concrete syntax that propagate constraints down to the features that inherit them. However, these have to be discounted later in the quantitative model analysis that measures declared dependency structures.

4.5.8.3. Propositional Semantics

Developing a propositional abstraction of Kconfig is—similarly to CDL—a none-trivial problem. Steven She developed two different abstractions, one that introduces two variables for tristate features, and one that uses only one variable. The latter is simpler to use with SAT-based analysis, but harder to interpret. These abstractions were implemented by him as model transformations in the LVAT tool suite (see Appendix A.3).

4.6. The Configurators

Kconfig and CDL are equipped with GUI-based configurators shown in Fig. 4.2. We studied the supported configuration processes and the implementations of the configurators to analyze choice propagation, reasoning, and conflict resolution support.

4.6.1. Process

Both configurators support a configuration process known as reconfiguration: The tool is initialized with a configuration loaded from a file, or based on default values, which is modified stepwise by the user to reach a desired state. After each step, the configurator checks constraints and reports potential conflicts.

The reconfiguration paradigm is different from valid-domain computation [HSJ+04, HA07], used by some feature-based configurators, such as FMP [AC04], where the user starts with a set of undecided features and each configuration step assigning a value
4.6. The Configurators

to a feature triggers the computation of the values that are still allowed for the other features (so-called valid domains). Valid-domain computation helps the user reach a valid configuration, that is, one that does not violate any configuration constraints, while saving manual work by inferring the values of the dependent features.

Some configurators provide fixed response times with precompilation of the configuration space, such as Mendonça et al.’s approach [MWCC08] with binary decision diagrams [Bry86, MT98]. Since scalability is an issue for precompilation, Janota [Jan10] presents a scalable lazy approach for propositional feature models. Thus, due to limited expressiveness, all these academic approaches could not be used on our rich languages to provide configuration guidance.

4.6.2. Reasoning and Limitations

The Kconfig configurator offers little support for propagating user configuration choices. If the dependencies of a given feature are not satisfied, the tool prohibits selecting it. The user has to find out which other features need to be reconfigured to enable the selection. A rudimentary propagation support is offered by the select construct; it enforces a selection of a single feature when the feature hosting the statement is selected. The selection is made without respecting any constraints. This imperative behavior can lead to illegal configurations and requires Kconfig developers to explicitly specify any transitive dependencies to maintain consistency. For example, LATENCY_TOP contains selects for both KALLSYM and KALLSYM_ALL. KALLSYM_ALL depends on KALLSYM, thus, the sole selection of KALLSYM_ALL would be sufficient if the configurator used a propagating reasoner. In fact, the official documentation and the Linux kernel commit log contain multiple warnings and complaints about the error-proneness of using this construct [LSB+10]. Still, the Linux model is full of select statements, as this is the only way to obtain (limited) propagation in the configurator.

The CDL configurator is far more intelligent than its Kconfig counterpart. When the user modifies a configuration, the tool detects all constraint violations and offers the user support to resolve them via an inference engine.

This engine works as follows. Every change to the model is wrapped in a transaction, and the configurator checks for any constraint violation. If one occurs, the inference engine tries to resolve the conflict by a heuristics-based recursive search algorithm. It builds a tree of transactions, starting a transaction for each new sub-conflict that arises when testing conflict resolutions. The engine estimates the benefit of particular (sub-) conflict resolutions, by using the number of required changes and source of the values being changed, for example, user, default or inference. If a resolution is beneficial, it gets committed to the parent transaction. If one overall solution is found for the top-level conflict, the tool lists necessary changes and requests confirmation. Otherwise, the conflict requires manual resolution.

We investigated the inference engine’s source code with respect to correctness and completeness. The resolution is correct, since the proposed resolutions are verified against the model constraints. The resolution is incomplete as:
4. Variability Modeling Languages

- The inference rules are incomplete. For example, the engine has rules for handling cardinality constraints on interfaces of 0 or 1, but not for arbitrary bounds.
- The recursion depth is limited to three levels; thus, reasoning on transitive requires dependencies is incomplete.
- The engine uses a greedy search, evaluating resolutions to sub-conflicts in separation and pruning all but the optimal one. This may prune all successful branches.

Although the inference engine is less powerful than general CSP solvers, it performs very well on the actual eCos model. The support for mutex and xor groups is particularly effective and the resolution of requires dependencies is far more maintainable than the select statement in Kconfig.

The main limitation of the CDL configurator is that if several resolutions exist, it finds at most one and possibly not the desired one. The following comment\(^9\) on the mailing list indicates that developers struggle with this problem:

```plaintext
[...] if CYGPKG_MYPKG_OP1 is active, make sure that the list of tests for that package is a substring of CYGDAT_MYPKG_ACTIVE_TESTS. This works 50% of the time. Problem is the other 50% of the time, rather than fiddling with the substrings, it enables / disables my subpackage!
```

Our findings underscore the importance of building configurators based on strong reasoners. Tools employing complete reasoners do exist for package configuration involving simple use dependencies and version ranges. For example, p2 in Eclipse is using a SAT solver [LBR09].

4.7. Conclusions

CDL and Kconfig are two successful real-world languages used in industrial contexts. They were developed independently from each other, but also independently from feature modeling research. Since they share many similar concepts, they confirm the importance of the modeling constructs discussed in literature. However, both appear to follow different design philosophies. While CDL turns out to be a well-engineered and very expressive language, Kconfig appears more like a scripting solution with hacks and intricate semantics of some language constructs.

**Modeling concepts.** With our qualitative analysis, we provide empirical evidence that feature modeling concepts from FODA are used in practice. However, some concepts have characteristics not discussed in literature yet, such as the separation of syntactic and configurator hierarchy, which enables decoupling of the developer and user view. Furthermore, the languages benefit from being domain-specific—domain vocabulary increases understandability.

We also show that more advanced concepts, such as visibility conditions, derived features, modularization or binding modes are needed. These concepts aim at scaling

variability modeling, in particular the creation, maintenance, and configuration of models. Further, our analysis unleashed intricate semantic interactions among the advanced concepts, deepening our understanding of such languages.

Specifically, derived defaults, visibility conditions, and derived features, which are marked as rare in Table 4.1, are generally not supported by state-of-the-art feature modeling languages such as TVL [BCFH10] and pure::variants. Derived defaults were proposed by researchers [CE00], but not provided by feature modeling languages.

**Tools.** We observed limitations in the configurators. The Kconfig configurator lacks reasoning procedures to support choice propagation. To mitigate this, Kconfig includes an imperative construct (select) for specifying limited choice propagation directly in the model, but which turns out to be very error-prone. eCos boasts a far more intelligent configurator based on a home-grown inference engine. Unfortunately, the reasoning procedures of the engine are incomplete and may propose undesirable configuration choices. Interestingly, both configurators follow a reconfiguration paradigm: any configuration task starts with an initial, possibly default, configuration and continues by modifying this initial configuration.

**Mapping.** We have demonstrated the feasibility of extracting feature-to-code mappings from imperative build logic using static analysis techniques. Although the extractor needs to be custom-built and works with heuristics, explicit presence conditions enable a wide range of useful tools, amortizing the building effort. In fact, we applied the extracted presence conditions to evaluate the variability-aware parsing approach in [KGR+11].

**Semantics.** Finally, we widened our understanding of variability model semantics. We provide a configuration space semantics in a denotational style, which is an abstraction of the full semantics.

On top of our semantics, and in addition to the propositional configuration space semantics, other abstractions have been developed, such as the configurator semantics in [Xio11]. It models reasoning about configurations from the perspective of the configuration process by the user, to provide intelligent guidance. Thus, it can ignore—among others—derived (calculated) features, which not constrain user-changeable features. In contrast, developing a C code analyzer that is aware of the variability model, would require another abstraction, since it may query the valid values of a particular calculated data feature. This would amount to asserting a constraint on the calculated feature, and in turn constrain user-changeable features. Thus, such an analyzer would then need to consider all operations from the calculated feature, which is not supported by the abstracted configurator semantics of [Xio11].
Variability Models

This chapter shares content with the technical report “Variability Modeling in the Systems Software Domain” [BSL+12] (under review for IEEE TSE).

After our analysis of the two languages CDL and Kconfig, we now turn to 13 actual models and study the usage of variability modeling concepts, both qualitatively and quantitatively. The qualitative analysis aims at unleashing design criteria that modelers used when creating the models. The quantitative analysis determines which language concepts are used in practice and how frequently.

In the remainder of this chapter, we first describe our methodology, then introduce the host projects of our subject models, and finally report our results. These comprise a characterization of the models’ contents, identified common organizational structures, and the constraints and dependency structures among features in the models. The conclusions at the end of this chapter contain a critical discussion of our findings.

5.1. Methodology

We started with an extensive search for open source projects that use CDL or Kconfig, beyond their host projects eCos and the Linux kernel. We used search engines to lookup websites with the “Kconfig” or “CDL” keyword, code search engines with corresponding file endings as keywords, and searched mailing lists and forums. We believe that our extensive search identified all currently existing open source models using CDL or Kconfig.

The qualitative analysis part focused on characterizing model contents and organizational structures reflected in the feature hierarchy. To analyze the content, we manually inspected the models and iteratively developed a classification schema for features, which is part of our conceptual framework. This schema was discussed with colleagues and then applied to characterize the content of each model. To identify patterns of feature organization, we inspected the first three levels of the configurator hierarchy for each model.

One interesting hit was for example a forum discussion about separating the Kconfig infrastructure from the Linux kernel to foster its adoption in other projects: http://forum.soft32.com/linux/PATCH-Start-genericize-kconfig-projects-ftopict347037.html
For the quantitative analysis, we developed our own analysis tool infrastructure CDLTools and LVAT\textsuperscript{2}. See Appendix A for further information. We first extended the original configurators of eCos and Linux to exploit their parsers and to export the relevant data (feature tree and feature properties) into our own format. We then loaded these files into our developed analysis infrastructure to calculate statistics, and to further transform the models into propositional formulas for SAT-based analyses, such as to check hierarchy rules or identify dead features. Fig. 5.1 schematically shows our quantitative analysis setup. Both infrastructures are freely available.

To characterize feature hierarchy, feature kinds, feature representation, and most importantly, constraints as the primary source of complexity, we re-used and defined appropriate metrics. For each model, their values are given in tables or visualized using reasonable diagrams throughout this chapter. Table 5.6 at the end of this chapter summarizes all metrics.

The qualitative analysis addresses RQ2.1 (model content) and RQ2.2 (model organization), while the quantitative analysis also covers RQ2.3 (constraints). For RQ2.4, we compare our results to common assumptions about models in the literature.

### 5.2. The Systems

Table 5.1 lists all gathered subject systems and the sizes of their variability models. One is a CDL model and twelve are Kconfig models. The models range from very small (ToyBox,
Table 5.1.: Model analysis case studies

<table>
<thead>
<tr>
<th>Language</th>
<th>Model</th>
<th>Version</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kconfig</td>
<td>Linux X86</td>
<td>2.6.32</td>
<td>6320</td>
</tr>
<tr>
<td></td>
<td>axTLS</td>
<td>1.2.7</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>BuildRoot</td>
<td>2010.11</td>
<td>1938</td>
</tr>
<tr>
<td></td>
<td>BusyBox</td>
<td>1.18.0</td>
<td>881</td>
</tr>
<tr>
<td></td>
<td>CoreBoot</td>
<td>4.0</td>
<td>2269</td>
</tr>
<tr>
<td></td>
<td>EmbToolkit</td>
<td>0.1.0-rc12</td>
<td>1357</td>
</tr>
<tr>
<td></td>
<td>Fiasco</td>
<td>2011081207</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>Freetz</td>
<td>1.1.3</td>
<td>3471</td>
</tr>
<tr>
<td></td>
<td>ToyBox</td>
<td>0.1.0</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>uClibc</td>
<td>0.9.31</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>uClinux-base</td>
<td>20100825</td>
<td>383</td>
</tr>
<tr>
<td></td>
<td>uClinux-dist</td>
<td>20100825</td>
<td>1620</td>
</tr>
<tr>
<td>CDL</td>
<td>eCos i386PC</td>
<td>3.0</td>
<td>1256</td>
</tr>
</tbody>
</table>

72 features) to very large (Linux, 6320 features). The average number of features is 1670 (median 1306); thus, our subjects are among the largest variability models known so far—recall that the largest model in the S.P.L.O.T. repository (Section 2.3.5) had only 290 features. In the following, we briefly introduce all our subject projects and their configurability.

5.2.1. eCos

As explained previously (Section 4.2.1), the embedded operating system eCos 3.0 supports 116 different hardware architectures (targets). An eCos model is aggregated in the configurator by choosing a target and a template. We decided to analyze the model of the i386PC target and the all template—the most inclusive template containing almost all hardware-independent packages. Our results are representative for eCos, since all architecture-specific models turned out to have many overlapping features, and similar characteristics in terms of size, diagram shape, and feature kinds and representation. This observation is further backed up by our follow-up study on constraints of all eCos models [PNX+11].

5.2.2. Kconfig Systems

The Kconfig language and its tools were designed for the Linux kernel and are developed and distributed together with the kernel codebase. Although it never became a standalone project, we discovered that Kconfig has been adopted by at least ten other open source projects in the systems software domain—perhaps naturally, as the strict resource requirements of such systems often require static configuration. We now introduce all our Kconfig projects.
5. Variability Models

**Linux kernel.** As explained previously (Section 4.2.2), the Linux kernel has an individual model for each architecture. We chose the most common one—the Intel X86 architecture. This model is distributed over 504 Kconfig files across the code base.

**axTLS.** AxTLS is a small, memory-optimized client/server library implementing the TLSv1 SSL protocol. It contains a tiny http and https server, test tools, and various interfaces for major programming languages, such as Java and C#. AxTLS’ model is rather small with 108 features distributed over five Kconfig files.

**BuildRoot.** BuildRoot is a tool for developers of embedded systems that generates a complete embedded Linux system with a root file system and all necessary packages, as opposed to just a kernel. The project is a large collection of scripts to stepwise generate the system. All steps are configurable and comprise: downloading and building a cross-development toolchain for the target architecture; building development and debugging tools; building core system programs and shell commands (preferably BusyBox- and uClibc-based, see below); as well as installing a kernel and boot loader. BuildRoot also has several hundreds of packages containing user space applications and libraries, such as GUI-, networking- or system-related programs.

**BusyBox.** BusyBox is a command-line tool for Linux-based embedded systems that combines many standard shell commands, such as `ls`, `cp` or `rm`, in a single executable. The BusyBox configurator allows customizing the executable by selecting only commands and capabilities needed on the target system. In particular, it allows linking BusyBox to uClibc (described shortly) to save even more space.

**CoreBoot.** The CoreBoot project delivers a free open source BIOS as an alternative to proprietary BIOS implementations in PCs and Workstations. CoreBoot provides the basic code that is necessary to initialize the mainboard with all its devices, such as RAM, PCI bus, and serial interface. After initialization, CoreBoot executes third-party payload, which can be a bootloader for an operating system, device-specific firmware (such as OpenBios, see below), or an operating system kernel directly.

**EmbToolkit.** EmbToolkit (Embedded Systems Toolkit) is a build system designed for embedded system developers similar to BuildRoot. EmbToolkit creates a cross-development toolchain with a custom C compiler, C library, and other development and debugging tools. The preferred C library is EGLIBC (another lightweight implementation of the standard C library), but uClibc can also be used alternatively. EmbToolkit generates a root filesystem containing core system tools including BusyBox and GUI-, networking- or system-related programs. These are installed as packages selectable in the configurator.
5.2. The Systems

**Fiasco.** Fiasco [Hoh02] is a derivative of the L4 microkernel family, used in conjunction with the real-time operating system DROPS\(^3\). Fiasco runs on a variety of systems, ranging from small embedded to large multi-processor architectures with Intel x86, ARM, or PowerPC processors. It supports preemptive multi-tasking with hard priorities for processes, hardware-assisted virtualization, in-kernel debugging, and provides an object-oriented kernel API. In contrast to the Linux kernel, Fiasco is less customizable; its configurability comprises the target hardware, debugging, and build configuration options, but not whole subsystems or drivers, which are outside the kernel.

**Freetz.** The Freetz (for *Free Fritz*) project provides an alternative firmware for consumer internet routers of the popular AVM FritzBox series. Freetz extends the proprietary firmware with extra functionality, such as an improved firewall, various servers (such as HTTP, VPN, SMB), and many other tools as packages. It also allows users to remove unnecessary features of the original firmware by selecting individual patches.

**ToyBox.** ToyBox is our smallest subject. It has the same goal as BusyBox: combining a subset of the GNU shell commands into one executable. It was started by a former BusyBox maintainer, who found that BusyBox was too difficult to extend. ToyBox currently implements 35 of BusyBox’ 309 commands and three additional ones. The project appears to have been a playground for the author. It is now largely abandoned, the last release dates back to the end of 2009.

**uClinux.** uClinux is a Linux distribution for embedded systems. At its core is a tailored version of the Linux kernel for micro-controllers, which today supports 14 hardware architectures, such as ARM, ADI Blackfin or MIPS. Originally created as a fork of the Linux 2.2 kernel, it is widely recognized today and its core parts had been merged into the official Linux kernel.

    The configuration of uClinux forgoes in a multi-level fashion (similar to staged configuration [CHE05b]) in three steps, each governed by a dedicated Kconfig model. First, basic features are configured: hardware architecture and libraries (the uClinux-base model in Table 5.1). Then the kernel is configured using templates for supported architectures. Finally, a wide variety of software packages can be selected from the uClinux distribution (the uClinux-dist model in Table 5.1). We study the model of the first and the third step. The second step model is essentially a Linux kernel model for the target hardware—similar to the mainline x86 kernel model already included in our study.

**uClibc.** Initially a sub-product of uClinux, the uClibc project is now an independent implementation of the standard C library for embedded microprocessors. It provides only a tailored subset of the functions present in the regular C library (glibc) used with Linux distributions, excluding functionality not needed on embedded systems. To further

\(^3\)http://os.inf.tu-dresden.de/drops/overview.html
address the space requirements, it can be configured to support a minimal set of needed functions, reflecting the needs of a given project.

The following three sections report the results of our models study. We first characterize the contents of our models, then their organizational structures, and finally analyze their constraints.

5.3. Model Content

Our subject models span fairly different domains and are used to configure diverse aspects of the projects. To illustrate their content, we report observations from our qualitative analysis in the first, and the quantitative analysis in the second part of this section.

5.3.1. Feature Themes

Although most features configure domain- and project-specific functional aspects of the systems, we found technical features that are less concerned about functionality, but instead configure the build process, debugging levels, the target hardware environment, and so on. To characterize the model contents, we defined themes of features by manual inspection. These themes are part of our conceptual framework. We distinguish between project-specific and technical feature themes, more precisely:

- **Project-specific** features are those representing the main, domain-specific content of a model. They either describe functional or non-functional aspects and belong to none of the following technical themes.

  Most features in the models are project-specific, for example *networking options* in Linux, *SSL encryption options* in axTLS, or the *JFFS2 filesystem* in eCos.

- **Build** features configure the build process of a system and have no impact on functionality. A sub-theme is *test cases*, which we define separately below.

  Examples of build features are *compilation (CC flags) and linker (LD flags) options*, but also *download sites* in projects that download software packages (such as EmbToolkit).

- **Deployment** features configure the installation process.

  Examples are installation options, such as the *target folder* for axTLS, but also decisions so as to *move files from the firmware image to a USB drive* in Freetz, or to *create symbolic filesystem links to the BusyBox executable*.

- **Diagnostics** features aim to provide runtime analysis facilities, such as debugging or profiling.

  Examples are the *BigInt Performance Test feature* in axTLS, a feature enabling *debugging symbols* in BusyBox, or a feature adding *tracing tools* to the Freetz firmware image.
5.3. Model Content

- **Hardware environment** features customize the system to run on a specific hardware, such as CPU, memory, or I/O devices. Typical examples are features that determine whether the processor supports APIC in Linux, set the router’s flash memory size in FreeTz, or configure serial ports in eCos.

- **Lifecycle** features configure explicitly deprecated or experimental functionality. Deprecated features are obsolete or not officially supported any more, but often remain in the model for compatibility or dependency reasons. Examples are the Open Sound System in Linux, the msh command in BusyBox, the PS/2 keyboard init in CoreBoot, or hardware architectures whose support is broken in uClibc. Experimental features enable functionality in alpha or beta mode, such as profiling support in Linux, a central configure cache file in BuildRoot, or the CYG_HAL_STARTUP feature’s value ROM in eCos.

- **I18N** features comprise internationalization options. Examples are features that select the firmware language (EN, DE, A-CH) in FreeTz, enable Unicode support in BusyBox, or configure timezone support in uClibc.

- **Imported** features were copied from other models. They often occur in projects that include other projects with their own variability models. For example, EmbToolkit includes both BusyBox and uClibc, therefore, most of their features were copied into EmbToolkit’s model.

- **External library** features configure included libraries in the project. Note that if the library has its own model, we classify copied features as imported. Examples are features that include certain shared libraries in FreeTz, configure the EGLIBC library in EmbToolkit, or select a specific thread library in BuildRoot.

- **Software environment** features configure the presence of certain software (libraries or applications) in the target runtime environment. Examples are features to select the target execution platform (Linux, Cygwin or Win32) in aTLS, to configure whether the platform has shadow passwords in uClibc, or to set the location of existing kernel modules in BusyBox.

- **Test case** features trigger and configure unit tests during the build process. Test cases exist for almost every major component in eCos. Sample features comprise HTTP server tests, POSIX CRC tests, or CPU load measurement tests.

5.3.2. Feature Classification

Table 5.2 shows all themes and their occurrence in each model. The models (columns) are ordered according to the number of feature themes they comprise; and themes (rows) according to the number of models containing them.
5. Variability Models

Table 5.2.: Themes of features in the models

<table>
<thead>
<tr>
<th>Theme</th>
<th>EmbToolkit</th>
<th>Freetz</th>
<th>eCos</th>
<th>BuildRoot</th>
<th>uClibc</th>
<th>uClinux-dist</th>
<th>BusyBox</th>
<th>Fiasco</th>
<th>axTLS</th>
<th>Linux</th>
<th>X86</th>
<th>CoreBoot</th>
<th>uClinux-base</th>
<th>ToyBox</th>
</tr>
</thead>
<tbody>
<tr>
<td>project-specific</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diagnostics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>build</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hardware env.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lifecycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>deployment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i18n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imported</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ext. library</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>software env.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test case</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While some models contain features of almost every theme, such as EmbToolkit and Freetz, others are very sparse, such as the minimalistic ToyBox model. Nevertheless, all models contain technical features in addition to “ordinary” project-specific features; mainly to configure diagnostics (debugging) and the build process.

We also observe that many models contain deprecated and experimental features (theme lifecycle), both to the same extent. However, no explicit concept for lifecycle features exists in the languages, although we know from experience that many companies need to support such features in their models. Instead, a distinguished feature often switches the visibility of lifecycle features, such as `BR2_DEPRECATED` (“Show packages that are deprecated or obsolete”) in BuildRoot or `EXPERIMENTAL` (“Prompt for development and/or incomplete code/drivers”) in Linux.

5.3.2.1. Feature Kinds

In Section 4.3.1, we introduced two classifications for different kinds of features: grouping and individual features, and the role of features. With respect to both classifications, most models are similar, but significant outliers exist.

**Grouping and individual features.** Recall that the configurator hierarchy shown to the user can deviate from the syntactic hierarchy in the models—in Kconfig due to the nesting of configs based on dependencies (although presented differently than menu
and menuconfig nesting$^4$); and in CDL due to re-parenting. Thus, we consider two statistics—the syntactic and the configurator grouping of features: the former by counting the grouping features (see feature kinds in Table 4.1); the latter by counting non-leaf features.

The proportion of syntactic grouping features (menus, menuconfigs, and choices) is similar among all Kconfig models, but very low with 3.5%; in contrast to the eCos model with 26%. Interestingly, the proportion of configurator grouping features differs significantly from the syntactic grouping in all Kconfig models; indicating that many configs are additionally nested in the configurator. This proportion ranges between 11% and 28% (average 19%), except for the two outliers CoreBoot and Freetz with only 4%. In the eCos model, the proportions of syntactic and configurator grouping features only differ by 2%, since some syntactic grouping features are leaves without children. Table 5.3 shows detailed numbers about grouping.

Inspecting the outliers CoreBoot and Freetz reveals different reasons for their low proportion of configurator grouping features. In CoreBoot, large groups exist that contain up to 293 of the leaf features; interestingly, most of these leaves are invisible derived features (mainboard-specific constants). When considering only the visible features in the hierarchy, this proportion is within the normal range (21%) again. In Freetz, the tree is significantly degenerated with one feature having 68% (2377) of all features as children—almost all are leaves. These children represent specific “terminfos” (holding characteristics of Unix consoles) for the ncurses library, which is used to build textual user interfaces. However, these 2377 features are not shown by default due to a visibility condition controlled by the feature “Show all items”.

**Roles of features.** We observe that every model contains user features, implementation features, and derived features. Capability features are difficult to identify, but are certainly contained in one third of the models.

Specifically, the percentage of user features (shown and modifiable by users) is similarly high among almost all models, ranging from 68% to 97%, with the outlier Coreboot (18%) due to its high degree of derived features (as explained above). For implementation features, we can only give upper bounds by counting those features that define a symbol that can be referenced in code (regardless of whether actually used). This upper bound is 96.5% in average for all Kconfig models, and 81% for eCos. The proportion of derived features is rather low, ranging from 1% to 18% (average 6%) among almost all models, but again with the outlier CoreBoot (78%). Capabilities are difficult to identify in Kconfig, since there is no explicit language support. A pattern we found is to prefix such features with HAVE_. Searching for this pattern reveals lower bounds: Linux: 0.8%, CoreBoot: 0.6%, uClibc: 0.8%, and none in any other Kconfig model. CDL has an explicit capability concept (interfaces); 11% of the features in the eCos model are capabilities.

---

$^4$The hierarchy induced by menus and menuconfigs is shown in the left window of the configurator and requires explicit drill-down by the user. Config hierarchies are shown by indentation in the right window and are, thus, more lightweight to navigate.
### Table 5.3: Grouping statistics

<table>
<thead>
<tr>
<th>Os Family</th>
<th>grouping</th>
<th>syntactic</th>
<th>configurator</th>
<th>difference</th>
<th>runtime XOR</th>
<th>mutex</th>
<th>runtime sum</th>
<th>model size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyBox</td>
<td>0%</td>
<td>4%</td>
<td>12%</td>
<td>3%</td>
<td>1%</td>
<td>2%</td>
<td>4%</td>
<td>1%</td>
</tr>
<tr>
<td>CoreBoot</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>EmbToolkit</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>Fiasco</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>FreezT</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>Linux</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>uClibc</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>uClinux-base</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>uClinux-dist</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
</tbody>
</table>

- **number of features**: an overall measure of the variability in a system, calculated as the sum of the number of features in each system. This is a comprehensive measure that includes all features, regardless of their grouping constraints. It is a useful measure for systems with complex variability models, as it provides an overview of the total variability in the system.

### Notations and Considerations

- **grouping constraints**: these constraints dictate how features are grouped together, allowing for more precise control over feature selection and configuration. Grouping constraints can be based on a variety of factors, such as the platform or use case.

- **model size**: this is a measure of the complexity of a system's variability model, calculated as the number of unique feature combinations that can be selected. A larger model size indicates a more complex variability model.

- **menuconfig, menu, optional tristate choice**: these are common feature selection mechanisms used in software systems, allowing users to add or remove features as needed.

- **component, package, features with children**: these are additional considerations that can affect variability, such as the presence of dependencies between features or the ability to add or remove entire packages.

- **mandatory tristate choice**: this is a feature selection mechanism that allows users to choose from one of three options, such as enabling, disabling, or configuring a feature for a specific purpose.

### Table 5.3: Grouping statistics

<table>
<thead>
<tr>
<th>Os Family</th>
<th>grouping</th>
<th>syntactic</th>
<th>configurator</th>
<th>difference</th>
<th>runtime XOR</th>
<th>mutex</th>
<th>runtime sum</th>
<th>model size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyBox</td>
<td>0%</td>
<td>4%</td>
<td>12%</td>
<td>3%</td>
<td>1%</td>
<td>2%</td>
<td>4%</td>
<td>1%</td>
</tr>
<tr>
<td>CoreBoot</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>EmbToolkit</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>Fiasco</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>FreezT</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>Linux</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>uClibc</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>uClinux-base</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
<tr>
<td>uClinux-dist</td>
<td>0%</td>
<td>26%</td>
<td>16%</td>
<td>22%</td>
<td>21%</td>
<td>11%</td>
<td>10%</td>
<td>4%</td>
</tr>
</tbody>
</table>

**Notations and Considerations**

- **grouping constraints**: these constraints dictate how features are grouped together, allowing for more precise control over feature selection and configuration. Grouping constraints can be based on a variety of factors, such as the platform or use case.

- **model size**: this is a measure of the complexity of a system's variability model, calculated as the number of unique feature combinations that can be selected. A larger model size indicates a more complex variability model.

- **menuconfig, menu, optional tristate choice**: these are common feature selection mechanisms used in software systems, allowing users to add or remove features as needed.

- **component, package, features with children**: these are additional considerations that can affect variability, such as the presence of dependencies between features or the ability to add or remove entire packages.

- **mandatory tristate choice**: this is a feature selection mechanism that allows users to choose from one of three options, such as enabling, disabling, or configuring a feature for a specific purpose.
5.3. Model Content

5.3.2.2. Feature Representation

Switch features are the basic and most common type in our models. Nevertheless, every model except the minimalistic ToyBox also contains features with data values (numbers or strings). Their proportions are rather low (0–11%) compared to switch features; yet, this observation calls for adequate language and tool support, especially with regard to constraints. Fig. 5.2 and Table 5.4 precisely show the breakdown of features by type\(^5\).

Surprisingly, in a quarter of our models, we find relatively high proportions of data features, in particular 27% in axTLS and more than 50% in eCos and CoreBoot. This observation is interesting, since the majority of examples found in the literature has few or no such features [SLB+10]. Further, Linux heavily uses the three-state logics for controlling binding mode; more than half of the features are of the tristate type. However, since support for loadable kernel modules is unique to Linux, no other model has any tristate feature.

Supporting number (int, hex, float) and string data features appears to be equally important in most models; their proportions are similar, but slightly tending to string features. Only Linux and CoreBoot have significantly more number than string features.

Usage of data features. Considering the models with high proportions of data features—axTLS, CoreBoot, and eCos—shows that data features are used for diverse purposes.

In axTLS, data features configure the built-in webserver (such as port, ssl expiry time, folders), paths to external libraries (such as Java, Perl), or SSL certificate details (such as common name, organization name). However, the high percentage of data features might be biased by the rather small model.

\(^5\)Note that the eCos percentages not add up, since features can be both switch and data (type booldata) in CDL.
### Table 5.4: Feature Representation

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Linux</th>
<th>axTLS</th>
<th>BuildRoot</th>
<th>BusyBox</th>
<th>CoreBoot</th>
<th>EmbToolkit</th>
<th>Fiasco</th>
<th>Freetz</th>
<th>ToyBox</th>
<th>uClibc</th>
<th>uClinux-base</th>
<th>uClinux-dist</th>
<th>eCos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectable</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
</tr>
<tr>
<td>String</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
</tr>
<tr>
<td>None</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>Number</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>Integer</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectable</td>
<td>42%</td>
</tr>
<tr>
<td>String</td>
<td>12%</td>
</tr>
<tr>
<td>None</td>
<td>2%</td>
</tr>
<tr>
<td>Number</td>
<td>4%</td>
</tr>
<tr>
<td>Integer</td>
<td>4%</td>
</tr>
</tbody>
</table>

Number of Features: 90
In CoreBoot, almost every data feature (98%) is derived, invisible, and represents a constant (e.g., number of IRQ slots, mainboard-specific source folders). These constants exist for each of the 166 mainboards supported (as explained previously).

In eCos, some feature kinds contain data values by default: interfaces always carry a number (count of implementing features that are enabled), and packages always have the flavor `booldata`, with the data part representing the package version as a string. 15% of eCos’ features belong into this category. Further, 2% of features represent enumerations. There are also 6% of features representing compiler flags, 0.3% linker flags, and 3% holding names of files with test code. The remaining data features (28% of all features) represent diverse configuration constants, such as priorities, buffer sizes, and supported I/O ports. Apparently, many of these constants are specific to an embedded operating system and would either be set dynamically or not be configurable in a system like the Linux kernel.

5.4. Organization and Hierarchy

This section describes the organizational structure and summarizes characteristics of the feature hierarchies found in the models. The first part reports qualitative observations and aims at understanding how the systems are decomposed into features. The second describes quantitative measures of the configurator hierarchies, aiming at providing useful assumptions for tools in order to reasonably visualize models.

5.4.1. Organizational Structures

Our analysis shows that projects use different strategies to group features. The strategies vary not only from project to project, but also within a project. For example, some features are grouped together by their functionality, such as networking and filesystem features, while others are grouped by the mechanism by which variability is realized, such as features that are applied as patches or compiler flags.

We describe model composition strategies by showing how each project organizes features. We start with Freetz as it uses many different strategies. We then proceed with the remaining Kconfig projects, and finally describe the organization of the Linux kernel and the eCos operating system. Summaries of the Freetz, Linux, and eCos feature hierarchies are shown in Fig. 5.3a, Fig. 5.3b, and the left-hand side of Fig. 5.4. Each box represents a grouping feature labeled by the feature name, the number of its descendants (excluding descendants of the sub-groups that are already shown in the figure), and a label\(^6\) indicating the theme of the group according to Section 5.3.1 and Table 5.2, if applicable. The height of each box indicates the number of features within the group.

**Freetz.** The Freetz model—summarized in Fig. 5.3a—is a prime example of a project that uses different strategies to group features. The “hardware type” group allows detailed configuration of the hardware, such as WLAN version. These features are

\(^6\)BLD=Build, DIA=Diagnostics, HDW=Hardware environment, LIB=External library
Figure 5.3.: Summarized Freetz and Linux hierarchies
grouped according to the *hardware env.* theme from Table 5.2. The “patches” group contains features that are applied as patches to the code to change the system by removing branding, help, altering storage names, and so on. The strategy used for this group is the variability mechanism by which such features are realized. The “package selection” group contains options to include certain utility packages, such as curl (a command line tool for transferring data) and inetd (a server daemon for internet services). In general, this group contains diverse features that all configure functionality (*project-specific*).

It contains the groups “standard packages”, “web interface”, “debug helpers”, “testing”, and “unstable”. The features in “debug helpers” are grouped by the *diagnostics* theme, and the “testing” and “unstable” packages by the lifecycle theme. Lastly, the “advanced options” group contains a large number of configuration options that can be used to: configure package download sites; add external processing features (such as IP anonymizer and bittorrent server); configure BusyBox; add modules from the Linux kernel; add cryptography, compression, and other shared libraries; and to set compiler options.

In summary, Freetz uses a variety of strategies to organize features. A common strategy that Freetz uses is to group features by one of the themes from Section 5.3.1 and Table 5.2. Some strategies, however, follow an even more specific theme, such as the package download sites feature, which can be considered a sub-theme of the *build* theme. Finally, some strategies, such as compiler options and patches, are cross-cutting around themes. For example, the features “Remove dtrace” and “Remove ftpd” are features located in the patches group, given that dtrace and ftpd are removed from the product by applying patches to the software. However, as dtrace and ftpd are external libraries, these features clearly cross-cut the *external library* theme.

**BuildRoot, EmbToolkit, uCLinux.** These projects group features by hardware architecture (theme *hardware environment*) and by the root file system (theme *project-specific*). The choice of architecture affects values of architecture-dependent features using defaults and visibility conditions. Unlike the other two projects, uCLinux separates the configuration of architecture and root file system through staged configuration (see Section 2.2.2). A configurator is initially launched for the architecture selection. Depending on the choices made in the first configurator, a different default configuration for the root file system is used.

**axTLS, CoreBoot, uClibc.** These projects use the same strategy as above where an architecture choice (theme *hardware environment*) affects the choices of architecture-specific features (theme *project-specific*). In axTLS, the architecture is a platform choice (such as Linux, Cygwin or Win32), CoreBoot’s architectures are motherboards (e.g. AMD or Intel), and uClibc has processor architectures (such as Alpha, ARM or i386). Interestingly, CoreBoot extensively uses multiple declarations of a single feature to define mainboard-specific constants (as pointed out previously in Section 5.3.2.1). For example, the `BOARD_SPECIFIC_OPTIONS` feature is declared 142 times. The model is modularized such that each motherboard is declared in its own Kconfig file. CoreBoot relies on the configurator merging the multiple declarations into a single feature.
Figure 5.4.: Summarized eCos model hierarchy. 64% of the features can easily be mapped to architectural concerns. The size of the feature boxes indicates the scale of the corresponding subtree.
5.4. Organization and Hierarchy

Figure 5.5.: eCos architecture (adapted from [Mas03]). Shaded architectural concerns could not be mapped to grouping features in the model.

**Fiasco.** Fiasco’s hierarchy has only four top level groups. Like the previous projects, it starts with a group of hardware features (theme *hardware environment*) comprising architecture (Intel, AMD64, ARM), platform (PC or Linux usermode), CPU, and more detailed options that all affect derived invisible constants used in the remainder of the model. This group is the largest in the model with 105 features. Thereafter, a group of only 16 features configures the functionality of the kernel (theme *project-specific*); while the third group comprises debugging (theme *diagnostics*; and the fourth compiler options (theme *build*).

**BusyBox, Toybox.** These two projects separate their features into two groups: build-related features that affect compilation (theme *build*), and by the shell commands (theme *project-specific*). ToyBox has two top-level menus for these groups. BusyBox, being the larger project, further groups the shell commands into sub-categories, such as archival, console, or networking.

**Linux.** Although, as explained shortly, the Linux model hierarchy has a depth of 8, we found that, for the purpose of describing the overall organization, it is sufficient to present only the top hierarchy level, as shown in Fig. 5.3b. Differently from Freetz, Linux’ top level of groups is already very specific.

Similar to Freetz, top level groups are about core hardware configuration: “General setup”, “Enable the block layer”, “Processor type and features”, “Power management and ACPI options”, and “Bus options: PCI, etc”. The remaining groups—except “Kernel hacking”, “Security options”, and “Library routines”—are for configuration of different functionality (such as networking, file systems, and cryptography), devices, and architectural components, The “Kernel hacking”, “Security options” and “Library
routines” are groups of features that cross-cut functionality and architectural component
groups. While “Security options” and “Library routines” are grouped by the *project-
specific* theme, the “Kernel hacking” features are grouped for their common *diagnostics*
theme.

We conclude that Linux’ main strategy for grouping features is their common function-
ality, architectural component or hardware. While experimental or deprecated features
(theme *lifecycle*, for example, could also be grouped together, Linux gives priority to
grouping them by functionality or by their architectural components. Alternatively
in Freetz, *lifecycle*-themed features are put into separate groups, such as testing and
unstable, but these cross-cut the functionality groups. Although Linux does not place
*lifecycle*-themed features into groups, they are tagged with a dependency on the EXPER-
IMENTAL feature. This allows their visibility to be toggled by enabling or disabling
EXPERIMENTAL. Examples are the features “User namespace” in the “Namespaces
support” group and “PCI Express ASPM support” in the “PCI Express support” group.
Both are only visible when EXPERIMENTAL is selected.

eCos. In eCos, the variability model is aggregated from smaller models that are dis-
tributed over the 500 packages in the codebase. Each package forms a subtree with a
feature of kind package at its root. By default, all these subtrees become children of
the synthetic root of the aggregated model, except for reparented features. We find two
common use cases for reparenting: First, to place global build options under a top-level
*component* with this name. Second, to place packages into the subtree of another package.
For example, many core hardware-specific packages are reparented into the “eCos HAL”
(Hardware Abstraction Layer) package.

The organization of the model can be characterized as follows. The first child of the
synthetic root node is “Global Build Options” containing the aforementioned reparented,
bundle-specific features from several packages. Next child is the package “eCos HAL” with
hardware-specific options, and into which other hardware packages are mounted, such as
the many i386-specific packages. If the user selects another target (hardware architecture)
in the configurator, other packages would be mounted into this HAL subtree. Thereafter,
the packages for the I/O subsystem and several rather technical packages appear, such
as the configuration of the eCos kernel or of various C libraries, such as libc, libm (math)
or snmilib. The rest of these top-level packages comprise more application-oriented
functionality, such as networking, clients and servers, but also the filesystems supported
in the final eCos instance.

In summary, eCos features are grouped largely by having a common architectural
component. Fig. 5.4 shows the eCos model with links from the groups to the architectural
concern for which the group is responsible for configuring. These concerns were extracted
from an eCos book [Mas03], which we reproduce in Fig. 5.5. For Linux, we did not
find such a clear mapping from feature groups to architectural concerns, using the
interactive Linux kernel map7 as a reference. Linux has a much more fine-grained and
complex architecture and feature model. For example, although the Linux architecture

---

7 [http://makelinux.net/kernel_map](http://makelinux.net/kernel_map)
has a networking component, it is subdivided into “socket access”, “protocol families”, “protocols”, “virtual network device”, and “network device drivers”. Such subdivisions are not explicit in the Linux model.

5.4.2. Model Hierarchies

We quantitatively analyzed the configurator hierarchies of our subject models. To give an impression of their shapes, we first provide plots of the three smallest models, ToyBox, axTLS, and Fiasco, in Fig. 5.6.

Shape. Our analysis shows that all models are wide and shallow. Their average depth ranges between 3 and 4 (shallow outlier CoreBoot with 2). The maximal depth is as low as 4 for uClibc and uClinux-base and not more than 8 for the huge Linux model; see the leaf-depth distributions in Fig. 5.7b. At the same time, branching factors (number of children per feature) vary to a great extent in our models, which contrasts the nicely balanced trees in literature. Although the vast majority of features (83% in average) are leaves, we observe many features with more than 100 children. Practically, none of these models could be rendered as a tree structure like in Fig. 2.6 or Fig. 4.3, which is the common visualization in literature. The Linux kernel model would only be a flat line if plotted with the scaling of the models in Fig. 5.6.

Further analysis shows that in all models, the number of features with a given number of children decreases sharply with the increase of the number of children. Fig. 5.7a shows histograms of branching factors in the models. It excludes leaves, which represent the majority of features in the models (72% to 96%, 83% average). The second-largest class are single-child parents (7% average), followed by two-child parents (3% average). Features with more than ten children are very seldom; nevertheless, the maximum number of children (maximal branching) is as much as 158 in Linux and 29 in eCos. Among all models, the median of maximum branching factors is 84; however, we find outliers with 173 (uClinux-dist), 293 (CoreBoot), and whopping 2377 (Freetz) child features.

Hierarchy rules. Relatively few features violate hierarchy rules—child-to-parent implications—of feature modeling. Thus, we believe that practitioners find hierarchical organization of dependencies natural. Recall that, unlike in feature modeling and CDL, Kconfig uses hierarchy to depict a visibility relation instead of a presence condition, allowing a child feature to be configured without its parent. This possibility is indeed exploited in the Linux model. Sometimes, children even exclude their parent. We verified with a SAT solver applied to the derived boolean semantics of the Kconfig models (see Section 4.5.8) that all models except axTLS, Fiasco, and ToyBox contain features not implying their parents in the configurator hierarchy. Fig. 5.7c shows these proportions among all models. A nice example from the Linux model is the conditionally derived feature JFFS2_ZLIB in Fig. 4.3 (Line k-32), which is automatically selected if the parent is not, since JFFS2_ZLIB is a conditionally derived features, as we explained in Section 4.5.5.

In eCos, all features in the configurator hierarchy imply their parent. However, we
Figure 5.6.: Hierarchy plots of the three smallest models
found 39 (3%) re-parented features, which not imply their syntactic parent anymore. Most re-parentings move packages in the hierarchy, but 10 options and two components were re-parented as well. For example, the GLOBAL_OPTIONS component from HAL_1386_PC package was promoted to the top-level and, in addition to its syntactic children, two new options were re-parented under this component.

Our observations indicate special needs to develop modeling interfaces: first, to support wide and shallow models; second, to support high variation in branching from very limited to very wide. Furthermore, variability modeling languages have to support visibility control (discussed shortly in Section 5.5.2) to suppress inactive features or whole subtrees in the configurator.

## 5.5. Constraints

Complementing the constraints residing in the hierarchy (child-parent implications), each model has additional constraints declared over features. This section reports observations about group constraints and the various types of feature constraints.
5. Variability Models

5.5.1. Group Constraints

Feature groups are among the core concepts in feature modeling. In fact, groups are regularly used in all of our models. But surprisingly, OR groups—most commonly mentioned in literature—are neither supported by the Kconfig language nor occur in the eCos model. Kconfig’s slightly similar grouping concept—runtime XOR (cf. Section 4.5.4)—appears only twice in Linux. Instead, the most frequent type of group constraints is XOR, which is contained in every model except ToyBox. Table 5.3 (grouping with constraints) shows detailed numbers.

In Linux and eCos, less than 1% of the features impose group constraints on their children. The other models have higher percentages. Among all, the average is 4%, whereas outliers are EmbToolkit with 8% and uClinux-base with even 19% of features representing XOR groups. MUTEX groups are very rare—only one exists in the eCos model.

The insignificance of OR and MUTEX groups is surprising. We speculate that both are realized separately with constraints, such as dependencies to a capability. Unfortunately, we cannot measure the latter due to a lack of a syntactic capability concept in Kconfig (cf. Section 4.5.1). Identifying such implicit semantic groups is possible, but requires significant effort using SAT-based analysis, which is beyond our scope.

Let us see how group constraints are used in practice. The two runtime XOR groups in Linux are motivated by binding time: this constraint allows including multiple alternative features in the configured kernel as dynamically loadable modules; only one of them will be loaded at runtime. The only MUTEX group in eCos represents three alternative random number generators. A possible reason for the lack of MUTEX groups in Kconfig models is the need to define a build symbol even when no group member is selected, see for example the feature JFFS_CMODE_NONE in Fig. 4.3.

Recall that CDL interfaces generalize group cardinality constraints. This generality is not exploited in practice, though. There is no cardinality constraint that is a proper \((m, n)\)-interval, as opposed to intervals with lower bound of 0 or 1 and upper bound 1 or *. Moreover, although an interface can place a group constraint on features that are not siblings, all interfaces are implemented by sibling features. Still, interfaces and implementing features are usually far apart, that is, do not have a common parent and are implemented across different packages. In other words, the group constraint is activated (implied) by the parent of the interface, which is not the parent of the set of constrained features. This form of a group constraint is more general than what is found in feature modeling, where the parent of the group activates the group constraint. Such generalized group constraints are used to model the case where a given package defines an interface required by its implementation and multiple other packages provide alternative implementations of that interface. This case is relatively frequent, 81 interfaces are constrained this way in the eCos model.

In general, it suffices to include n-ary XOR, OR, and MUTEX operators in the constraint language and in tools. Since only basic cardinalities are used in eCos, CDL’s interfaces appear overly general. However, they represent capabilities and, thus, improve modularity in the eCos model.
<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Linux</th>
<th>axTLS</th>
<th>BusyBox</th>
<th>CoreBoot</th>
<th>EmbToolkit</th>
<th>Fiasco</th>
<th>Fretz</th>
<th>ToyBox</th>
<th>uClibc</th>
<th>uClinux-base</th>
<th>uClinux-dist</th>
<th>eCos</th>
</tr>
</thead>
<tbody>
<tr>
<td>any constraint</td>
<td>92%</td>
<td>77%</td>
<td>60%</td>
<td>95%</td>
<td>86%</td>
<td>61%</td>
<td>85%</td>
<td>98%</td>
<td>96%</td>
<td>75%</td>
<td>74%</td>
<td>74%</td>
</tr>
<tr>
<td>configuration constraint</td>
<td>88%</td>
<td>41%</td>
<td>48%</td>
<td>64%</td>
<td>82%</td>
<td>48%</td>
<td>77%</td>
<td>92%</td>
<td>37%</td>
<td>51%</td>
<td>1%</td>
<td>53%</td>
</tr>
<tr>
<td>value restriction</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
<td>2%</td>
<td>0%</td>
<td>0%</td>
<td>0.1%</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>unconditionally derived</td>
<td>12%</td>
<td>1%</td>
<td>2%</td>
<td>1%</td>
<td>78%</td>
<td>17%</td>
<td>23%</td>
<td>1%</td>
<td>6%</td>
<td>18%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>conditionally derived</td>
<td>3%</td>
<td>0%</td>
<td>0.4%</td>
<td>0%</td>
<td>0.4%</td>
<td>13%</td>
<td>0%</td>
<td>1%</td>
<td>0%</td>
<td>0.3%</td>
<td>0%</td>
<td>0.2%</td>
</tr>
<tr>
<td>visibility condition</td>
<td>5%</td>
<td>10%</td>
<td>5%</td>
<td>1%</td>
<td>3%</td>
<td>22%</td>
<td>3%</td>
<td>1%</td>
<td>0%</td>
<td>5%</td>
<td>18%</td>
<td>2%</td>
</tr>
<tr>
<td>explicit default</td>
<td>16%</td>
<td>59%</td>
<td>12%</td>
<td>92%</td>
<td>4%</td>
<td>10%</td>
<td>11%</td>
<td>94%</td>
<td>90%</td>
<td>41%</td>
<td>0%</td>
<td>46%</td>
</tr>
<tr>
<td>expression (computed)</td>
<td>1%</td>
<td>0%</td>
<td>0.1%</td>
<td>0%</td>
<td>0.3%</td>
<td>0%</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
<td>0.3%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>literal</td>
<td>15%</td>
<td>59%</td>
<td>12%</td>
<td>92%</td>
<td>3%</td>
<td>10%</td>
<td>10%</td>
<td>94%</td>
<td>90%</td>
<td>41%</td>
<td>0%</td>
<td>46%</td>
</tr>
<tr>
<td>CTCR metric(^1)</td>
<td>93%</td>
<td>70%</td>
<td>71%</td>
<td>79%</td>
<td>96%</td>
<td>88%</td>
<td>87%</td>
<td>96%</td>
<td>46%</td>
<td>75%</td>
<td>55%</td>
<td>68%</td>
</tr>
<tr>
<td>model size(^2)</td>
<td>6,320</td>
<td>108</td>
<td>1,938</td>
<td>881</td>
<td>2,269</td>
<td>1,357</td>
<td>171</td>
<td>3,471</td>
<td>71</td>
<td>369</td>
<td>383</td>
<td>1,620</td>
</tr>
</tbody>
</table>

\(^1\) Cross Tree Constraints Ratio (percentage of features participating in cross-tree constraints)  
\(^2\) number of features
5. Variability Models

5.5.2. Feature Constraints

All models declare additional configuration, default, and visibility constraints. All features have a parent—except the synthetic roots—and in the majority of the models, we find that there are more features with dependencies to other features across the hierarchy (cross-tree constraint) than features without. In the following, we discuss the frequency and usage of the various types of feature constraints, and the number of cross-tree dependencies per feature. The latter is defined as the reference of another feature in a constraint. Our observations are supported by Table 5.5, which shows the percentage of features declaring a certain type of constraint, and Fig.5.8 and Fig.5.9, which show dependencies and their growth.

5.5.2.1. Types of Constraints

The vast majority of features, in average 77% among all models and up to 98% in Freetz, declare constraints of some sort (configuration, visibility, default), as can be seen in Table 5.5. In the following, we explore the usage and quantity of different types of constraints.

**Derived features** are mostly used to perform calculations that otherwise would be hidden in the build system. This way, feature dependencies are specified uniformly and explicitly in one model. Recall that Linux supports conditionally derived features, which are derived or user-changeable with a default value, depending on a condition. 3% of Linux features belong into this category, whereas 12% (Linux) and 18% (eCos) of features are unconditionally derived.

**Visibility** control is essential in the models. All except ToyBox declare explicit visibility conditions, in average 7% of the features. For example, in Linux, 5% of features have an explicitly specified prompt condition (like JFFS2_ZLIB in Fig.4.3, Line k-32), rather than just via depends on, and 10% of features in eCos use active_if.

Two language constructs are commonly used in the models: a pure configuration constraint (like requires) and a combined configuration-and-visibility condition (like active_if).

**Default values** (also computed) are used a lot in the models, saving the user unnecessary configuration work. All models except uClinux-base declare explicit defaults. However, their proportions differ significantly: Slightly more than a third of our models have low (<20%) percentages of features with explicit defaults, such as Linux (16%). The others make heavy use of explicit defaults: for at least half of the features (uClibc, uClinux-dist, and axTLS); or significantly more: 69% in eCos, and at least 90% in ToyBox, BusyBox, and Freetz. Only four models compute (via expression) their defaults: 4% of features.

---

8In [BSL’10], we discounted unconditionally and conditionally derived features in the Linux model, since these are implicitly given by either omitting the feature prompt, or setting a conditional default.
in Linux, 10% in eCos, and only one feature in both BuildRoot and uClibc. All other
defaults are specified with literals.

5.5.2.2. Dependencies

To characterize dependencies, we discuss two metrics: the number of features referenced
in constraints of a feature, and the Cross Tree Constraints Ratio (CTCR), a metric
adapted from [MWC09].

The average number of features referenced among all features from all models is 1.06
(median 1). In Linux, most features refer to 1–3 other features (maximum of 52); this
range is much lower in eCos, with typically 0–1 cross-tree dependencies (maximum of 20).
Some features declare a large number of cross-tree dependencies; up to 127 in BuildRoot
and 101 in EmbToolkit. These outliers call for adequate tool support.

We visualize the number of dependencies per feature as a boxplot in Fig.5.8. Note
that these boxplots also count the dependency on the parent feature (+1). Interestingly,
the average number of dependencies per feature seems to grow linearly with the size
of the models and does not explode, as can be seen in Fig.5.9. This finding is in line
with our observations from a study of the Linux model evolution [LSB+10]. It indicates
that feature models abstract over implementation dependencies, since code dependencies
usually grow quadratically [Egy03].

Table 5.5 also provides an adapted version of the CTCR metric [MWC09] (named ECR in [Men09, MWCC08]) for all models. It provides the percentage of features that
have a dependency or are the dependency target of another feature—put simply, the
proportion of features participating in cross-tree constraints. This metric turns out to be
surprisingly high with 75% in average among all our models.
5. Variability Models

5.5.2.3. Examples.

Let us look at some examples of constraints in our two main models. In eCos, most constraints are logical expressions, but arithmetic and string operations are not uncommon. For example:

```c
cdl_option CYGNUM_FS_FAT_NODE_POOL_SIZE {
    display "Node pool size"
    flavor data
    default_value { (CYGNUM_FILEIO_NFILE + 2) }
    legal_values 1 to 9999999999
    requires { CYGNUM_FS_FAT_NODE_POOL_SIZE >= (CYGNUM_FILEIO_NFILE+2) }
}
```

```c
cdl_option AT91_CLOCK_SPEED {
    display "CPU clock speed"
    calculated { AT91_CLOCK_OSC_MAIN * AT91_PLL_MULTIPLIER / AT91_PLL_DIVIDER / 2 }
    legal_values { 0 to 220000000 }
    flavor data
}
```

String concatenation (denoted by “.”) is often used to produce lists of test or implementation source files:

```c
option CYGPKG_LIBC_STDIO_TESTS{
    display "C library stdio function tests"
    calculated ["tests/sprintf1 tests/sprintf2 tests/sscanf tests/stdiooutput " .
        ((CYGPKG_IO_FILEIO && CYGPKG_FS_RAM) ? "tests/fileio " : "") ]
    flavor data
}
```

Other constraints check whether a particular file name is included in a list; e.g. requires is_substr(LIBS, "libtarget.a"). Such constraints are part of the feature-to-code code mapping. In Linux, these are computed in KBuild (see Section 4.5.6), outside the model.
5.6. Conclusions

The constraints in the Linux model are mostly logical expressions, such as a single feature or more complex expressions, such as:

```c
config X86_SMP
  bool depends on SMP && ((X86_32 && !X86_VOYAGER) || X86_64)
  default y
```

Linux constraints often reference integer or string features using equality tests, for example:

```c
menuconfig DRM
  tristate "Direct Rendering Manager (XFree86 4.1.0 and higher DRI support)"
  depends on (AGP || AGP=on) && PCI && !EMULATED_CMPXCHG && MMU
  select I2C
  select I2C_ALGOBIT
```

In a single case, an integer feature in Linux uses another feature as a bound in a range constraint:

```c
config SERIAL_8250_RUNTIME_UARTS
  int "Number of 8250/16550 serial ports to register at runtime"
  depends on SERIAL_8250
  range 0 SERIAL_8250_NR_UARTS
  default "4"
```

5.6. Conclusions

Our quantitative analysis shows which of the previously identified variability modeling concepts are used in models. These findings directly constitute requirements for tools and languages, with priorities based on the frequency of concept occurrence. Our quantitative analysis also provides realistic properties about the structure and content of variability models, challenging some assumptions in the literature.

**Feature types and constraints.** We observe that switch features are the basic and most common type; nevertheless, data features are important for almost every model. In addition to arbitrary Boolean constraints (including mutual exclusion), the constraint language should support constraints on data features, using arithmetic and string operators. These appear to be typical for systems software. In particular, eCos intensively uses arithmetic operators and comparisons.

Our results also indicate that visibility conditions and derived features with expressive constraints—both concepts being largely neglected in academic languages—are essential for large-scale models. Moreover, both languages offer modularization capabilities that are exploited in practice; however, as we will see later, modularized variability models do not enable distributed variability management—more precisely, we found no empirical evidence.

**Cardinalities.** Group and feature cardinalities were introduced as FODA extensions (cf. Section 2.3.3); however, their benefit is still discussed in the research community. We have, so far, not found any empirical evidence for the need of feature cardinalities.
5. Variability Models

```tcl
for { set pin 0 } { $pin < 16 } { incr pin } {
    cdl_option CYGHWR_HAL_M68K_MCF5272_GPIO_PORTC_PIN[set pin] {
        display "Configure pin PC[set pin]"
        flavor data
        legal_values { "in" "out0" "out1" }
        default_value \n            is_substr(CYGHWR_HAL_M68K_MCF5272_BOARD_PINS, "c[set pin]_in") ? "in" : \n            is_substr(CYGHWR_HAL_M68K_MCF5272_BOARD_PINS, "c[set pin]_out0") ? "out0" : \n            is_substr(CYGHWR_HAL_M68K_MCF5272_BOARD_PINS, "c[set pin]_out1") ? "out1" : 
                "invalid"
        description "Pin PC[set pin] can be configured as a GPIO input, or a GPIO output (initial value 0 or 1)."
    }
}
```

Figure 5.10.: Embedded for loops in a CDL model

Although arbitrary group cardinalities are supported in CDL, the model makes no use of cardinalities other than XOR and MUTEX. Feature cardinalities [CHE05a], which enable multiple “instantiations” of features, are not supported by our languages—perhaps not surprisingly, since even in feature modeling, these are heavyweight extension compared to group cardinalities. However, we found embedded Tcl for loops in some eCos models that generate a number of occurrences of a feature in the configurator, which indicates that developers—although rarely—might find feature cardinalities useful. In some cases, even the number of generated features varied depending on the processor type (determined by the currently loaded target). Fig.5.10 shows an example of 16 generated feature “instances” in CDL.

**Model assumptions.** Many evaluations of academic variability modeling techniques rely on generated models based on assumptions about structure and constraints. Two examples are the works of Thüm et al. [TBK09] and Mendonça et al. [MWC09], who both present reasoning techniques for feature models. Our results significantly challenge their assumptions. Thüm et al. generate trees with maximal branching factors of 10 (too low, see Section 5.4.2), with 25% of inner features representing OR groups (too high, see Section 5.5.1), and 10% of all features having additional constraints (too low, see Section 5.5.2.1). Mendonça et al. [MWC09] assume an average CTCR of 30% (too low, see Section 5.5.2.2).

**Model content and organization.** Our qualitative analysis shows that the models are used to configure nearly every variable aspect of their project, including the build process, deployment, and test cases. However, we found very few “lifecycle” features that deal with deprecated or experimental functionality in our open source models. This finding is surprising, since supporting old features is a common requirement in industrial variability modeling.

Although we found that many features are grouped according to a common hardware
component, there is so far no indicator that the feature hierarchy in any way resembles the software architecture of the project. In combination with the linear growth of feature dependencies (Fig. 5.9), these findings confirm that variability models are orthogonal structures that abstract over the solution space. In effect, variability models cannot be seen as views on the implementation\(^9\), thus, cannot easily be generated due to the amount of unique domain- and project-specific knowledge contained in them. An approach that tries to mitigate this problem to support the reverse-engineering of feature models, which builds upon our results, is provided in [SLB\(^+\)11]. It emulates the domain- and project-specific knowledge using textual metrics over existing feature descriptions.

\(^9\)However, they can be seen as views on ontologies that model the project-specific concepts and their relationships [CPKK06]. In fact, in [SLB\(^+\)11], we derived a feature model of the FreeBSD kernel from an ontology we created by traversing specific types of relationships beginning with one root concept. Both the ontology and the feature model are available at: http://code.google.com/p/variability/wiki/FreeBSDOntology.
### 5. Variability Models

<table>
<thead>
<tr>
<th>Category</th>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>number of features</td>
<td>N</td>
<td>Model size.</td>
</tr>
<tr>
<td>Feature Kinds</td>
<td>configurator grouping features</td>
<td>N/P</td>
<td>Non-leaf nodes in the configurator’s hierarchy view.</td>
</tr>
<tr>
<td></td>
<td>syntactic grouping features</td>
<td>N/P</td>
<td>Non-leaf nodes in the model’s hierarchy (syntactic nesting).</td>
</tr>
<tr>
<td>Feature Roles</td>
<td>user features</td>
<td>N/P</td>
<td>Features that represent configuration options editable by the user.</td>
</tr>
<tr>
<td></td>
<td>implementation features</td>
<td>N/P</td>
<td>Features that can be referenced in code or build system (solution space and mapping).</td>
</tr>
<tr>
<td></td>
<td>derived features</td>
<td>N/P</td>
<td>Features with computed value.</td>
</tr>
<tr>
<td></td>
<td>capability features</td>
<td>N/P</td>
<td>Features that represent abstracted functionality.</td>
</tr>
<tr>
<td>Feature Representation</td>
<td>switch features</td>
<td>N/P</td>
<td>Features with a switchable value by the user, such as Boolean (selected/unselected) or three-state values.</td>
</tr>
<tr>
<td></td>
<td>data features (number/string)</td>
<td>N/P</td>
<td>Features taking a number or string as value.</td>
</tr>
<tr>
<td></td>
<td>none features</td>
<td>N/P</td>
<td>Features not editable by the user, and not representing a value.</td>
</tr>
<tr>
<td>Feature Hierarchy</td>
<td>leaf-depth</td>
<td>Dist. (N)</td>
<td>Nesting level of leaf nodes in the configurator hierarchy, starting with 0 for the root node.</td>
</tr>
<tr>
<td></td>
<td>branching factors</td>
<td>Dist. (N)</td>
<td>Number of children per feature.</td>
</tr>
<tr>
<td></td>
<td>average leaf-depth</td>
<td>N</td>
<td>Average depth of leaves.</td>
</tr>
<tr>
<td></td>
<td>maximal leaf-depth</td>
<td>N</td>
<td>Maximal configurator hierarchy depth.</td>
</tr>
<tr>
<td></td>
<td>maximal branching</td>
<td>N</td>
<td>Maximal number of children per feature.</td>
</tr>
<tr>
<td>Hierarchy Rules</td>
<td>features not implying parent</td>
<td>N/P</td>
<td>Number of features c that can be selected even though their parent p is not:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[ { c \mid \exists \sigma \in { \cdot } . \sigma (c) \land \neg \sigma (p) } ]</td>
</tr>
<tr>
<td></td>
<td>features not implied by child</td>
<td>N/P</td>
<td>Features p with at least one child c that does not imply it: [ { p \mid \exists \sigma \in { \cdot } . \sigma (c) \land \neg \sigma (p) } ]</td>
</tr>
<tr>
<td>Constraints</td>
<td>OR groups</td>
<td>N/P</td>
<td>Features imposing an OR group constraint among their children.</td>
</tr>
<tr>
<td></td>
<td>XOR groups</td>
<td>N/P</td>
<td>Features imposing an XOR group constraint among their children.</td>
</tr>
<tr>
<td></td>
<td>MUTEX groups</td>
<td>N/P</td>
<td>Features imposing a MUTEX group constraint among their children.</td>
</tr>
<tr>
<td></td>
<td>arbitrary group cardinality</td>
<td>N/P</td>
<td>Features imposing a proper ((m, n)) cardinality constraint among their children ((m &gt; 2, n \notin {1, \ast})).</td>
</tr>
<tr>
<td>Feature Constraints</td>
<td>features with constraint</td>
<td>N/P</td>
<td>Feature with explicitly declared (i.e. non-hierarchy) constraints, including configuration, visibility, and default constraints.</td>
</tr>
<tr>
<td></td>
<td>derived features</td>
<td>N/P</td>
<td>See above at feature roles. Unconditionally derived features are always computed, while conditionally derived features can either be set by the user, or are computed, based on an expression.</td>
</tr>
<tr>
<td></td>
<td>(unconditionally/conditionally)</td>
<td></td>
<td>Features declaring a condition that determines when they (and their children) are visible and user-changeable; otherwise, they are either invisible or grayed out.</td>
</tr>
<tr>
<td></td>
<td>features with visibility condition</td>
<td>N/P</td>
<td>Features declaring a condition that determines when they (and their children) are visible and user-changeable; otherwise, they are either invisible or grayed out.</td>
</tr>
<tr>
<td></td>
<td>features with default constraint</td>
<td>N/P</td>
<td>Features declaring an explicit default, which is either just a literal, or an expression.</td>
</tr>
<tr>
<td></td>
<td>(literal/computed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>features referenced</td>
<td>Dist. (N)</td>
<td>Number of features referenced in all constraints of a feature.</td>
</tr>
<tr>
<td></td>
<td>average features referenced</td>
<td>N/P</td>
<td>Mean of all dependencies per feature in a model.</td>
</tr>
<tr>
<td></td>
<td>Cross-Tree Constraints Ratio</td>
<td>P</td>
<td>Proportion of features participating in cross-tree constraints, that is, features that either have at least one dependency, or are the target of the dependency of another feature. This metric is adapted from [MWC09].</td>
</tr>
</tbody>
</table>

N: Numeric  P: Percentage  Dist.: Distribution

Table 5.6.: Model metrics used for the quantitative analysis
Software Ecosystems

This chapter shares content with the paper “Variability Mechanisms in Software Ecosystems: Closed versus Open Platforms” [BPT+].

We now broaden our perspective on variability from closed to open platforms and investigate organizational structures, variability mechanisms, and dependencies. To address research question RQ3, we perform five case studies of two closed and three open platforms. Our study is both qualitative and quantitative, the latter by analyzing significant subsets of the real ecosystems that our subject systems established. Our aim is to compare variability in both kinds of platforms, in order to draw conclusions about the applicability of variability models and the characteristics of variability in domains beyond systems software. This expansion of our research is a small, but self-contained step towards a theory on variability mechanisms in software ecosystems, which are seen as natural extensions of software product lines.

In this chapter, we first describe our methodology and introduce our subjects, we then describe the conceptual framework that emerged during exploration, and finally report the results of our study in the three Sections 6.3–6.5. Appendix B details many of our estimations. We will further analyze our results, for example to discuss the applicability of variability models, in the subsequent Chapter 7.

6.1. Methodology

The major part of our analysis is qualitative and exploratory. It aims at identifying mechanisms and organizational structures in the studied ecosystems and relationships among them. During the analysis, we iteratively built a conceptual framework of these mechanisms and structures, allowing us to compare their use across the ecosystems. The framework is summarized in Section 6.2 and in the concept hierarchy shown in the left-most column in the subsequent tables. We seeded the framework with mechanisms known from SPLE and then expanded to those specific to open ecosystems. Many are inspired from the previous language study (variability models, dependency types), but also from Czarnecki et al. [CE00] (binding time/mode, openness), and Szyperski et al. [Szy02] (interaction, encapsulation); others were added as discovered.
6.1.1. Subject Selection Criteria

Our selection of case study subjects ensures high representativeness of the resulting conceptual framework. We chose three successful ecosystems with open platforms, spanning diverse domains, and approaching variability in different ways. They range from a package management system specifying variability information in manifest files associated with packages, via a component-oriented architecture, to a highly dynamic service-oriented architecture with runtime resolution of dependencies. Together with our previous subjects eCos and Linux—feature-based systems with variability models and static compile-time binding—we cover a broad range of systems in our study.

eCos and the Linux kernel were already introduced in Sections 4.2.1, 4.2.2, and their architectures in Section 2.1.2. Both aim at technically skilled end users and developers, who derive instances via a configurator, and that their platforms are predominantly closed. They also—by design—offer no facilities to easily use third-party extensions. In eCos, although openness was a goal of its packaging mechanism, contributing still requires programming effort; while in Linux, additions must be applied to the source tree as patches or Git branches. “Out-of-tree” development is actively discouraged. In contrast, our new subjects Debian, Eclipse, and Android rely on platforms that were designed to be open, by offering abstractions and end-user facilities to take advantage of assets from a free market. Nevertheless, all five subjects are successful ecosystems, fostering inter-organizational reuse and spanning communities of developers.

Debian\(^1\) is a complete operating system with a large selection of applications. It is available for many hardware architectures, ranging from embedded systems to high performance computers. Its consumers are both non-technical end users and system administrators with high technical expertise. Debian provides suitable installers and configurators for beginners and experts.

The Eclipse IDE\(^2\) is a foundation for highly customizable development tools. Note that we do not consider the Eclipse RCP framework—suited to build arbitrary GUI software. Although users of the Eclipse IDE are technically skilled developers, extending the system is supported by a convenient installer.

Android\(^3\) is a free operating system for mobile devices, including smartphones, tablets and netbooks, that can be extended with third party applications (apps), most of which run in a virtual machine (Dalvik). The target consumers of Android are non-technical end users, deriving their system by installing apps with a user-friendly installer.

The first row of Table 6.1 summarizes the domain and target audience of all five ecosystems. Note that, even though Eclipse is a package in Debian, and Linux the underlying kernel of Android and Debian, we clearly distinguish these ecosystems, analyze and compare them on their own in our study. We do not explore their overlaps and interactions.

\(^1\)http://www.debian.org  
\(^2\)http://www.eclipse.org  
\(^3\)http://www.android.com
6.2. Conceptual Framework

6.1.2. Data Sources and Analysis Infrastructure

In the qualitative analysis, we rely on official documents such as the Debian Policy [debb] and the Eclipse Development Process description [Ec110]. All corresponding sources are cited as we use them in the text. We also examined tools and languages used in the subjects. For the quantitative measures, we use statically extracted data. Since analyzing whole ecosystems is infeasible given their open and uncontrolled nature, we mined substantial subsets by considering the most vibrant parts—the major distribution sources.

For eCos, we consider all i386-specific and hardware-independent packages from the repository (version 3.0). For Linux, we analyze the x86 architecture from the 2.6.32 codebase. Debian’s subset are all binary i386 packages from the main component of the 6.0 distribution. For Eclipse, we analyze the Helios 3.6 modeling distribution together with bundles from the associated repository. For Android, we gathered nearly all available free apps from the app store over a period of 14 months. The first row of Table 6.5 details the datasets underlying our quantitative analysis. These datasets are available online and further detailed in Appendix B.3.

We developed an analysis tool infrastructure that relies on ecosystem-specific extractors\(^4\). More precisely, the Debian script\(^5\) extracts package indices used by the native installers; the Eclipse extractor\(^6\) exploits the Eclipse platform API to query information about bundle manifest; and the Android extraction\(^7\) used a third-party library to query and download apps from the Google Play store. Since the analysis of Android apps was most challenging, we had to develop a static analysis infrastructure\(^8\) for Android bytecode to identify dependencies. Details are provided in Appendix B.4. For eCos and the Linux kernel, our infrastructure relies on extracted data using CDLTools and LVAT (Appendix A).

6.2. Conceptual Framework

Similar to our qualitative analysis of variability modeling language concepts in Chapter 4, we first introduce our conceptual framework that emerged from and was iteratively refined during our study. We will use it in the remainder by instantiating it with empirical data from our subjects.

6.2.1. Software Ecosystem

As pointed out in Section 2.4, research has not settled on a technology-oriented definition of ecosystems. Based on the empirical results of our exploratory study, we define a

\(^{4}\)These extractors were specified by us, but developed by students—co-authors of the corresponding paper [BPT+].

\(^{5}\)Reinhard Tartler

\(^{6}\)Rolf-Helge Pfeiffer

\(^{7}\)Steffen Dienst

\(^{8}\)Also developed by Steffen Dienst and supervised in the context of this dissertation.
software ecosystem as a universe of shared assets centered around a technical platform. In these ecosystems, various roles, mainly suppliers and consumers, interact in order to develop, manage, and consume assets. A platform denotes the technical aspects of an ecosystem: a variability-enabled architecture, a set of shared core assets, tools, frameworks, and patterns together with organizational and process-related concerns.

Every vital ecosystem has a controlled central part, the main platform, which is managed by the platform supplier. The free market is the less-controlled, complementary part of the ecosystem, and provides third-party assets extending the main platform. Alternative platforms may exist as derivatives of the main platform for specific needs. For example, Ubuntu is derived from the Debian main platform for desktop and laptop users. Since derivatives do not belong to the free market, we ignore them in this study.

6.2.2. Variability Representation

Assets are any artifacts, such as source code, binaries, media files, or documentation. Each of the studied platforms packages assets into basic units, such as Debian packages or Eclipse bundles. Composite units, such as Debian meta packages, aggregate sets of basic units.

Variability in the platforms has two forms: basic units can be optional, or vary inside, or both. Unit parameters, such as properties in Eclipse, describe variability within basic units.

Variability information (dependencies and unit parameters) is specified either within a variability model or in distributed manifests. Recall that variability models are system-wide abstractions and that features as abstract entities are mapped to units and unit parameters. Instead of making decisions directly on the assets, derivation is based on deciding features. Manifests directly reflect variability information of the assets, without the ability to introduce abstractions, for example, to optimize dependency structures.

6.2.3. Instance Derivation

An instance (e.g. a customized Linux kernel or Android system) is a concrete system derived from the main platform and the free market by making decisions—more precisely, by selecting and configuring assets, thus, resolving variability. Usually, an instance can be reconfigured later.

Each ecosystem supports derivation and reconfiguration by automated tools: configurators for the variability model-based platforms (eCos, Linux kernel) and installers for manifest-based platforms (Debian, Eclipse, Android). Such automated tools assist users with intelligent choice propagation, conflict resolution, and optimization based on the dependencies. The latter are declared either among features within the variability model, or among basic or composite units within the manifest.
6.3. Organization and Scale

In this section, we describe the organizational structures we identified in each ecosystem, and the scales they achieved over time. Investigating these structures aims at understanding what parts of an ecosystem are controlled, and to what extent distributed development or variability management is feasible.

6.3.1. Organization

We now briefly characterize the main platform and the free market of each ecosystem; and how development and variability management are organized therein. The second row in Table 6.1 summarizes our observations with respect to the conceptual framework.

**eCos**’ main platform is its free edition, maintained and developed by the primary supplier eCosCentric and external contributors [Mas03]. A commercially developed derivative, eCosPro, serves the market of independent solution vendors. The main platform is controlled by a group of (currently) nine volunteer maintainers. They control the access to the source repository and the associated infrastructure and review and integrate contributions from the community, prioritizing registered projects. Both development and variability management are **centralized** in the main platform. Although eCos’ packaging mechanism was designed to encourage contributions, only a marginal free market emerged on the fringe of the main platform, with relatively few free and commercial packages. No uniform distribution channel exists for free market contributions. However, a number of external contributions are listed on the eCos website, most of

---

**http://ecos.sourceforge.org/ contrib.html**
which extend the variability model of the core platform by implementing eCos packages. Even porting to new architectures can be realized as extensions to the main platform.

**Linux’** main platform is the mainline kernel—the primary branch in the official git repository. The kernel codebase is subdivided into more than 100 subsystem trees, each related to a specific part of the kernel—such as SCSI drivers or x86 architecture code—and controlled by a maintainer. The development is highly *distributed*, comprising thousands of developers and maintainers. Any contributor can post patches—source changes—to the kernel mailing list, which are then thoroughly discussed, reviewed, and eventually integrated into a git development branch. This process allows to integrate fragile and cross-cutting changes to the kernel. The final integration into the mainline kernel is ultimately decided by Linus Torvalds, the inventor, and few maintainers called “lieutenants”.

In contrast to the development, we learned that variability management is *centralized*, with only a few maintainers controlling the variability model. All contributed features are eventually integrated into a single hierarchy. Notably, the maintainers spend substantial time fixing dependency inconsistencies when integrating new features into the model, as we observed in another study [LSB+10]. Although no uniform distribution channel (beyond mailing lists) outside the main platform exists, an unorganized free market with third-party modules, comprising mostly drivers, emerged.

**Debian’s** main platform is the central repository containing the official distribution. Both development and variability management are *distributed*, comprising over thousand registered package maintainers, who maintain ready-to-install packages that are sourced from free and open source software [Kra05]. In particular, they maintain variability information specified in the package manifests, like the one shown in Fig. 2.11 in Section 2.4. The main platform tries to be as inclusive as possible, with little restrictions (besides legal and license issues) to contributors, while reviews of contributions still assure quality. Maintainers strictly adhere to the Debian constitution [deba], and the Debian policy [debb]; both regulate governance, rights, and roles inside the main platform.

Complementing the main platform, a free market with mostly commercial and non-free packages exist, such as Google Chrome or the Adobe Flash Player. However, this free market is scattered over many third-party repositories, no uniform distribution channel exists.

**Eclipse’s** main platform is represented by the official yearly releases of the IDE. The main platform consists of independently managed projects following the Eclipse Development Process (EDP) [Ecl10] and is controlled by its supplier, the Eclipse Foundation—a nonprofit consortium of industry members providing full-time developers. Contributions are encouraged, for example by contributing new components (bundles, see Section 6.4.1.1), or even establishing a new project. However, contributions undergo thorough reviews,

---

10The **APT-GET.org** website, for example, lists many third-party repositories of Debian packages.
and projects face a formal and lengthy approval process, as specified in the EDP. Given this organization, both the development and the variability management is distributed in the main platform.

Eclipse has a complementary free market, mainly represented by the Eclipse Marketplace\textsuperscript{11} and further repositories, such as Yoxos\textsuperscript{12} (a free service offering custom Eclipse instances), or smaller update sites that can be used by the Eclipse installer.

Android’s main platform comprises the operating system and pre-installed apps. While the development is distributed, the variability management is centralized and fully controlled by Android’s supplier, the Google-led Open Handset Alliance. The source code has been released as the Android Open Source Project\textsuperscript{13} in 2008. It is divided into individual sub-projects, each having a project lead—typically a Google employee [andb]. Contributions to the main platform are possible, but must pass thorough reviews by so-called approvers, experienced project members. Most of the main platform apps are commercially developed by Google, however.

A vibrant free market is an essential goal of Android, with the main distribution channel, the Google Play store, being wide open to third-party contributions of arbitrary applications. Apps on the Play Store are not centrally tested or validated, the quality is entirely in liability of its contributor. This results in a broad range of quality, from the main platform apps provided by Google that need to pass rigorous internal tests, to trivialities like “hello-world” apps or even malware.

6.3.2. Scale and Growth

Our subjects differ considerably in scale, both in the total size—ranging from 1.2M LOC to over 1.2G, and in the proportions between main platform and free market. For each ecosystem, we conservatively estimated (lower bounds) main platform and free market sizes, which are shown in Table 6.2, and detailed in Appendix B.1. Based on exact numbers for the initial, and lower bounds of the current scales, we estimated growth rates. We use these numbers to characterize orders of magnitude of the ecosystems, but carefully avoid any further conclusions from them.

6.3.2.1. Size of Main Platform and Free Market

eCos has the smallest main platform, comprising 502 packages—aggregating 4,000 files and nearly one million LOC in its repository. Its free market is marginal, with an estimated lower bound of only 14 third-party packages of free and commercial extensions.

Linux is much larger, not surprisingly given that it supports a much wider variety of hardware. Linux’ main platform (v. 2.6.32) comprises almost 26,000 files and around 7,000 kernel modules. Unfortunately, we could not estimate the size of the possibly large, but unorganized free market. However, the free market is potentially huge, due to almost

\textsuperscript{11}http://marketplace.eclipse.org
\textsuperscript{12}http://ondemand.yoxos.com
\textsuperscript{13}http://source.android.com

115
### 6. Software Ecosystems

Table 6.2.: Estimated scales and growth rates of ecosystems (as of 03/2012)

<table>
<thead>
<tr>
<th></th>
<th>eCos</th>
<th>Linux</th>
<th>Debian</th>
<th>Eclipse</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main platform scale</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Units</td>
<td>3948(^1)</td>
<td>25,861(^1)</td>
<td>28,232(^2)</td>
<td>5,787(^3)</td>
<td>83(^4)</td>
</tr>
<tr>
<td>Features</td>
<td>2,859</td>
<td>10,415</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LOC</td>
<td>0.9M</td>
<td>7.9M</td>
<td>762M</td>
<td>21.2M</td>
<td>1M</td>
</tr>
<tr>
<td><strong>Free market scale</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Units</td>
<td>&gt;1,530(^5)</td>
<td>—</td>
<td>&gt;15,179(^5)</td>
<td>&gt;1,897(^5)</td>
<td>&gt;403K(^4)</td>
</tr>
<tr>
<td>Features</td>
<td>&gt;315</td>
<td>—</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LOC</td>
<td>&gt;279K</td>
<td>—</td>
<td>&gt;410M</td>
<td>&gt;6.9M</td>
<td>&gt;620M</td>
</tr>
<tr>
<td><strong>Growth rates</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inception year</td>
<td>1999(^v1.1)</td>
<td>1991(^v0.01)</td>
<td>1996(^v1.1)</td>
<td>2001(^v1.0)</td>
<td>2008</td>
</tr>
<tr>
<td>Inception LOC</td>
<td>76k</td>
<td>10k</td>
<td>13M</td>
<td>141k</td>
<td>1.128M</td>
</tr>
<tr>
<td>Current LOC</td>
<td>1.2M</td>
<td>7.9M</td>
<td>1.2G</td>
<td>28.1M</td>
<td>621M</td>
</tr>
<tr>
<td>Growth per year</td>
<td>32%</td>
<td>39%</td>
<td>35%</td>
<td>80%</td>
<td>507%</td>
</tr>
</tbody>
</table>

1 Files  2 Packages  3 Bundles  4 Apps  5 Android OS and apps

700 Linux distributions in existence today\(^14\). Many of these Linux distributions have their own patches with kernel additions and customizations.

Debian has the most inclusive and largest main platform in our study, given the comparatively little restrictions to contribute new packages. Crawling all third-party repositories listed on the APT-GET.org website\(^15\) shows that the free market is relatively small, consisting of third party repositories that altogether have half the number of packages of the main platform.

Eclipse’s main platform and free market are both of medium size, compared to the others. The main platform Helios 3.6—consisting of 1,097 source repositories\(^16\) with estimated 5,787 bundles—is three times larger than the two free market repositories Eclipse Marketplace and Yoxos, which approximate to a lower bound of 1,897 bundles. However, the free market might be significantly larger, since the ecosystem is scattered, and bundles are available on many other third-party repositories called update sites.

Finally, Android is an ecosystem with a free market that is over 600 times larger than the main platform\(^17\). Two third of the free market apps are freeware, the reset commercial. Compared to the free market, Android’s main platform, which is relatively closed and with a strong filtering of outside contributions, is marginal. The Google Nexus S mobile device contains only 83 apps (Android v2.3.4).

6.3.2.2. Growth Rates

We estimated yearly growth rates of our subjects by fitting an exponential growth function to the size difference between initial release and current state. The third row of Table 6.2 shows our estimates, further detailed in Appendix B.1.2. Not surprisingly, these indicate that platforms with uniform distributions and intended free markets (Eclipse, Android) grew considerably faster than those with a focus on the main platform (eCos, Linux, Debian). Debian has a yearly growth rate just between eCos and Linux, despite having a low-entry barrier to the main platform. Android has been growing at the unprecedented rate of 507% per year.

6.4. Variability Mechanisms

In our study, we identified and characterized variability mechanisms both from a technical—how instances vary—and a consumer perspective—how and when consumers make decisions. Table 6.3 summarizes our observations.

6.4.1. Variability Representation

To characterize how variability is represented, we first identify those parts that vary among instances, using our abstraction of units, unit parameters, and composite units, as described in our conceptual framework.

6.4.1.1. Units, Unit Parameters, and Composite Units.

In eCos, basic units are source files with internal variability controlled by preprocessor symbols (unit parameters) and realized via \#ifdef statements. Composite units are eCos packages, which are aggregations of source files, test cases, or other resources, together with a variability model of the package. Recall that eCos’ configurator aggregates partial models into a single whole, depending on the set of loaded packages (cf. Section 4.2.1).

Linux has two types of basic units: (1) source files with preprocessor symbols (unit parameters) as in eCos, and (2) loadable kernel modules that extend Linux at runtime. No concept for composite units exists.

Debian’s basic units are packages—file archives with helper scripts and a manifest, such as the one shown in Fig. 2.11 (Section 2.4.1). Composite units are realized by meta packages, whose purpose is to aggregate other packages via dependencies. The tool debconf\(^{18}\) realizes unit parameters and is used by scripts to configure the packaged software. It prompts users to make configuration choices during package installation.

Eclipse’s basic units are OSGI [OSG09] bundles—dynamically loadable modules tying together artifacts such as Java classes, images, configuration files, and metadata. Bundles run in a virtual machine. Unit parameters are provided by several mechanisms, including the preference store and configuration admin service. Composite units, called “features”, aggregate multiple bundles with branding and update information.

\(^{18}\)http://joeyh.name/code/debconf
Android is composed of apps—individual application programs representing basic units. Most apps run in a virtual machine (Dalvik). Android has no concept of composite units, and no dedicated mechanism for unit parameters. Apps read global settings from a special class or a data storage.

6.4.1.2. Variability Specification

Debian, Eclipse, and Android declare variability information in textual or XML-based manifest files, which contain meta information, mainly identity, version, dependencies, and encapsulation specifications. eCos has a rudimentary manifest per package, containing only identity data and a textual description. The main information resides within a partial variability model describing the variability of a package. Finally, the Linux kernel lacks any packaging mechanisms, thus, relies completely on a centralized variability model.

A main difference between variability modeling is the ability to optimize dependencies, due to the abstracted mapping to implementation assets. In the quantitative analysis, we will expand on dependency structures.

6.4.1.3. Grouping and Categorization

Units and features need to be organized in some form. eCos and Linux organize features hierarchically in the variability models, whereas units are organized in diverse, often informal, ways in the open platforms. Categorization facilities are integrated in the Eclipse Marketplace and Google Play. Debian goes further and offers community-driven categorizations using Deftags [Zin05].

6.4.2. Decisions

We now look closer into derivation and reconfiguration decisions. The most distinguishing characteristics of decisions we identified are their lifecycle, binding, and tool support.

Decision lifecycle. A single decision establishes presence or absence of a basic unit in an instance. A decision lifecycle characterizes when and how end users decide the presence or absence of units—whether they derive an instance from scratch, or only reconfigure one. In Linux and eCos, users derive an instance using configurators. In the other ecosystems, end users normally reconfigure an initial instance provided by the supplier. Eclipse comes in one of eleven pre-instantiated editions. An Android instance is delivered with the mobile device. A Debian end-user usually installs a minimal system before it can be reconfigured by installing and removing packages.
<table>
<thead>
<tr>
<th>Variability representation</th>
<th>eCos</th>
<th>Linux Kernel</th>
<th>Debian</th>
<th>Eclipse</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability model</td>
<td>feature-model-like</td>
<td>feature-model-like</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Features</td>
<td>packages, components, options, interfaces</td>
<td>configs, choices, menuconfigs, menus</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Language</td>
<td>CDL</td>
<td>Kconfig</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Manifest (Schema)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Asset Base</td>
<td>files</td>
<td>files, kernel modules</td>
<td>packages</td>
<td>bundles</td>
<td>apps</td>
</tr>
<tr>
<td>Basic units</td>
<td>packages</td>
<td>N/A</td>
<td>meta packages</td>
<td>features</td>
<td>N/A</td>
</tr>
<tr>
<td>Composite units</td>
<td>preproc, symbols</td>
<td>preproc, symbols</td>
<td>debian options</td>
<td>properties/ preferences</td>
<td>N/A</td>
</tr>
<tr>
<td>Unit parameters</td>
<td>variability model</td>
<td>variability model</td>
<td>tasks, sections, debugs</td>
<td>market place</td>
<td>app store categories</td>
</tr>
<tr>
<td>Grouping and categorization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decision lifecycle</td>
<td>derivation</td>
<td>derivation, reconfig.</td>
<td>reconfiguration</td>
<td>reconfiguration</td>
<td>reconfiguration</td>
</tr>
<tr>
<td>Decision binding</td>
<td>static</td>
<td>static &amp; dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>Tools</td>
<td>configurator (config-tool), build system</td>
<td>configurator (Kconfig), build system (Kbuild)</td>
<td>package managers</td>
<td>installer, market place (client (p2))</td>
<td>installer app (e.g. Google Play)</td>
</tr>
<tr>
<td>Interface mechanisms</td>
<td>C header files</td>
<td>C header files</td>
<td>package-specific</td>
<td>Java interfaces and OSGI manifest</td>
<td>expl. public components, predef. data formats</td>
</tr>
<tr>
<td>Interface specification</td>
<td>documented interfaces for components, e.g., drivers</td>
<td>documented interfaces for components, e.g., drivers</td>
<td>package-specific, documented policies for some domains</td>
<td>explicit public interfaces defined by OSGI manifest</td>
<td>expl. public components, predef. data formats</td>
</tr>
<tr>
<td>Managed by runtime system</td>
<td>N/A</td>
<td>static &amp; dynamic linking</td>
<td>dpkg-triggers, documented policies</td>
<td>Equinox OSGI</td>
<td>Dalvik VM</td>
</tr>
<tr>
<td>Interaction mechanisms</td>
<td>early static</td>
<td>early static &amp; dynamic</td>
<td>not specified</td>
<td>late static &amp; dynamic</td>
<td>late dynamic</td>
</tr>
<tr>
<td>Interaction binding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. Software Ecosystems

**Decision binding.** Decisions can have different binding mode and binding time. Binding mode characterizes whether a decision can be changed. For eCos and Linux, it is static, since these systems require to re-derive the instance for changes. However, Linux also allows late dynamic decision binding by means of loadable kernel modules. Debian, Eclipse and Android are dynamic as they allow basic units or composite units to be installed and removed at run-time.

**Tools.** In contrast to the configurators used for static configuration in our closed platforms, the open platforms include an installer that allows end users to extend their instance. Installers are characteristic for reconfiguration processes, where units are usually downloaded in order to be installed or updated on the running system.

### 6.4.3. Encapsulation

Our closed platforms offer no encapsulation concepts beyond C header files; only implementation guidelines for interfaces of loadable kernel modules exist in Linux. In Debian, interfaces are solely package-specific; however, Debian has policies for some domains, such as Java libraries or Emacs extensions. Eclipse encapsulates all classes and resources in the bundle; public functionality—Java packages, OSGi service interfaces, extension points—must be declared in the manifest. Android apps can provide public components that are described and advertised to other apps with intent filters (explained shortly in Section 6.5.1.1).

### 6.4.4. Interactions

Interactions among basic units requires identifying and binding the concrete target. Our two closed platforms use static interaction binding, as technically, all selected basic units are linked into a single binary image. Linux additionally supports late dynamic interaction binding (using loadable kernel modules). Although interaction binding is mostly package-specific in Debian, several policy documents prescribe guidelines for interaction in some specific package domains.

As a major difference, the open platforms Eclipse and Android both provide a runtime system with full control over interactions. Eclipse offers three facilities: direct class referencing, extension points and services. Except for services, using the Service Activation Toolkit or declarative services, interaction targets are bound late but statically—due to Java classloader restrictions. Android provides a purely dynamic facility for interaction with its intent mechanism. The interaction target—specified by parameters of an intent—is continuously reevaluated at runtime and could easily change when apps are exchanged or reinstalled.
### 6.4. Variability Mechanisms

#### Table 6.4.: Dependency mechanisms

<table>
<thead>
<tr>
<th>Direct dependency</th>
<th>eCos</th>
<th>Linux Kernel</th>
<th>Debian</th>
<th>Eclipse</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>features</td>
<td>features</td>
<td>basic units</td>
<td>basic units</td>
<td>basic units</td>
</tr>
<tr>
<td>Types (hard/soft)</td>
<td>hierarchy, requires, active_if, default, calculated</td>
<td>select, prompt condition, default</td>
<td>depends, pre-depends, recommends, breaks, conflicts, suggests, enhances</td>
<td>Require-Bundle</td>
<td>explicit intent</td>
</tr>
<tr>
<td>Capability-based dependency</td>
<td>CDL interfaces</td>
<td>N/A</td>
<td>virtual packages</td>
<td>Java packages</td>
<td>intent filters</td>
</tr>
<tr>
<td>Target</td>
<td>same as direct dep.</td>
<td>N/A</td>
<td>same as direct dep.</td>
<td>Import-Package</td>
<td>implicit intent</td>
</tr>
<tr>
<td>Types</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>via API</td>
<td>via API</td>
</tr>
<tr>
<td>Common vocabulary</td>
<td>implements</td>
<td>N/A</td>
<td>provides</td>
<td>Export-Package</td>
<td>via intent filter</td>
</tr>
<tr>
<td>Provide capabilities</td>
<td>any Boolean, arithmetic &amp; string operators</td>
<td>any Boolean operators, and number/string equality</td>
<td>any Boolean operator, and version comparison</td>
<td>conjunction, implication, and version comparison</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.4. Variability Mechanisms
6.5. Dependencies

In our study, we identified the following mechanisms to express dependencies and resulting dependency structures. Table 6.4 summarizes the core characteristics.

6.5.1. Specification, Semantics & Expressiveness

Our ecosystems approach specifying dependencies in diverse ways. We now report the main characteristics we identified among our subjects.

6.5.1.1. Specification

eCos and Linux declare dependencies among features in their variability models. Due to their high level of abstraction, variability models allow flexible specification of intricate dependency structures. This flexibility comes at the cost of maintaining additional artifacts, such as the variability model (cf. our evolution study [LSB+10]) and feature-to-code mapping, which need to be carefully coordinated. Debian’s and Eclipse’s specification of dependencies among basic units in manifests is more direct, but less flexible. In contrast, Android approaches the problem entirely dynamically. No static specifications of dependencies among apps are used. Apps can only declare to be open for interaction by setting a flag, or defining an intent filter, stating that the app can handle specific service requests. Android’s installer does not enforce dependencies statically, instead, apps handle unsatisfied dependencies dynamically at runtime. Appendix B.4.1 provides further details on Android’s intent mechanisms.

We identified a special class of dependencies occurring in each ecosystem: dependencies on capabilities, as opposed to direct dependencies. Capabilities are abstractions over functionality provided by one or more units or features. For example, the capability to open URLs is provided by multiple web browsers. In Fig. 6.1, we detail the roles assumed by units and capabilities in dependencies: providing and depending on other units and capabilities. Some platforms provide explicit capability constructs, such as CDL interfaces in eCos and virtual packages in Debian (line 5 in Fig. 2.11, Section 2.4.1). Eclipse uses names of Java packages as capabilities. Android provides the richest specification via intent filters. These form a simple domain-specific language, or an ontology, which can be used by contributors to increase reuse. Interestingly, the community recognized that standardized vocabulary fosters app interactions. OpenIntents\(^{19}\) is a registry that provides additional vocabulary for intents and intent filters, contributed by developers. Finally, Kconfig has no explicit capability construct, but some features in the Linux model play this role, as identified in Section 5.3.2.1.

6.5.1.2. Dependency Metamodel

Based on our qualitative observations, we define a metamodel of dependencies that abstracts over concrete dependency types in our subjects. It fosters our understanding of

\(^{19}\)http://www.openintents.org
dependencies, but, more importantly, allows a quantitative analysis and comparison of dependency structures in ecosystems.

Figure 6.1: Dependency metamodel (in labels, unit(s) to be replaced with feature(s) where applicable)

In the metamodel, associations represent dependencies, and association roles their direction; more precisely:

- **Forward dependencies**
  1. *Units a unit depends on* is probably the most common type of dependencies; a unit/feature directly requires the referenced units/features to function.
  2. *Capabilities a unit depends on* are dependencies from units/features to abstract capabilities (instead of units/features directly). Capabilities itself have a relationship to units/features (see 5, 6).

- **Reverse dependencies**
  3. *Units depending on a unit* represents those units/features that directly depend on a given unit/feature. It shows how many other units/features use or require functionality of a certain unit/feature.
  4. *Units depending on a capability* represents those units/features that depend on a given capability. It shows how many units/features use or require functionality represented by a certain capability.

- **Relationship between Units/Features and Capabilities**
  5. *Capabilities provided by a unit* is a relationship determining those capabilities that a certain unit/feature provides. Recall that capabilities are: special kinds of features (CDL interface) in eCos, labels (virtual package/Java package) in Debian/Eclipse, and rich constructs (intent filters) in Android.
  6. *Units providing a capability* represents the reverse direction, that is, those units/features that provide a certain capability.

### 6.5.1.3. Semantics

We also classified the dependencies by their semantics (modality). Hard dependencies must always be satisfied. Soft dependencies represent suggestions or defaults. Recall that
### 6. Software Ecosystems

#### 6.5. Dependency Structures

To study dependency structures, we computed cardinalities for all association ends in our dependency metamodel (Fig. 6.1). In the following, we report the key observations and refer to Appendix B.5 for detailed diagrams.

---

**Table 6.5.: Dependency statistics (as of 03/2012)**

<table>
<thead>
<tr>
<th>Ecosystem subset</th>
<th>eCos</th>
<th>Linux</th>
<th>Debian</th>
<th>Eclipse</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic units</td>
<td>1023(^1)</td>
<td>10,326(^1)</td>
<td>28,232(^3)</td>
<td>2,105(^4)</td>
<td>281,079(^5)</td>
</tr>
<tr>
<td>Features</td>
<td>1,244</td>
<td>6,308</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LOC</td>
<td>302K</td>
<td>4.3M</td>
<td>782M</td>
<td>7.8M</td>
<td>433M</td>
</tr>
<tr>
<td>LOC per basic unit(^1)</td>
<td>295</td>
<td>416</td>
<td>27,699</td>
<td>3,709</td>
<td>1,539</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic units/features</th>
<th>W/ dependencies</th>
<th>direct(^1)</th>
<th>99%</th>
<th>100%</th>
<th>N/A</th>
<th>N/A</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>to capability(^1)</td>
<td>8%</td>
<td>N/A</td>
<td>24%</td>
<td>27%</td>
<td>68%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W/ depending units(^2)</td>
<td>42%</td>
<td>31%</td>
<td>62%</td>
<td>57%</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Providing capability(^1)</td>
<td>10%</td>
<td>N/A</td>
<td>13%</td>
<td>80%</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies (^1)(^1)</th>
<th># per basic unit/feature(^2)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Capabilities</th>
<th>W/ depending units(^2)</th>
<th>44%</th>
<th>N/A</th>
<th>54%</th>
<th>11%</th>
<th>N/A</th>
</tr>
</thead>
</table>

\(^1\) Files  \(^2\) Loadable modules  \(^3\) Packages  \(^4\) Bundles  \(^5\) Apps
\(^\dagger\) Average  \(^\ddagger\) Median  \(\circ\) Numbers refer to our meta model (Fig. 6.1).

---

we even observed conditionally hard or soft dependencies (defaults) in Kconfig, which assume a different modality depending on a side condition. Table 6.4 shows the keywords in the variability languages/schemas declaring a certain type of dependency.

#### 6.5.1.4. Expressiveness

The constraint languages for declaring dependencies differ in expressiveness. In contrast to the rich languages CDL and Kconfig, the manifest schemas in our open platforms are less expressive. Debian supports any Boolean dependencies among packages and comparisons on version ranges. Exclusions are specified via conflicts and breaks, and defaults via recommends. Debian provides even more modalities, mainly to drive package update, replacement, and removal processes. Eclipse supports implications, conjunctions, and version comparisons, but lacks negations and disjunctions. It is not easily possible to exclude bundles or declare alternatives\(^20\).

#### 6.5.2. Dependency Structures

---

\(^20\) Except when using the p2 system [LBR09] on top.
6.5. Dependencies

6.5.2.1. Connectivity

The connectivity of the dependency graph indicates the proportion of units and features for which dependency information has to be maintained. The number of units or features having direct (1) in Fig. 6.1) and capability-based (3) dependencies is surprisingly high, regardless of platform openness. The highest is observed in Linux, where almost all of features reference others, and in eCos, where it reaches 99%. These numbers are high, partly because every non-root feature implies its parent in the model hierarchy. Still, many features (30% in eCos, 85% in Linux) declare cross-hierarchy dependencies, which are known to critically influence hardness of reasoning—both for configuration tools [MWC09] and for users, by introducing intricate implications of choices. Finally, in the open systems, most basic units also participate in many dependencies: Debian has the highest amount with 96%, followed by Eclipse with 89%, and Android with 69%.

6.5.2.2. Density

The density of the dependency graph indicates how much dependency information needs to be maintained per unit or feature. To assess it, we considered the number of dependencies per unit or feature. Their distributions are shown in the boxplot in Fig. 6.2.

![Figure 6.2: Dependencies per feature or basic unit (x-axis cut off)](chart.png)

Except Android, the open platforms have more dependencies per unit than the others per feature. Interestingly, there are many outliers, such as an app with 96 dependencies in Android, a package with 323 dependencies in Debian, and a bundle with 419 dependencies in Eclipse. Some Debian outliers have many soft dependencies (modalities like suggests and recommends), although most dependencies are hard in Debian (cf. Appendix B.5.2.2). While many Eclipse outliers are caused by many Java package imports (capability-based dependencies), most dependencies are direct ones on bundles (cf. Appendix B.5.2.1).

We also investigated the reverse dependencies (2 and 3 in Fig. 6.1). If units have many, they are particularly hard to evolve, since dependencies on them are not specified directly together with the unit. Evolution of such units can break dependencies easily. We obtained numbers for all systems except Android, due to limitations of our static analysis, as we point out in Appendix B.4.3.2. We find that the open ecosystems have higher proportions of units being referenced (Debian: 62%, Eclipse: 57%) than the others.
6. Software Ecosystems

for features (eCos: 42%, Linux: 31%). We further notice that, particularly in Debian, 44% of packages depend on \texttt{libc6}, whereas in the other subjects, we could not identify such an outstanding central unit or feature.

6.5.2.3. Capabilities

Interestingly, the percentage of units or features with direct dependencies drops significantly from eCos with 99% to Android with only 14%. The opposite is observed for capability-based dependencies, which rise from 8% in eCos to 68% in Android. Dependencies on capabilities increase variability (more than one web browser can fulfill the open URL capability), decrease coupling (an app no longer depends on a specific browser), improve flexibility and communication among developers, since capabilities indicate that specific functionality is available.

6.6. Conclusions

Earlier in Section 2.4.1, we formulated some conjectures about variability in open platforms. Our qualitative analysis shows that the mechanisms in open platforms differ with respect to asset packaging, encapsulation, and parameterization.

\textbf{Variability Mechanisms.} Mechanisms in closed platforms are characterized by fine-grained basic units, expressive dependency facilities, and early static decision binding. In open platforms, we expectedly found easy-to-use mechanisms that promote contributions: uniform distribution channels within a free market, asset packaging, runtime resolution of dependencies, highly dynamic runtimes, and interface mechanisms. Furthermore, the fast-growing ecosystems (≥80% a year) rely on dynamic decision binding and service-oriented composition mechanisms, like runtime-service lookup, download and installation.

While our closed platforms allow almost arbitrary changes by each third-party contribution, they need heavyweight processes and strict governance to assure a certain quality. Our open platforms with free markets are characterized by coarse-grained variability and by restricting the possible impact of contributions using asset packaging and encapsulation with defined interface mechanisms. Interestingly, Debian is in between, with more control over contributions by defining review processes in the Debian Policy document, but without strict interface mechanisms and relying on a virtual machine\textsuperscript{21} that controls interactions between packages, as Eclipse and Android do.

\textbf{Capabilities.} Our discovery of capability-based dependencies shows that these are essential for open platforms. Capabilities are used (1) to improve flexibility and reduce coupling, since targets can be exchanged easily, and (2) to support communication among developers, since capabilities indicate that specific functionality is available—likely through an API. Although capabilities reduce coupling, thus, foster distribution, their use relies on a centralized and stable vocabulary.

\textsuperscript{21}Which, of course, would be impossible, since Debian packages contain native applications.
6.6. Conclusions

**Organization.** The ratios between main platform and free market scales differ significantly, and in effect, the parts of each ecosystem that are controlled by the supplier. There is no clear correlation between the openness of a platform and the focus (on main platform or free market) of an ecosystem: Linux and Debian focus on the main platform, Eclipse and Android on the free market. eCos tried to establish a free market, but failed and has the majority of its packages within the main platform. Interestingly, even though our fast-growing and dynamic ecosystems foster an uncontrolled free market, they strictly control the main platform. In any case, a vibrant free market requires uniform distribution channels like Eclipse’s Marketplace or Android’s Play Store.

**Dependencies.** The proportion of basic units and features with dependencies is very high, regardless of the domain and the openness of the platform. For Android, which not declares dependencies, we show that static analysis can extract such dependencies from Dalvik bytecode with an estimated recall of 60% (Appendix B.4.3.2).

In general, we observe differences between closed and open platform in the facilities to express dependencies, and in the actual dependency structures. We will expand on this observation in the next chapter.
We now interpret our study results and discuss their impact on the research field of software product lines and software ecosystems. This analysis addresses our last research question RQ4. For practitioners, we also compile a set of guidelines on variability modeling, referring to the corresponding parts of our dissertation for further details. Finally, we give an outlook on our currently ongoing study on industrial variability modeling and provide a selection of preliminary results.

### 7.1. Towards a Theory

While we presented the core research results of our dissertation in the last three chapters, the exploratory style of significant parts of this work provides a good basis for constructing a theory. But although we studied a well-defined, broad range of systems that span diverse domains and leverage different variability mechanisms, it would be inappropriate to claim statistically significant conclusions by generalizing our observations. Thus, we develop testable hypotheses that explain core observations we made in our research.

In the following, we first provide a high-level overview on the conceptual framework we iteratively developed during our qualitative analyses. We then expand on major observations and correlations, depict our hypotheses, and formulate interesting future research questions indicated by our data.

#### 7.1.1. Conceptual Framework: Putting it All Together

During the whole course of our analysis of variability modeling languages, models, and ecosystems, we iteratively developed, refined, and instantiated our conceptual framework based on empirical data. It also faced intensive discussions with the co-authors of the corresponding publications, to reach consensus about the concepts. The framework covers both product lines and ecosystems, and relates technical variability modeling concepts to organizational structures. It needs to be extended, refined, or even refuted with follow-up research in further domains.

The qualitatively developed framework aims at two goals: i) introduce abstractions over the system-specific terms to enable comparison, and ii) provide units of analysis for the quantitative studies. It combines all identified concepts from Chapters 4, 5, and 6. Table 7.1 summarizes the technical concepts of variability, and references corresponding
sections and tables. Fig. 7.1 illustrates the high-level concepts of the framework, with a focus on the organizational structure, and actors participating in an ecosystem.

Recall that the focus of our work lies in the variability modeling languages and models study in Chapters 4 and 5. It provided the basis for our exploratory analysis of open platforms in Chapter 6, whose main benefit is to broaden our perspective on variability to open platforms and modern, dynamic variability mechanisms. Our discourse from closed to open platforms, and our investigation of organizational structures led to this high-level framework.

Figure 7.1: Conceptual framework: overview of ecosystem organization and variability mechanisms
### 7.1. Towards a Theory

Table 7.1.: Conceptual framework: overview

<table>
<thead>
<tr>
<th>category</th>
<th>concepts</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>variability models</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(feature-model-like)</td>
<td><em>language concepts</em></td>
<td>Overview in Section 4.3 and Table 4.1</td>
</tr>
<tr>
<td></td>
<td>feature kinds, representation,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hierarchy, grouping, constraints, and others</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>language semantics</em></td>
<td>Section 4.4.8 and Section 4.5.8, introduction in Section 2.3.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>model concepts</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>feature themes</td>
<td>Section 5.3.1, Table 5.2</td>
</tr>
<tr>
<td></td>
<td>hierarchy structures</td>
<td>Section 5.4.1</td>
</tr>
<tr>
<td></td>
<td>model metrics</td>
<td>Overview in Table 5.6; values in Tables 5.3, 5.4, and 5.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mapping</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>feature-to-code mapping</td>
<td>Sections 4.3.6, 4.4.6, 4.5.6; static analysis of build code in Section 4.5.6.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>manifests</strong></td>
<td>metadata and dependency types</td>
<td>Section 6.4.1.2, Table 6.3, manifest dependency types in Table 6.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Asset Base</strong></td>
<td>basic units, unit parameters, and composite units</td>
<td>Section 6.4.1.1, Table 6.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Types</strong></td>
<td>direct and capability-based</td>
<td>Metamodel in Section 6.5.1.2 and Fig. 6.1, Capabilities in Section 6.5.1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Structures</strong></td>
<td>distribution properties</td>
<td>Section 6.5.2, Fig. 6.2</td>
</tr>
<tr>
<td></td>
<td>in variability models</td>
<td>Section 5.5.2.2, Fig. 5.8, Fig. 5.9</td>
</tr>
<tr>
<td></td>
<td>in ecosystems</td>
<td>Section 6.5.2, Appendix B.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tools</strong></td>
<td>configurators and installers</td>
<td>Section 4.6, Table 6.3</td>
</tr>
<tr>
<td><strong>Asset composition</strong></td>
<td>encapsulation and interaction</td>
<td>Sections 6.4.3, 6.4.4, and Table 6.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Decisions</strong></td>
<td>configuration process</td>
<td>Section 4.6.1, overview in Section 2.2.2</td>
</tr>
<tr>
<td></td>
<td>decision lifecycle and binding</td>
<td>Section 6.4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Organization</strong></td>
<td>main platform and free market, development and variability management</td>
<td>Section 6.3.1, Table 6.1</td>
</tr>
<tr>
<td></td>
<td>scale and growth</td>
<td>Section 6.3.2, Table 6.2</td>
</tr>
</tbody>
</table>
7. Discussion and Outlook

7.1.2. Phenomena, Hypotheses, and Research Questions

We now report our most interesting findings as phenomena (facts that hold about our subjects), hypotheses (proposed explanations), and interesting research directions (marked with a dashed underline) indicated by empirical data. We also discuss their impact on the research fields of SPLE and software ecosystems.

7.1.2.1. Applicability of Variability Models

As can be seen in Tables 6.1 and 6.3, the existence of a variability model correlates with a centralized variability management structure. In eCos and Linux, many developers can contribute code and changes to the variability model. However, a core team must watch the impact of changes:

**Hypothesis 1** A centralized variability model is fragile and has to be managed centrally by a small team.

However, the benefit of centralized variability models is that they enable fine-grained variability mechanisms and almost arbitrary cross-cutting contributions, since the variability model abstracts over the implementation, leveraging a flexible feature-to-code mapping.

Furthermore, distributed variability management correlates with the existence of manifest files. However, the opposite direction does not hold. eCos manages variability centrally using a model that is distributed via eCos packages. But since eCos failed to create a vibrant free market with distributed management, there is so far no evidence that distributed variability management could work when variability is described in a distributed variability model, richer than simple manifest files.

Interestingly, the structures of development and variability management are independent. This phenomenon challenges assumptions in the literature [VGP08, vGPB10, Sch10] that only distributed variability management is suited for distributed (or composition-oriented) development. The Linux kernel follows a highly distributed development process while managing variability centrally upon the variability model.

7.1.2.2. Relationship Between Variability Models and Manifests

We found that a clear difference between manifests and variability models is that manifests are always fully distributed, created as individual units with bilateral relations to other manifests, and managed as individual units. In contrast, variability models—even if split over multiple files—are created around a central hierarchy, and used and evolved as a whole. Their languages are also richer.

Variability models appear to impact dependency structures. In closed platforms, the median of declared dependencies per feature is lower than per basic unit in the open platforms. This phenomenon is seen across the subject spectrum without Android, which does not declare dependencies.
Hypothesis 2 Centrally managed variability using variability models facilitates sparse dependency structures.

Variability models let developers optimize and collapse implementation-level dependencies, while the coordination cost for these activities in a distributed setting may be too high. Still, there can be other reasons for the lower number of dependencies in the systems with variability models, so this controversial hypothesis requires confirmation.

7.1.2.3. Domain Impact

The facilities to declare constraints, and in effect dependencies, are more sophisticated and more expressive in the closed platforms with variability models. The reasons for this can be manifold, but this is likely related to the requirements of the systems domain:

Hypothesis 3 Dependency mechanisms in systems software are more expressive than in end-user applications due to the need for low-level, fine-grained, and static configuration.

The community has neither refuted nor confirmed the following controversial phenomenon emerging from our data: variability models are well-suited for projects in highly technical domains. However, this can be expressed negatively: non-technical consumers are unable to deal with complex dependencies in large models, while sparse dependencies hardly need models to be handled.

We are, however, unaware of studies explaining this complexity by performing a systematic requirements analysis and linking the requirements to dependency facilities.

7.1.2.4. Dependencies

One of our most interesting findings are capability-based dependencies, which target abstractions of functionalities—capabilities—instead of basic units or features directly\(^1\). We are not aware of SPLE literature describing such dependencies, nor any academic language supporting them. Such dependencies are essential and used in all our subjects to varying extents, even if the language (Kconfig) has no explicit concept for capabilities. Their widespread use indicates two important requirements for open platforms: i) language support and ii) management of centralized stable vocabularies.

The ecosystems with open platforms are larger and grow faster, and have a significantly higher proportion of capability-based dependencies. Although there are many reasons for the growth, such as business context, the sheer manpower of a vibrant community, or the huge market demand (in particular for mobile phones), we hypothesize that:

Hypothesis 4 A high amount of capability-based dependencies positively influences growth.

For significant impact, capabilities should not just be labels (Debian, Eclipse), but described in a rich DSL, similar to intent filters (see Appendix) in Android.

\(^1\)Recall that in variability models, capabilities are a special kind of features, see Section 4.3.1.
Finally, the proportion of features and basic units that have dependencies is surprisingly high in all our subjects. Although the numbers between closed and open platforms are hardly comparable, these measures determine the complexity that tools supporting variability, including configuration, derivation, and analysis tools must cope with.

7.1.2.5. Beyond Variability Modeling

Finally, we provide a brief discussion on observations beyond the focus of our dissertation. These are worth formulating, since they potentially affect variability modeling. The following hypothesis strives to explain the lightweight processes in open platforms, which contrast the thorough contribution filtering in closed platforms, in particular the Linux kernel.

**Hypothesis 5**  *Closed platforms must compensate missing guarantees of encapsulation and interface mechanisms with heavyweight processes and strict policies to assure quality.*

Recall that variability management in closed platforms aims at taming variability, to avoid diversity that has no business advantage. This is achieved by mechanisms such as: variability modeling, scoping (controlling and restricting contributions), maintaining variability information (unit parameters, dependencies, versioning) of basic units. These mechanisms are rather heavyweight, require advanced technical skills, and hinder contributions.

Open platforms add variability mechanisms that are different from these practices: uniform distribution channels within a free market, packaging mechanisms, maintaining capabilities, providing a common capability vocabulary, runtime resolutions of dependencies, little restrictions to contributions, highly dynamic runtimes, and interface mechanism. These variability mechanisms aim at encouraging contributions, and in fact, appear in our fastest-growing ecosystems Eclipse and Android.

The accumulation of new mechanisms of very different nature in open platforms calls for recognition of a new discipline in variability research: *Variability Encouragement*. Verifying its underlying activities—such as maintaining capability vocabularies and controlling processes with little restrictions to contributions—and relating them to known software engineering practices is an interesting agenda for follow-up research.

Furthermore, although variability management is always decentralized in the free market\(^2\), sub-groups might have emerged that coordinate variability management activities, such as scoping for a range of basic units. Identifying such groups would be a next research step to foster understanding of organizational structures in software ecosystems.

7.2. Guidelines for Practitioners

Our hypotheses and distilled observations in the previous section already provide useful hints for the development and management of software product lines and ecosystems.

\(^2\)In the main platform, however, we observed no correlation between openness of a platform and variability management, see second row in Table 6.1.
### 7.2. Guidelines for Practitioners

Table 7.2.: Derived high-level guidelines for language and tool design

<table>
<thead>
<tr>
<th>category</th>
<th>guideline</th>
<th>details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>basic concepts</td>
<td>switch and data features</td>
<td>Section 5.3.2.2, Fig. 5.2</td>
</tr>
<tr>
<td></td>
<td>feature hierarchy</td>
<td>Section 5.4.2, Fig. 5.7</td>
</tr>
<tr>
<td></td>
<td>XOR groups (other cardinalities rarely used)</td>
<td>Section 5.5.1</td>
</tr>
<tr>
<td></td>
<td>feature constraints beyond propositional logics</td>
<td>Section 5.5.2</td>
</tr>
<tr>
<td>scalability</td>
<td>defaults (literals and expressions)</td>
<td>Section 5.5.2.1</td>
</tr>
<tr>
<td></td>
<td>visibility conditions</td>
<td>Sections 5.4.2, 5.5.2.1</td>
</tr>
<tr>
<td></td>
<td>derived features</td>
<td>Section 5.5.2.1</td>
</tr>
<tr>
<td></td>
<td>modularization</td>
<td>Sections 4.4.7, 4.5.7</td>
</tr>
<tr>
<td></td>
<td>capabilities (low priority in closed, but essential—with centralized vocabulary—in open platforms)</td>
<td>Sections 5.3.2.1, 6.5.1.1, 6.5.2.3</td>
</tr>
<tr>
<td>usability</td>
<td>domain-specific adaptations (keywords)</td>
<td>Section 4.3.1</td>
</tr>
<tr>
<td>maintenance</td>
<td>decoupling of syntactic and configurator hierarchy</td>
<td>Sections 4.3.3, 5.3.2.1</td>
</tr>
<tr>
<td></td>
<td>clean language design to avoid intricate semantic interactions (e.g. between configuration constraints and visibility), which complicate development of reasoners</td>
<td>Sections 4.5.5, 4.5.8.2</td>
</tr>
<tr>
<td>Tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GUI support</td>
<td>shallow, wide models with high variations in branching</td>
<td>Section 5.4.2, Fig. 5.7</td>
</tr>
<tr>
<td></td>
<td>high proportions of features with constraints (73% avg.)</td>
<td>Section 5.5.2</td>
</tr>
<tr>
<td></td>
<td>high proportions of features participating in dependencies (75% avg.)</td>
<td>Section 5.5.2.2</td>
</tr>
<tr>
<td>process</td>
<td>choice propagation based on CSP or SMT reasoners, instead of error-prone imperative constructs</td>
<td>Section 4.6.2</td>
</tr>
<tr>
<td>scalability</td>
<td>scalability as standard evaluation requirement of tools, using our model characteristics</td>
<td></td>
</tr>
</tbody>
</table>

Based on our deep analysis of variability modeling languages and variability models, we can provide even more specific guidelines for practitioners, such as language designers, developers, and users.

Table 7.2 summarizes high-level guidelines we can derive from the results in Chapters 4 and 5. The table contains pointers to more details about a specific requirement. We emphasize that our guidelines are particularly applicable in the systems software domain. However, we conjecture that other domains have similar models. One of the most significant observations is the high proportion of dependencies in all models, which tools have to be designed for. Fortunately, these dependencies seem to only grow linearly with the model size, making scalable and intelligent tool support feasible without the need to introduce further abstractions.
7. Discussion and Outlook

7.3. Threats to Validity

Experiments conducted in the real world can never be perfect. As in any other empirical study, our results face threats to validity. In the following, we discuss these threats: first with respect to variability models in closed platforms, and second to our analysis and comparison of open platforms and their ecosystems.

7.3.1. Software Product Lines

Threats to external validity. The main threat to the external validity of our findings is that they are based on only two languages and a limited set of models. On the other hand, most are large, independently developed real-world projects, with different objectives, ranging from Linux as a general purpose kernel, over configurable system software tools, to eCos as an entire specialized real-time operating system for embedded devices. We believe that other related domains, especially embedded real-time such as automotive and avionic control software, will share many characteristics with the studied systems. Further, comparison to other feature modeling languages, shows that both are representative of the space of feature modeling.

Furthermore, we only look at the available artifacts: the languages, manuals, models, and mailing lists. We have not interviewed developers and users. We currently perform such interviews (see Section 7.4). In this dissertation, however, our confidence is based on formalizing the language concepts and on exhaustively testing the configurators and build systems with hand-crafted examples.

For Linux and eCos, we only examined one architecture each; however, both architectures represent large and mature portions of the systems: Linux’s x86 architecture covers 61% of the total of 10415 features and 67% of the total of 8M SLOC; the eCos’ i386PC covers 44% of the total of 2859 features and 33% of the total of 0.9M SLOC.

Threats to internal validity. A threat to the internal validity is that our statistics are incorrect. To reduce this risk, we instrumented the native tools to export models in our own format rather than building our own parsers. We thoroughly tested our analysis infrastructure using synthetic test cases and cross-checked overlapping statistics. We tested our formal semantics specification against the native configurators and cross-reviewed the specifications. We used the Boolean abstraction of the semantics to translate both models into Boolean formulas and run a SAT solver on them to find dead (always inactive) features. We found 114 dead features in Linux and 28 in eCos. We manually confirmed that all of them are indeed dead, either because they depend on features from another architecture or were intentionally deactivated. The other models mostly have no (axTLS, BusyBox, Fiasco, uClinux-dist), or just a few (four features in Freetz, Toybox, and uClinux-base) dead features. Only Buildroot (54 features), CoreBoot (58 features), and EmbToolkit (53 features) have proportionally many dead features.

Finally, since we have not performed interviews with the language designers, we might have misunderstood the original intention of certain language concepts and of actual features in the models. For example, the feature themes were determined by manual model
7.3. Threats to Validity

7.3.2. Software Ecosystems

**Threats to external validity.** We have purposely selected a wide range of open platforms for comparison with the closed platforms, to increase the generality of our conclusions. One may question their comparability, as they exhibit diverse technologies, abstraction levels, and granularities of units. It is also not given that the studied subjects are representative for open platforms in general. We mitigate this threat by using an exploratory research method: instead of testing hypotheses, we record observed phenomena and generate hypotheses. Further, we limit data sources to reliable documents, freely available source code, and tools. Confronting our results with other data, such as developer interviews, would be a valuable future project.

Specifically, the dependencies seem difficult to compare between the ecosystems with variability model and those without—the relevance of declared dependencies might differ among our subjects. For example, Android apps are rather self-contained and bundled with libraries, whereas Debian and Eclipse invest a significant effort into reducing code duplication by providing common library packages as units and making dependencies explicit. Still, all these numbers indicate scalability requirements for tools, such as configurators and installers, and in that sense (algorithmic hardness) are useful standalone and, to a large extent, comparable.

**Threats to internal validity.** In the quantitative ecosystems analysis, some numbers are estimated using interpolations and safe assumptions (lower bounds) and may be inaccurate. We address this threat by giving detailed information on our data sources, providing additional diagrams (Appendix B.5) and implementation details on the Android analysis (Appendix B.4).

The analysis of dependencies in Debian and Eclipse disregards dependencies on particular unit versions that may impact accuracy. We believe this simplification is acceptable, as such dependencies are mainly used to assist system upgrades, not in scope of our work.

All ecosystems except Android declare dependencies. It is not clear whether our extracted—via static analysis—dependencies for Android are comparable to declared dependencies—in fact it is subject of ongoing research, whether actual and declared dependencies are generally comparable or not. Therefore, we avoid comparing dependency numbers for Android to other systems.

Finally, since the platforms show significant differences both in scope and number of developers, one might question their comparability to each other. For instance, Debian with over 1000 developer is in a better position to implement cross-cutting changes to its repository than eCos, which is driven by a handful of volunteers. Investigation of how the employed processes affect the collected data is left for further research.
7.4. Outlook: Industrial Variability Modeling

As a natural follow-up of our work on open source variability modeling, we started a study on industrial practices. However, given the obstacles mentioned earlier, such as highly protected variability models, we cannot perform artifact studies. Although we received permission from a few companies to analyze models, it is questionable whether we could generalize from such a small number of case studies. Thus, our research tools are quantitative survey questionnaires, qualitative interviews, and grounded theory \([\text{GS67}]\) to analyze results of the latter.

A project website\(^3\) reports the current status of our study. In this section, we provide a glimpse on preliminary results of our survey questionnaire\(^4\). We also conducted eight interviews, but are still analyzing the results\(^5\) and will continue with more interviews soon.

7.4.1. Methodology

Our goal is to understand characteristics of industrial variability models, their creation process\(^6\), and the tools\(^7\) that are used. Therefore, we follow a mixed-methods approach: first, we design and distribute a short survey questionnaire on variability modeling; second, we analyze its results and identify case studies for deeper analysis; third, we perform interviews; and fourth, analyze results using grounded theory with open coding \([\text{Kha09}]\).

We distributed the questionnaire to over 60 practitioners and researchers having industrial experience, including an invitation to forward the questionnaire to further colleagues. Our selection comprised our own industrial partners, academic colleagues with industrial background, customers of Fraunhofer IESE\(^8\), and companies from the software product line Hall of Fame (Section 2.1.1). We also spread the survey questionnaire at VaMoS’12\(^9\).

**Questionnaire Design.** We designed a short questionnaire targeting practitioners that participated in at least one software product line project applying variability modeling. The final questionnaire is contained in Appendix C and elicits the following information:

- the purpose and benefit of variability modeling;
- the notations and tools used;
- the scale of models created;
- problems that occurred;
- context information, such as characteristics of the product line.

---

\(^3\)http://gsd.uwaterloo.ca/industrial-variability-modeling  
\(^4\)The questionnaire was designed as part of Ralf Rublack’s Master’s thesis (Diplomarbeit), supervised in the context of this dissertation.  
\(^5\)The analysis, and some interviews, are performed in collaboration with the PhD candidate Divya Nair.  
\(^6\)Divya Nair’s focus  
\(^7\)Ralf Rublack’s focus  
\(^8\)In collaboration with Martin Becker, http://www.iese.fraunhofer.de.  
\(^9\)http://www.vamos-workshop.net
7.4. Outlook: Industrial Variability Modeling

At the end of the questionnaire, we asked for contact information to verify results, identify information about duplicate product line projects, and to possibly follow-up with clarification questions or an interview invitation.

7.4.2. Preliminary Survey Results

Although we have not fully analyzed the questionnaire results and interviews yet, we provide a selection of preliminary results that provide an insight into industrial practices and give a glimpse on how our results from the open source projects compare to commercial models.

We received 42 responses by individuals from 16 countries, most of which originating from Germany (24%), USA (12%), Canada (12%), Sweden (7%), Austria (5%), Norway (5%), Brazil (5%), and Spain (5%). The majority of participants had a clear industrial background, with professional experience ranging from <1 to >10 years, and comprising roles such as developer, modeler, team leader, project manager, domain expert, product manager, marketing expert, and researcher. The product lines our respondents were involved with stem from a broad range of domains, for example: automotive, eCommerce, business applications, defense, enterprise resource planning, cyber-physical systems, power industry, telecommunication, and many more.

In general, most of our respondents (>91%) find variability modeling useful\(^{10}\). 76% use a separate model to describe variability, while 47% annotate existing implementation artifacts, for example using built-in annotations of the Spring\(^{11}\) [Joh04] component framework.

7.4.2.1. Benefit of Variability Modeling

While we hypothesize that the primary application of our open source variability modeling languages CDL and Kconfig is product configuration, our survey reveals many more benefits of modeling variability. As shown in Fig. 7.2, many respondents use variability modeling to manage, plan, and document variability, for example, to support developers in keeping an overview on variability, which is also confirmed by our interviews so far. Interestingly, some respondents also use it for marketing purposes, and—as a free-text response—to estimate costs of products.

7.4.2.2. Notations and Tools

As shown in Fig. 7.3, the feature model is the dominant representation of variability among our survey participants. However, many formal (e.g. decision model, DSL, ADL, UML-based representation) and informal (spreadsheet, free-text description) alternatives are commonly used too. Some participants also use the configuration facilities of a component

---

\(^{10}\) However, there is a significant bias in this question, since we only approached participants that successfully applied variability modeling. Thoroughly addressing this issue (first question in Appendix C) would require to identify companies that failed in applying variability modeling, which is beyond our scope.

\(^{11}\) http://www.springframework.org
7. Discussion and Outlook

Figure 7.2.: Benefit of applying variability modeling

framework (e.g. Spring, OSGi, EJB), or describe variability as semi-structured key/value pairs in XML- or text-based property files. Although the latter techniques can hardly be called variability modeling, this multitude of different approaches calls for further research on the benefits and limitations of variability modeling, in particular on indicators that can predict whether the additional effort of explicitly modeling variability pays off in a project.

Figure 7.3.: Notations used to specify variability

Many of the tools listed in Section 2.2, which we know from literature, are used by our respondents. As shown in Fig. 7.4, the most frequently used commercial tool is pure::variants—perhaps not surprising given that 65% of our respondents are based in Europe, while the second major commercial tool Gears focuses on north american customers.

Our survey also reveals many smaller tools we were not aware yet, such as the decision-model-based tool Tecnalia PLUM [AE07], Hephaestus [BTB09], v.control [MR09], or SPREBA [SG09]. Interestingly, 35% of the respondents use home-grown domain-specific tools, for example, based on Eclipse EMF/xtext, Simulink or IBM Rational Software Architect. Even Microsoft Excel is reported.
7.4. Outlook: Industrial Variability Modeling

7.4.2.3. Scales of Variability Models

Given the unpredictable, heterogeneous modeling languages of our participants, we carefully asked for the number of “units of variability” of their variability models. These units were in most cases features or variation points—both reported by 74%; followed by configuration options (63%), decisions (29%), and calibration parameters (23%). One participant also reported deltas. Investigating overlaps between these units among the responses, for example, whether participants treat features and configuration options as the same entity, is part of our ongoing research. Furthermore, it is not clear whether respondents using pure::variants refer to the feature model or the family model—the latter representing the solution space [Beu04].

Table 7.3 summarizes the percentage of participants reporting their number of models with a specific size. Although it requires further research on the particular “units of variability”, it shows that very large models exist, with almost 22% of participants having models with more than 10,000 units. The majority of models has less than 1000 units, while most participants (additionally or solely, as yet to be analyzed) deal with models that have less than 50 units.

Table 7.3.: Scales of variability models

<table>
<thead>
<tr>
<th></th>
<th>&lt;50 units</th>
<th>51–100 units</th>
<th>101–1000 units</th>
<th>1001–10000 units</th>
<th>&gt;10000 units</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 models</td>
<td>11.9%</td>
<td>19.0%</td>
<td>19.0%</td>
<td>40.5%</td>
<td>38.1%</td>
</tr>
<tr>
<td>1 model</td>
<td>35.7%</td>
<td>23.8%</td>
<td>28.5%</td>
<td>14.3%</td>
<td>11.9%</td>
</tr>
<tr>
<td>2–5 models</td>
<td>9.5%</td>
<td>14.3%</td>
<td>11.1%</td>
<td>0%</td>
<td>4.8%</td>
</tr>
<tr>
<td>&gt;5 models</td>
<td>16.7%</td>
<td>7.1%</td>
<td>7.1%</td>
<td>7.1%</td>
<td>4.8%</td>
</tr>
<tr>
<td>sum (≥1 model)</td>
<td>61.9%</td>
<td>45.2%</td>
<td>46.7%</td>
<td>21.4%</td>
<td>21.5%</td>
</tr>
</tbody>
</table>

7.4.2.4. Complexity Problems

We asked for specific complexity issues that our practitioners faced with variability modeling. As shown in Fig. 7.5, the most frequently reported problem lies in the
7. Discussion and Outlook

evolution, followed by the visualization of variability models. Dependency management and problems with the configuration process, such as resolving conflicts, have only slightly lower frequency. Since our checkbox for dependency management included explosion of dependencies as an example, we strive to expand on dependency management in the interviews. Recall that dependencies in our open source product lines only grew linearly.

![Figure 7.5.: Reported complexity problems](image)

Finally, the free-text answers to this question included, among others, modularization for multi product lines [Kru06], tests, model reduction, but also statements such as “getting developers to understand why we do this, and the correct patterns to use”.

![Figure 7.6.: Reported strategies to cope with complexity problems](image)

The strategies that our respondents use to tackle these issues are manifold, see Fig. 7.6. 60% organize multiple models in a hierarchy; however, since this proportion is surprisingly high, we will investigate whether some accidentally referred to the intra-model organization, such as the feature hierarchy. Two other frequent strategies—decomposition into multiple models and the use of model reasoners—confirm our observations from variability modeling in the systems software domain. Many of the remaining responses in Fig. 7.6 require further investigation in order to draw conclusions. Two interesting free-text answers confirm our Hypothesis 1 (Section 7.1.2), such as the statement “assign configuration / variability dependent tasks to a small selection of people”.

142
7.4.2.5. Further Observations

Product line introduction strategy. Only 30% of our respondents developed any of their product lines in a pro-active strategy, that is, scoped, designed, and developed the platform before any product was derived, as is the typical SPLE approach. More frequently—in 50% of the cases—multiple existing products were re-engineered into a product line, and still 45% of the respondents evolved a single initial product into a product line. A combination of any of these three strategies is reported by a quarter of our participants.

This observation confirms, so far, one of the common hypotheses in the product line community that only a minority of real-world product lines follow a pro-active strategy. It is a call for further research on migration support from single products into product lines with systematic variability management.

Broad perspective on variability modeling. One of the most interesting comments on the survey questionnaire itself retroactively supports our study on variability in non-embedded and non-systems-software projects that use modern dynamic languages. It also confirms our Hypothesis 3 (Section 7.1.2):

Both the field and this study could use a broadening of perspective. My day job is building Java based server side software, which tends to be one of a kind, non product line type software. Java is a rich language and technologies such as spring and maven provide very rich variability tooling. Additionally, using things like puppet for deploying into cloud architectures as well as staging and testing infrastructure means that we have a lot of variability. We deploy in different configurations to different data centers, use feature flags as well as AB testing to test new functionality, etc. My feeling is that the research field still assumes a traditional low tech embedded software perspective where the lack of a lot of things need to be compensated for with variability modeling tooling and cumbersome build systems. So, I don’t model variability, instead I make software that has variation points that are explicitly configurable. The activities of developing and designing when following a continuous deployment model are inseparable.

7.4.3. Preliminary Interview Results

We conducted eight in-depth interviews (in person at conferences or via skype) with two tool vendors and suppliers of automotive, eCommerce, campus management software, power management, and engine control software. These provide interesting insights into the development process of variability models, and the strategies to cope with scalability- and complexity problems.

For example, we learn that most product lines are incepted by refactoring a range of existing products, where a major challenge lies in the identification and organization of features based on product differences. The strategy of one tool vendor to tackle this
7. Discussion and Outlook

challenge is to organize workshops with developers and domain experts and to iteratively identify the reasons for each product difference. After around three iterations, a feature is created that will be mapped to a set of product differences—as a prerequisite to merge the product code into a product line and realize variation points with variability mechanisms.

We also observe two kinds of companies: those that established their software product lines following practices from the literature, and those that were not aware of this field and started to develop variability management and modeling infrastructures on their own. The latter reflects the situation of our two open source subject languages CDL and Kconfig. Interestingly, so far, the product lines of the first kind are smaller in terms of variability model and code size. The second kind of companies created approaches and techniques known from SPLE, but with additional scalability concepts in their languages, tools, and processes. This emphasizes i) that we have no clear empirical evidence that SPLE approaches purely developed in academia scale to large product lines, and ii) that it is necessary to study real, large-scale industrial software product lines.

Finally, none of our interviewees confirmed the usability of configurators for end users. So far, all pointed out that an expert with further domain and implementation knowledge is required to derive a product. However, custom-made and user-friendly wizards mitigate this issue. Whether configurators like those of CDL and Kconfig (Section 4.2, Fig. 4.2) can be used by non-technically skilled consumers, remains an interesting future research question that needs to be addressed by a user study.
Conclusions

The real world is more complex than reflected in most existing theoretical research on variability modeling. While a large number of techniques for software product lines have been introduced both in academia and industry, by going a step back and studying existing practices, we lay the ground for improved language and tool support in the future, based on a refined theoretical foundation. We provide qualitative and quantitative empirical evidence to the variability modeling research community for the real-world use of its flagship concepts. However, our empirical discourse from software product lines to software ecosystems shows that more concepts have to be considered in research. We raise awareness for these concepts and also identify new and challenging research questions that can be addressed by building on top of our work.

With our in-depth analysis of two real-world variability modeling languages, which were conceived independently of the research community, and 13 instances stemming from small to large-scale industrially used projects, we contribute to the knowledge about variability modeling concepts, their semantics, and their use in real-world models. These results are complemented by a study of industrial variability modeling practices.

Our work also sharpens the relationship between software product lines and software ecosystems by investigating variability mechanisms in open platforms and discussing the applicability of variability models and related concepts from SPLE. In fact, the boundaries between product lines and ecosystems are blurred, and at least two of our investigated systems with closed platforms—eCos and the Linux kernel—can be seen as software ecosystems with a free market of third-party contributions around their main platform.

8.1. Summary of Results

With respect to our four main research questions formulated in Section 1.3, we claim the following results:

8.1.1. Research Question RQ1

RQ1.1. We confirm the use of the well-researched concepts of FODA feature models in our subject languages. More precisely, we identified concepts that have the same semantics as their counterparts in feature modeling, and we established a mapping. These feature
modeling concepts comprise Boolean (optional) features, a hierarchy, group and cross-tree constraints. However, some of these concepts have interesting characteristics, such as the separation of syntactic and configurator hierarchy, or domain-specific adaptations that foster understandability for developers and users.

**RQ1.2.** We identified concepts beyond FODA feature models, such as: visibility conditions, derived features and derived defaults, and binding modes. We also observed a mixed feature representation, that is, features with a Boolean and a data value in CDL, or three-state features that express binding modes as in Kconfig. Both introduce intricate semantic interactions not obvious from syntax, and complicate development of reasoners. However, their common use in models indicates that existing techniques should take such extended concepts into account. In general, most of the additional concepts aim at scaling variability modeling to the huge configuration spaces that our subject systems encompass.

**RQ1.3.** We formulated configuration space semantics for our real-world languages in a denotational style. These semantics turned out to be more comprehensive and intricate than those of most feature modeling languages. We implemented the semantics in our analysis tools. Our propositional abstraction, which approximates the configuration space of the full semantics, provided the basis for SAT-based analyses. We used it to reason about dead features and about violations of the child-parent implications in our models.

**RQ1.4.** We inspected the CDL and Kconfig configurator with respect to their configuration process and reasoning support. We learn that both follow a re-configuration process; however, each takes a different approach to ensure that the user reaches a valid configuration. The Kconfig configurator prevents the user from modifications that violate constraints using a simple mechanism; the eCos configurator allows such modifications, but detects violations and helps in resolving them using a home-grown, CSP-solver-like inference engine. However, the reasoning procedures are incomplete and may propose misleading guidance; nevertheless, the configurator can cope with more expressive constraints than most existing feature model reasoners.

**8.1.2. Research Question RQ2**

We analyzed our models qualitatively and quantitatively, the latter using a well-defined set of metrics (Table 5.6).

**RQ2.1 and RQ2.2.** Our qualitative model analysis characterized the model content and identified design criteria that were used to create the model hierarchies. For the former, we created themes of features and learn that the models are used to configure nearly every aspect of a project, including the build system and the installation process. For the latter, we identified common feature grouping patterns. Interestingly, the feature hierarchy does not reflect the architectural structure of the projects, which indicates that
8.1. Summary of Results

models are truly orthogonal. This challenges reverse-engineering efforts, as discussed in Section 5.6.

RQ2.3. Our quantitative model analysis shows the use and frequency of the identified modeling concepts, and thereby provides requirements for language and tool designers. The extracted models can be used as benchmarks to evaluate variability modeling techniques. In particular, we learn that:

Constraints beyond hierarchy are very frequently used. Grouping of optional features is very common with up to 28% of the features in a model. Constrained groups (mutex, xor) are rare, and or groups are never used. We find constraints over switch (Boolean) features, as well as arithmetic and string constraints. We found surprisingly high proportions of features with constraints and features participating in such. However, the identified linear growth of dependencies with model size is encouraging for tool support. Dependencies do not explode and the dependency graph remains relatively sparse.

RQ2.4 Our constraint and dependency measures, and the shapes of our models, significantly challenge previous assumptions made in literature. While we mainly see small, well-balanced feature trees, our models are large, have high variation in branching, grow in breadth instead of depth, and encompass high proportions of non-Boolean constraints. This empirical evidence calls for improved and more expressive variability modeling techniques, and their evaluation on our extracted models.

8.1.3. Research Question RQ3

Recognizing the demand for broadening the perspective on variability, we extend our discourse to open platforms with five case studies on software ecosystems. For RQ3.1, we studied the organizational structures and achieved scales of each, learning that each has a controlled, central part, and while development is always distributed, variability management in systems with variability models is centralized, requiring a small group of developers controlling it. For RQ3.2, we identified a broad range of variability mechanisms, characterize and compare them between closed and open platforms. We observe that closed platforms allow almost arbitrary changes, but need heavyweight processes to assure quality. With regard to RQ3.3, a frequent pattern we found in the facilities to declare dependencies (variability models or manifest files) are capabilities and capability-based dependencies. In the dependency structures—addressed in RQ3.4—such dependencies are increasingly used in ecosystems of open platforms to reduce coupling.

8.1.4. Research Question RQ4

The results of our previous research questions provided the basis to investigate correlations, to cross-link concepts, and to find causalities. Towards building a theory spanning variability modeling concepts in software product lines and software ecosystems, we contribute a conceptual framework and a set of testable hypotheses explaining observations
made throughout our discourse. Confirmation or refutation of our hypotheses by follow-
up research will eventually lead to a refined theory behind variability modeling with a
stronger empirical background.

Our analysis revealed that variability models—while providing system-wide abstractions
over code—work best in centralized variability management; that ecosystem growth relies
on capability-based dependencies; and that open platforms with vibrant free markets
imply capability-based dependencies, which rely on a centralized and a stable vocabulary.

8.2. Research Impact

Our work has been well-received by the community and already influenced follow-up
work and evaluation techniques:

- Experience gathered during our studies went into the design of OMG’s upcoming
  CVL standard [Obj09]—through participation of Krzysztof Czarnecki and Andrzej
  Wąsowski in the respective proposal [CVL12]. This includes, for example, derived
  features, and the richness of the constraint language.
- Among others, our extracted models were used by Janota [Jan10] to evaluate a scalable
  approach to valid-domain computation (cf. Section 2.2.2), by Xiong et al. [XHSC12]
  to provide scalable conflict resolution support evaluated on the eCos model, by
  Hubaux [Hub12] to evaluate a workflow-driven feature configuration process, and the
  (already frequently mentioned) reverse-engineering approach by She et al. [SLB+11].
  Our infrastructure was also used for our work in Passos et al. [PNX+11], a study of
  all 116 architecture-specific eCos models.
- The statically extracted feature-to-code mapping (presence conditions) from the
  Linux kernel was used in Kästner et al. [KGR+11] to evaluate the variability-aware
  parsing approach, and by Dietrich et al. [DTSPL12] to compare our explicit presence
  conditions to an alternative form of feature-to-code mapping, derived using dynamic
  analysis.
- Our Android dataset has been used by us in Mojica et al. [MND+13] to study user
  ratings of mobile apps.
- Our results guide the architecture design of the nationally funded R&D project
  EUMONIS\textsuperscript{1}, which strives for an open platform aiming at establishing an ecosystem
  with third-party contributions.
- Finally, our ASE’10 [BSL+10] paper as of today already has 60 citations according
  to Google Scholar\textsuperscript{2}, most of which cite it to emphasize the complexity of real-world
  variability models.

\textsuperscript{1}German Federal Ministry of Education and Research, project 01IS10033K: \url{http://www.eumonis.org}
\textsuperscript{2}\url{http://scholar.google.com}
8.3. Perspective

Our dissertation, its results and contributions opens the following perspectives on future research.

Model reasoners. In SPLE research, a variety of reasoners has been used to create feature model analyzers and configurators, including CSP solvers [WSB⁺08], SAT solvers [TBK09, MWC09], BDD packages [MWCC08], and OWL reasoners [WLS⁺05]. These works tested the reasoners on either small meaningful models or large automatically generated models; however, it is not clear how these tools will scale to handle the eCos and Linux kernel model. Investigating their scalability and improving these tools remains future work. In particular, leveraging SMT solvers [RT06, BSST09] to reason about models in our expressive languages is an interesting future research direction.

Furthermore, related work in Section 3.5 has emphasized the potential of transferring results from the field of knowledge-based configuration, in particular existing reasoners, to software configuration. However, despite existing research agendas and initial work, it is largely unclear how models from the two worlds compare.

Relationship to code. The dependencies among features in variability models reflect code dependencies. Each valid configuration has to lead to a valid product, thus, code dependencies need to be satisfied. However, based on our experiences, we hypothesize that variability models are over-constrained, that is, contain more dependencies than can be extracted from code.

One of our future work is to compare code dependencies with feature dependencies, investigate the overlap, and the unique dependencies in each. This amounts to statically extracting code dependencies, deriving the presence conditions of the corresponding code fragments, and deriving dependencies among features—a non-trivial static analysis infrastructure. We have such an infrastructure—the one that was developed by us to evaluate the reverse-engineering approach in [SLB⁺11] (cf. Section 1.6). However, it is based on the srcML³ C parser, which turned out to be error-prone⁴, since it was never designed to work with undisciplined #IFDEFs. Adapting the infrastructure to Kästner et al.’s partial preprocessor [KGR⁺11] is one of the next steps. Nevertheless, we already conducted a presudy on sampled dependencies in the model, aiming to identify the dependency in code, and characterizing the static analysis technique needed for recognition.

Industrial variability modeling. We will finish our currently ongoing study on commercial languages and models, as presented in Section 7.4. How these languages and models relate to our open source subjects is still a research challenge. We need to deepen our

³http://www.sdml.info/projects/srcml
⁴We calculated a large formula for FreeBSD, comprising over 10,000 conjunctive elements of varying size, and abstracted it to propositional logic, using a similar translation as in Section 4.4.9. However, the formula turned out to be unsatisfiable, and debugging unleashed limitations of our tool.
8. Conclusions

understanding of commercial languages, establish a mapping—as we did between CDL, Kconfig, and feature models (Table 4.1)—and gather further detailed information on the models. Therefore, we currently analyze the interviews using grounded theory.

**Modeling process.** Both the surveyed related work in Chapter 3 and our industrial variability modeling study show that methodological support for creating variability models is still a major lack. Studying the modeling processes in industry and deriving guidelines, such as to identify features or organize the model, constitutes a research objective that we currently address in our ongoing industrial variability modeling study.

**Software ecosystems.** Our exploratory study on software ecosystems paved the way to follow-up work that aims at understanding the rules and forces underlying the evolution of software ecosystems.

One direction for future research is variability encouragement. We have seen variability mechanisms in open platforms that are lean and lightweight and focus on encouraging variability to grow ecosystems. Closed platforms focus on managing variability, that is, avoid variability that has no clear business advantage. The new mechanisms of different nature found in open platforms call for research on variability encouragement, as has been done for variability management before.

Our huge extracted dependency graphs, in particular that of Android, are an invitation to further studies. A possible research direction is to study the feature interaction problem [CKMRM03] within ecosystems, in order to draw conclusions about modularity and how developers achieve it. For our largest ecosystem Android, this requires to emulate the matching of intents and intent filters in our static analysis infrastructure, in order to derive a more precise dependency graph\(^5\).

Finally, we observed distributed variability management among the open platforms. However, identifying sub-groups that perform variability management activities (such as scoping and dependency management) together, and studying their dynamics, could significantly increase our understanding of organizational structures and rules in software ecosystems.

---

\(^5\)Currently, we cannot calculate reverse dependencies due to complex relationships between intent filters, see Appendix B.4.
A.1. CDLTools

Our CDLTools analysis tool infrastructure is available as open source on Google Code\(^1\), together with documentation on its use. The project includes an extension to the original eCos configurator (ConfigTool) that outputs models in our own format (IML—intermediate model). The CDLTools’ main program can parse, analyze, and transform CDL models in the IML format. CDLTools is developed in Scala\(^2\) and relies on the strategic programming [LVV04] framework Kiama\(^3\). More information is available on the Google Code project website.

A.2. KBuildMiner

As described in Section 4.5.6, KBuildMiner is a static control flow analysis tool that can derive presence conditions from imperative KBuild Makefiles. It is also available on Google Code\(^4\) and implemented in Scala using Kiama.

We used it to extract file presence conditions from several versions of the Linux kernel. Since these did not contribute to the conclusions of our main dissertation, we only provide the datasets (see website) and two statistics about the presence conditions in this Appendix.

For Linux 2.6.28.6, we extracted 7,243 presence conditions out of 596 makefiles in the x86 branch, whereas we had to manually adapt 28 makefiles for our parser, which contained non-recognizable statements. The extracted conditions cover 94\% of all Linux source files. We found the following reasons for uncovered files: first, many C files were only included via other C files; second, additional obscure build logic was used; and third, files belong to additional non-kernel tools; and fourth, some files were actually unreachable.

Fig. A.1 visualizes basic properties of the collected presence conditions. We find that the majority of features (87\%) appear in less than four presence conditions (Fig. A.1a).

\(^1\)http://code.google.com/p/variability/wiki/CDLTools
\(^2\)http://www.scala-lang.org
\(^3\)http://code.google.com/p/kiama
\(^4\)http://code.google.com/p/variability/wiki/PresenceConditionsExtraction
A. Analysis Tool Infrastructure

Figure A.1: (a) The number of presence conditions that a feature appears in, and (b) size of presence conditions. X-axes are pruned to 15).

However diversity is wide, with some features such as SND (sound support) appearing in 424 presence conditions.

Next in Fig. A.1b, we see that the presence conditions become large with up to 24 unique features referenced. The largest condition belongs to isdn/hisax/arcofi.c, which provides common functions across all HiSax drivers—a set of drivers for various Siemens ISDN cards. Features in the driver set required the compilation of this file. Interestingly, we found only a small number files that were unconditionally included in the end product, reflected as files with zero features referenced. The majority of files is indeed part of the variability of the platform.

A.3. Linux Variability Analysis Tools

The Linux Variability Analysis Tools (LVAT) were developed by Steven She to parse and transform Kconfig models, based on our formal semantics. Steven She uses the tools for reverse-engineering feature model from formulas representing the configuration space of a product line.

More detailed information on LVAT is available on the Google Code project site\(^5\) and in corresponding publications [SLB\(^+\)11, SB10, ACSW12].

A.4. Models

All 116 extracted architecture-specific eCos models are available in the repository of CDLTools\(^6\). All Kconfig models stemming from our subject projects of the systems software domain (cf. Section 5.2), are available in a wiki page\(^7\) on Google Code.

We currently aim at improving the organization of the models on our website, and on the integration of our command-line-based tools into a coherent tool suite. Improving the usability of CDLTools, LVAT, and KBuildMiner is part of ongoing effort.

\(^5\)http://code.google.com/p/linux-variability-analysis-tools\n\(^6\)http://code.google.com/p/variability/source/browse/#hg/CDLTools/input/iml\n\(^7\)http://code.google.com/p/variability/wiki/KconfigModels
As a final note, and sadly to say, we have to admit that the names we gave to our tools scored very badly on the yearly FOSD Cool Wall\textsuperscript{8}. Therefore, this thesis contains the explicit commitment to take any effort to greatly improve the coolness factor of our future variability analysis tools ☺.

\textsuperscript{8}http://www.tu-braunschweig.de/Medien-DB/isf/sse/fosd12vortraege/coolwall2012.pdf
This Appendix provides supporting details for Chapter 6, that is, scale and growth rate estimations, statistics on dependency structures, and the implementation of the static analysis of Android bytecode.

Since the corresponding publication \([\text{BPT}^+]\) is under review, we have not published the tools and datasets yet. However, we provide a ZIP archive\(^1\) that contains raw and synthesized datasets (subfolder \texttt{datasets/}), and the sources of our tools (subfolder \texttt{tools/}) to extract and analyze the datasets.

\section*{B.1. Scales and Growth Rates}

We estimate scales and growth rates of our five subject systems. Given their diffuse boundaries—in particular of the free market—our strategy is to estimate conservative lower bounds for the main platform and free market sizes (shown in Table \ref{tab:scalegrow}). Note that we carefully avoided drawing strong conclusions using these estimations.

\subsection*{B.1.1. Current Sizes}

For \texttt{eCos} and \texttt{Linux}, we count the LOC of the current main platforms (\texttt{eCos 3.0}\(^2\), \texttt{Linux 2.6.32}\(^3\)) using the tool \texttt{sloccount} \cite{Whe02}. While we cannot estimate the free market of Linux, for \texttt{eCos}, we identify 9 freeware packages on the \texttt{eCosCentric} website\(^4\) and FTP server\(^5\) and calculate their LOC measures, as summarized in Table \ref{tab:scalegrow}. Although around ten more commercial packages are listed on the website, we stick to the safe lower bound of freeware packages.

For \texttt{Debian}, we estimate the size of the main platform by multiplying the number of i386 binary packages in the main repository of the Debian 6.0 (\texttt{squeeze}) release\(^6\) with an average package size that stems from an external report \cite{GBRM+09}. The latter shows that the average size remained stable at 28K LOC over time. For the free market, we\(^7\)

\begin{itemize}
  \item \([\text{http://informatik.uni-leipzig.de/~berger/ecosystems/appendix.zip (260MB)}]\)
  \item \([\text{http://hg-pub.ecoscentric.com/ecos-v3.0-branch/}}\]
  \item \([\text{ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.tar.gz}}\]
  \item \([\text{http://www.ecoscentric.com}}\]
  \item \([\text{ftp://ftp.ecoscentric.com}}\]
  \item \([\text{ftp://ftp.debian.org/debian/dists/squeeze/main/binary-i386}}\]
  \item \([\text{Crawling and aggregation was done by Reinhard Tartler}}\]
\end{itemize}
B. Software Ecosystem Statistics

<table>
<thead>
<tr>
<th>package</th>
<th>features</th>
<th>files</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>yaffs-gpl1_1_0</td>
<td>23</td>
<td>37</td>
<td>12553</td>
</tr>
<tr>
<td>nand-1_1_0 package set:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYGPKG_IO_NAND</td>
<td>20</td>
<td>27</td>
<td>2863</td>
</tr>
<tr>
<td>CYGPKG_DEVS_NAND_SAMSUNG_K9</td>
<td>7</td>
<td>4</td>
<td>436</td>
</tr>
<tr>
<td>CYGPKG_DEVS_NAND_SYNTH</td>
<td>27</td>
<td>5</td>
<td>2496</td>
</tr>
<tr>
<td>CYGPKG_DEVS_NAND_ST_NANDXXXX3A</td>
<td>1</td>
<td>2</td>
<td>294</td>
</tr>
<tr>
<td>CYGPKG_DEVS_NAND_MICRON_MT29F</td>
<td>6</td>
<td>3</td>
<td>315</td>
</tr>
<tr>
<td>CYGPKG_DEVS_NAND_ARM_AT91SAM9</td>
<td>2</td>
<td>3</td>
<td>189</td>
</tr>
<tr>
<td>bsd_crypto-20031113</td>
<td>3</td>
<td>24</td>
<td>5161</td>
</tr>
<tr>
<td>openssl-1.9.6b</td>
<td>46</td>
<td>551</td>
<td>95256</td>
</tr>
</tbody>
</table>

Table B.1.: eCos free market packages

collect package indices from all third-party repositories listed on the APT-GET.ORG website\(^8\), and estimate the size likewise.

For **Eclipse**, we estimate the main platform scale by downloading all project repositories associated with the Helios 3.6 release (listed on the Eclipse website\(^9\)) and running sloccount. For the free market, we multiplied all available bundles on Eclipse Marketplace and Yoxos with the average bundle size known from the main platform. The resulting moderate size (6.9M LOC) might be significantly larger in reality, since the ecosystem is scattered and bundles are also available on many other third-party repositories.

For **Android**, we estimate both main platform and free market sizes using an average app size calculated as follows. We selected a random sample of 150 apps, which we converted from Dalvik to Java bytecode, in order to reconstruct source code with a Java decompiler\(^10\) and measure using sloccount. As third-party libraries are directly compiled into apps, we also identified and subtracted their sizes by identifying duplicated code over the whole (281k apps) ecosystem subset (based on calculating hash codes for each Java package subfolder in the apps). Finally, using a median bootstrap analysis\(^11\), the average app size excluding libraries amounts to 1,541 LOC (confidence interval of [1164,2239]). Interestingly, around 3/4 of an app’s code belongs to libraries. Finally, we multiplied the average app size with a public estimation of currently available apps\(^12\). In contrast, Android’s main platform is marginal. The Google Nexus S mobile device (Android v2.3.4) contains only 83 apps with estimated 128K LOC, and the Android OS itself comprising around 1M LOC\(^13\).

---

\(^8\)http://apt-get.org
\(^9\)http://dev.eclipse.org/viewcvs/viewvc.cgi
\(^10\)http://java.decompiler.free.fr/?q=qjdgui
\(^11\)http://reference.wolfram.com/mathematica/howto/PerformABootstrapAnalysis.html
\(^12\)http://appbrain.com/stats/number-of-android-apps
\(^13\)http://thenextweb.com/google/2011/10/19/googles-andy-rubin-there-are-over-1-million-lines-of-code-in-android
### B.2. Tools

We developed significant infrastructure to quantitatively analyze our three open platforms, while we built upon LVAT and CDLTools (see Appendix A) for the closed platform eCos and Linux. The Appendix ZIP archive contains the Debian script (`tools/debian/stats.py`), the Eclipse tool (`tools/eclipse/`), and the Android static analysis tool (`tools/android/android-analysis.zip`). We also developed scripts in Scala and R to create diagrams and analyze statistics. These scripts are available on request.

---

**Table B.2.: Ecosystem growth rate estimations**

<table>
<thead>
<tr>
<th>System</th>
<th>Inception year</th>
<th>Lifespan (years)</th>
<th>LOC at inception</th>
<th>Current LOC (lower bound)</th>
<th>Basic units at inception</th>
<th>Current basic units (lower bound)</th>
<th>Avg. basic unit LOC</th>
<th>Exponent growth rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>eCos</td>
<td>1999</td>
<td>10</td>
<td>75,953</td>
<td>1,247,628</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32%</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>1991</td>
<td>20</td>
<td>10,239</td>
<td>7,982,651</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>39%</td>
</tr>
<tr>
<td>Debian</td>
<td>1996</td>
<td>15</td>
<td>13,129,326</td>
<td>1,200,000,000 ^4</td>
<td>43411</td>
<td>27699</td>
<td>-</td>
<td>35%</td>
</tr>
<tr>
<td>Eclipse</td>
<td>2001</td>
<td>9</td>
<td>140,790</td>
<td>28,100,000 ^4</td>
<td>38 ^6</td>
<td>7684</td>
<td>3705</td>
<td>80%</td>
</tr>
<tr>
<td>Android</td>
<td>2008</td>
<td>3.5</td>
<td>1,128,000 ^7</td>
<td>621,023,000</td>
<td>50 ^8</td>
<td>406,821</td>
<td>1541</td>
<td>507%</td>
</tr>
</tbody>
</table>

---

4. Estimated, see Section B.1.1
5. Estimated: avg. basic unit size multiplied by initial number of basic units
7. Estimated: avg. basic unit size multiplied by initial number of basic units (assuming also initial Android main platform size of 1M LOC leads to lower bound, cf. Section B.1.1)

---

**B.1.2. Growth Rates**

To estimate yearly growth rates, we fit an exponential growth function (compound interest) to the difference between ecosystem sizes at their inception and their current state. We carefully assure that these measures are lower bounds by either using exact estimations or upper bounds for initial sizes and lower bounds for the current sizes. Table B.2 shows detailed numbers. Omitted numbers indicate that these were not necessary for estimations.

For eCos, growth rate is based on the difference between version 1.1 from 1999 and 3.0 from 2009; for Linux between version 0.0.1 from 1991 and 2.6.32 from 2009; for Debian between version 1.1 from 1996 and 6.0 from 2011; for Eclipse between version 1.0 from 2001 and 3.6 from 2010; and for Android between version 1.0 from 2008 and 4.0 from 2012. Note that for Android, the free market growth is the main driver, marginalizing main platform growth.

---

**B.2. Tools**

We developed significant infrastructure to quantitatively analyze our three open platforms, while we built upon LVAT and CDLTools (see Appendix A) for the closed platform eCos and Linux. The Appendix ZIP archive contains the Debian script (`tools/debian/stats.py`), the Eclipse tool (`tools/eclipse/`), and the Android static analysis tool (`tools/android/android-analysis.zip`). We also developed scripts in Scala and R to create diagrams and analyze statistics. These scripts are available on request.
B. Software Ecosystem Statistics

B.3. Datasets

We provide both the raw datasets (the input to our extraction tools) and the synthesized datasets (CSV files) that are used to derive statistics and generate diagrams.

B.3.1. Raw Datasets

Debian’s raw dataset are all binary i386 packages from the main component of the 6.0 distribution, represented by the package indices datasets/debian/Packages.

For Eclipse, we used the largest edition (Modeling Tools) of Helios SR1\(^{14}\) together with all bundles from the Helios update site\(^{15}\): datasets/eclipse/eclipse-manifests.zip.

For Android, we preferred the largest repository, the official Google Play store, over many smaller ones\(^{16}\). We used an open source libary\(^{17}\) to download free and not forward-locked (a DRM mechanism) apps iteratively over a period of 14 months. Our raw dataset contains 281,079 unique apps. Due to license issues, we are not allowed to publish the whole Android dataset. It is available on request. However, we provide the mined intents from bytecode, see datasets/android/intents, and intent filters, see datasets/android/intent-filter.

B.3.2. Synthesized Datasets

The output of our tools are CSV files containing abstracted dependency information about all features/basic units in the analyzed ecosystem subsets. These files are instances according to our dependency metamodel introduced in Section 6.5.1.2. Dependencies are over features in eCos/Linux and over units in Debian/Eclipse/Android. Note that we use the term dependency in a general meaning (relationship between entities) and disregard precise semantics (such as hard dependencies versus soft dependencies).

B.3.2.1. CSV Files

We provide two kinds of CSV files for each of our five subjects under datasets/: Relationships.csv contains cardinalities for each association end in the metamodel, and SizesAndDependencies.csv relates these cardinalities to the sizes of basic units or features. We define the columns in these CSV files in Table B.3.

The CSV files use keywords for certain types of dependencies: unit-dependent_units (1), unit-dependent_capabilities (2), unit-depending_units (3), capability-depending_units (4), unit-provided_capabilities (5), and capability-providing_units (6).

Statistics and diagrams derived from the first kind (Relationships.csv) are shown below in Appendix B.5; and from the second (SizesAndDependencies.csv) in Appendix B.6.

\(^{14}\)http://www.eclipse.org/downloads/packages/release/helios/sr1 (build ID 20100917-0705)
\(^{15}\)http://download.eclipse.org/releases/helios
\(^{16}\)http://www.wipconnector.com/appstores
\(^{17}\)http://code.google.com/p/android-market-api
B.4. Static Analysis of Android Bytecode

Table B.3.: Format of CSV files with dependency information

<table>
<thead>
<tr>
<th>Relationships.csv</th>
<th>SizesAndDependencies.csv</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>id</strong></td>
<td><strong>id</strong></td>
</tr>
<tr>
<td><strong>count</strong></td>
<td><strong>count</strong></td>
</tr>
<tr>
<td><strong>type</strong></td>
<td><strong>size</strong></td>
</tr>
<tr>
<td><strong>capability</strong></td>
<td><strong>Number of forward dependencies (1 + 3)</strong></td>
</tr>
<tr>
<td><strong>flag</strong>: is this id a capability or a basic unit</td>
<td><strong>No. of reverse dependencies (2, 4)</strong></td>
</tr>
<tr>
<td><strong>provides</strong></td>
<td><strong>No. of provided capabilities (5)</strong></td>
</tr>
</tbody>
</table>

B.4. Static Analysis of Android Bytecode

This section shares content with the technical note “Static Analysis of App Dependencies in Android Bytecode.”

In this section, we provide details on our static analysis infrastructure that extracts dependencies from Android (Dalvik) bytecode. We describe our implementation, limitations, and how the resulting dataset should be interpreted.

B.4.1. Intent Mechanism

Android apps interact by instantiating data structures called *ints* and throwing them at runtime using certain API methods (see Table B.4). This highly dynamic facility for interaction gives rise to dependencies that are either soft (if the app handles missing targets dynamically) or hard (otherwise). However, Android apps cannot declare such dependencies statically in their manifest; thus, detecting dependencies requires static analysis techniques.

Targets of interactions are individual components of apps (*Activities, Services, Broadcast Receivers, or Content Providers* [anda]), which are described in the manifest. To be open for interaction, components are declared public by either setting an *export* flag or specifying an *intent filter*—*capabilities* (cf. main paper) advertised to the runtime. Intents classify into *explicit* (direct dependency) and *implicit* (capability-based dependency): explicit intents directly target components that have an export flag; implicit intents target capabilities, that is, components that have an intent filter. The runtime resolves implicit intents by matching them against all registered intent filter—requiring user interaction if several match, for example.

B.4.1.1. Intent Resolution

Intent filters declare several *action* and *category* keys, together with a *data* specification (a complete or partial URI) that the corresponding component can handle. Intents contain *action* and *category* keys, and a *data* field (URI). They may also carry extra (but irrelevant for matching) information, such as key-value pairs (*Bundles*) and *flags*. An implicit intent matches an intent filter if its information is a subset of the intent.

\[^{18}\]http://www.informatik.uni-leipzig.de/~berger/tr/2012-dienst.pdf

159
filter’s information. Thus, implicit intents can be seen as a minimal, and intent filters as a maximal specification of app capabilities. If an intent’s \texttt{component} field is set using a fully qualified class name, the intent is explicit and directly targets a concrete component.

Many \texttt{action} and \texttt{category} keys are predefined by the Android API, but in principle, arbitrary values can be used. Such third-party keys have to be documented and published, together with a specification (URI format) of expected data. Community efforts trying to establish intent registries emerged, such as OpenIntents\(^{19}\) (cf. Section 6.5.1.1).

\subsection*{B.4.1.2. Example}

Listing B.1 shows an example of an intent filter adapted from Android’s reference documentation [anda]. It matches all intents instantiated in Listing B.2.

Listing B.1: Intent filter example

\begin{verbatim}
<activity android:name="DemoItemsList"
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <action android:name="android.intent.action.EDIT"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="vnd.android.cursor.dir/vnd.demo.item"/>
    </intent-filter>
</activity>
\end{verbatim}

Listing B.2: Intent instantiations

\begin{verbatim}
Intent imp1 = new Intent("android.intent.action.VIEW");
imp1.addCategory("android.intent.category.DEFAULT");
imp1.setType("vnd.android.cursor.dir/vnd.demo.item");
...
// reference a DataStore that provides data with mimetype vnd.android.cursor .dir/vnd.demo.item
Intent imp2 = new Intent("android.intent.action.EDIT");
imp2.addCategory("android.intent.category.DEFAULT");
imp2.setData(URI.parseUri("content://com.demo.android.provider.DemoItems/ items");
...
// explicitly reference the component
Intent exp = new Intent(this, DemoItemsList.class);
...
\end{verbatim}

\subsection*{B.4.2. Dataflow Analysis}

We implemented an intra-procedural dataflow analysis [Sch, CC77, Cou00] on top of the FindBugs\(^{20}\) [AHM+08] framework. Findbugs is primarily used to find bug patterns in Java code, but provides useful facilities for any kind of static analysis, such as building and traversing a control flow graph (CFG), frame data analysis, or efficient fixed-point

\(^{19}\)http://www.openintents.org
\(^{20}\)http://findbugs.sourceforge.net/
calculations. Since FindBugs expects Java bytecode, we converted the Android (Dalvik) bytecode to Java bytecode first, using dex2jar\(^{21}\).

### B.4.2.1. Java Bytecode Instructions

Our goal is to identify instantiations of the `Intent` class and constructor parameters by tracing them back to constants using constant propagation techniques. The Java Virtual Machine (JVM) works stack-based: each method owns a stack frame, which consists of a (compiler-determined) number of slots. The first (bottom) slot holds the `this` (current object) reference. On top, the method parameters are stored. All other slots are used for local values. Each bytecode instruction may push or pop values on the stack. Thus, to find constant values being used in Intent constructor calls, we need to observe the effects of bytecode instructions on the stack slot values.

To illustrate relevant bytecode instructions, Listing B.3 shows a sequence of instructions that originates from the intent call example in Listing B.4, where the Google Play store is queried.

**Listing B.3: Bytecode of intent creation**

```
    ldc_w #372
    invokestatic #255
    astore_2
    new #171
    dup
    ldc_w #381
    aload_2
    invokespecial #375
    astore_3
    aload_0
    aload_3
    invokevirtual #280
```

**Listing B.4: Java-Sourcecode of a Google Play Store Query**

```java
Uri marketuri = Uri.parse("market://search?q=pname:com.demo.android");
Intent intent = new Intent("android.intent.action.VIEW", marketuri);
this.startActivity(intent);
```

The bytecode instructions in Listing B.3 provoke the following behaviour:

1. **LDC_W**: Load the string constant number \#372 from this class' constant pool and put it on the stack (e.g. `market://search?q=pname:com.demo.android`).
2. **INVOKESTATIC**: Invoke the static (class-scoped) function `android.net.Uri.parse()` with the string on top of the stack as parameter. Put the resulting `android.net.URI` object on the stack. \#255 is a reference to the method signature `<android/net/Uri/parse(Ljava/lang/String;)Landroid/net/Uri;>` in the constant pool of this class.
3. **ASTORE_2**: Pop the current stack value (the `android.net.URI` object) and store in stack slot 2.

\(^{21}\)http://code.google.com/p/dex2jar
4. **NEW**: Create a new instance of the class `android.content.Intent` and push on stack. `#177` references the class `<android/content/Intent>`.
5. **DUP**: Duplicate the topmost stack value.
6. **LDC_W**: Load the string constant number `#381` and push on stack (the value in this example is `android.intent.action.VIEW`).
7. **ALOAD_2**: Load the value from stack slot 2 (the `android.net.URI` object) and push on stack.
8. **INVOKEPECIAL**: Call the constructor of the intent class with two parameters (one string and one URI). Consume those objects as well as the topmost intent reference from the stack. Since the intent reference was duplicated in line 5, we still have a reference on the stack. `#375` is again a constant referencing the method `<android/content/Intent/<init>(Ljava/lang/String;Landroid/net/Uri;)V>`
9. **ASTORE_3**: Store the reference to the intent object into slot 3.
10. **ALOAD_0**: Push the `this` reference on stack.
11. **ALOAD_3**: Load stack slot 3 and push on stack.
12. **INVOKE_VIRTUAL**: Invoke the (possibly overloaded) instance method `startActivity()` on the `this` instance with the intent as parameter value. `#280` references the method `<android/content/Context/startActivity(Landroid/content/Intent;)V>`

### B.4.2.2. Control Flow Graph

Classic dataflow analysis traces the effects of code instructions according to a specific aspect, for example the impact on a variable. The first step is the construction of a CFG, whose nodes are sequences of uninterruptible code instructions (basic blocks), and whose edges represent any possible control flow between basic blocks (normal transfer of control, but also stack unravelling exceptions).

To analyze the value of a variable based on constant propagation, the following functions have to be implemented to emulate the effects of basic blocks and of incoming edges in the CFG:

1. Getting the initial value of the variable
2. Simulating the effect of the basic block on the variable
3. Merging multiple variables, e.g. for loops

To obtain the variable values, a fixed-point algorithm traverses the CFG and applies the functions to each basic block until the variables (stack frame slots) do not change anymore. In this case, the stack frame slots contain the final variable values—given that they could be propagated from a constant.

### B.4.3. Implementation

The variables we are interested in are the stack frame slots that are used as arguments for the constructor and various setter methods of an Intent instance. These arguments originate from the class’ constant pool through propagation. We implemented the CFG functions as follows:
1. The stackframe on the entry block is empty.
2. We implemented effects of `LDC` and `LDC_W`, which load values from the class' constant pool; `NEW`, which instantiates (Intent) classes; `INVOKE_SPECIAL`, which calls the constructor on instantiation; and `INVOKE_VIRTUAL`, which invokes setter methods. Constant propagation is based on interpreting the `ASTORE_` and `ALOAD_` instructions. We also implemented heuristics for frequent patterns: Target components of explicit intents are often set by calling a class’ `getName()` method; and `android.net.URI.parseUri()` is often used to construct URI instances. Finally, we implemented the effects of the various constructors (varying in terms of arguments) and all setter methods of the `Intent` class.
3. Merging two different stack frames is not implemented, since the effects can be too complex to emulate. Instead, if the two stack frames are different, which means that the loop modified the intent in arbitrary ways, we discard them—a conservative approach that avoids extracting incorrect values.

Table B.4.: Android API methods for intent creation and handling

<table>
<thead>
<tr>
<th>Calls</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.app.Activity startIntentSender()</td>
<td>android.content.pm.PackageManager</td>
</tr>
<tr>
<td>android.app.Activity startActivity()</td>
<td>queryBroadcastReceivers()</td>
</tr>
<tr>
<td>android.app.Activity startActivityForResult()</td>
<td>queryIntentActivities()</td>
</tr>
<tr>
<td>android.app.Activity startActivityFromChild()</td>
<td>queryIntentServices()</td>
</tr>
<tr>
<td>android.app.Activity startService()</td>
<td>getActivityIcon()</td>
</tr>
<tr>
<td>android.app.Activity startServiceIfNeeded()</td>
<td>resolveActivity()</td>
</tr>
<tr>
<td>android.app.Activity stopService()</td>
<td>resolveService()</td>
</tr>
<tr>
<td>android.app.Activity bindService()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity unbindService()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity sendBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity sendOrderedBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity sendStickyBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity sendStickyOrderedBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity startService()</td>
<td></td>
</tr>
<tr>
<td>android.app.Activity registerReceiver()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context stopService()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context bindService()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context unbindService()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context sendBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context sendOrderedBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context sendStickyBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context sendStickyOrderedBroadcast()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context startService()</td>
<td></td>
</tr>
<tr>
<td>android.content.Context registerReceiver()</td>
<td></td>
</tr>
</tbody>
</table>

We run the fixed-point algorithm on the CFG that repeatedly executes all the functions on basic blocks. When a fixed-point is reached, we search for the API methods given in Table B.4. We identify the intent instance among the method arguments and log all derived field values of this instance.
B. Software Ecosystem Statistics

B.4.3.1. Counting Dependencies

We now describe how we map the mined intent and intent filter information maps to our metamodel (Section 6.5.1.2). We calculated numbers for each association end as follows; assuring that our implementation leads to lower bounds of the actual numbers:

1. **Direct dependencies** are represented by explicit intents—intent objects whose component field is set to a fully qualified class name. We count all unique explicit intents that have such a value and were found by our analysis. We implemented two heuristics to assure that these explicit intents actually target components of another instead of the same app. We omit intents if the target class is declared as a component in the own manifest file, or a physical class with the same name exists in the same app. These heuristics are justified by corner cases we found, for example, intents targeting internal components that are not declared in the manifest file.

2. For the reverse direction of direct dependencies, we used the same procedure as for 1.

3. **Dependencies on capabilities** are represented by implicit intents. We count all unique (according to their values) implicit intents for which we could at least identify the action value. We implemented one heuristic to omit internal intents: we exclude intents with action keys that are defined in the own manifest file. However, we found corner cases where intent filters are created at runtime, which is possible via the Android API. Thus, internal intents could have been accidentally counted. We consider this issue a threat to validity, but believe that this case is extremely rare.

4. For the reverse direction of dependencies to capabilities, we cannot give numbers. This calculation requires emulating the complex (and hidden in Android code) matching algorithm for intents. Further, the emulated matching would likely be inaccurate, given that our extractions might be incomplete and represent lower bounds.

5. For the number of capabilities provided per app, we count their unique intent filters. This measure is an approximation of capabilities (but still a lower bound), since intent filters can subsume other intent filters. Essentially, intent filters represent a partial order with regard to subsumption (intent filters can be specializations of others).

6. For the number of apps providing a certain capability (i.e. the reverse direction of 5), we counted the apps that declare each unique intent filter.

B.4.3.2. Limitations

The intra-procedural character of our dataflow analysis imposes two conceptual limitations. First, we only catch intents that are both constructed and passed to an API call within the same method. Second, we cannot derive values that are the result of method calls. Our analysis only resolves (propagated) string constants, which can lead to missing
values, for example, when the action key is constructed algorithmically or obtained from a configuration service. However, we implemented some heuristics to deal with common string construction patterns, see Section B.4.3.

Further, we cannot give reliable numbers for reverse dependencies, since we would have to emulate Android’s intent matching algorithm (cf. Section B.4.3.1, association end 3). Given these limitations, we roughly estimated our recall by identifying all intent constructors calls in bytecode (NEW) and comparing them with those constructor calls for which our analysis identified at least the action key value. This recall is approximately 60%.

B.5. Dependency Structures

We now provide detailed supporting statistics about dependencies according to our metamodel from Section 6.5.1.2.

B.5.1. Dependency Type Histograms

For each unit/feature and capability, we calculate how many dependencies of a particular type (association roles) originate from it. These numbers are visualized in the following histograms. For brevity, we combine forward dependencies (1 and 3) in one histogram; as well as reverse dependencies (2 and 4). Both data series are stacked on top of each other. The x-axis is limited to 30.

B.5.1.1. eCos

![Forward Dependencies (1) and (3)](image)

![Reverse Dependencies (2) and (4)](image)

![Provided Capabilities by Units/Features (5)](image)

![Units/Features Providing a Capability (6)](image)
B. Software Ecosystem Statistics

B.5.1.2. Linux

Forward Dependencies (1) and (3)

Reverse Dependencies (2) and (4)

B.5.1.3. Debian

Forward Dependencies (1) and (3)

Reverse Dependencies (2) and (4)

Provided Capabilities by Units/Features (5)

Units/Features Providing a Capability (6)

B.5.1.4. Eclipse

Forward Dependencies (1) and (3)

Reverse Dependencies (2) and (4)

Provided Capabilities by Units/Features (5)

Units/Features Providing a Capability (6)
B.5.1.5. Android

Currently, we cannot provide reliable numbers on reverse dependencies $②$ and $④$ for Android due to limitations of our static analysis; please refer to Appendix B.4.3.1, association end $③$.)
B. Software Ecosystem Statistics

B.5.2. Comparisons

B.5.2.1. Dependencies and Types Among Ecosystems

B.5.2.2. Debian Types

To investigate hard and soft dependencies in Debian (motivated by particular outliers), we additionally compare the dependencies according to their concrete package type. Depends, Pre-Depends, Breaks, Conflicts represent hard (enforced by installer), and Recommends, Suggests and Enhances soft (not enforced, e.g. used as defaults or for optimization) dependencies. Further details on semantics and the particular behavior of the installer (Debian package manager), are given in [debb, GBRM*09]. The diagram clearly shows that soft dependencies are dominated by hard dependencies.
B.6. Dependencies and Sizes of Units

Further, for Debian, Eclipse, and Android, we analyze whether correlations might exist between 1) the binary sizes of basic units and 2) their number of dependencies (1, 3), or provided capabilities(5), or reverse dependencies (3). We identify no such correlations, thus, omit these statistics in our dissertation. The measures under each figure are calculated according to Pearson’s product-moment correlation.

Note that the top diagram is overlayed with a jitterplot, as median = lower quartile = upper quartile = 0 for all types except depends. 
B. Software Ecosystem Statistics

B.6.1. Debian

\[ \rho = 0.006604; \text{p-value} = 0.2672; 95\% \text{ conf. interval} : [-0.005062, 0.018267] \]

\[ \rho = 0.001546; \text{p-value} = 0.795; 95\% \text{ conf. interval} : [-0.010119, 0.013211] \]
B.6. Dependencies and Sizes of Units

\[ \rho = 0.035376; \text{p-value} = 2.754\times 10^{-9}; 95\% \text{ conf. interval: } [0.023721, 0.047022] \]
B. Software Ecosystem Statistics

B.6.2. Eclipse

\[ \rho = 0.042741; \text{p-value} = 0.04997; 95\% \text{ conf. interval: } [7.328e-06, 8.532e-02] \]

\[ \rho = 0.032357; \text{p-value} = 0.1379; 95\% \text{ conf. interval: } [-0.010391, 0.074987] \]
B.6. Dependencies and Sizes of Units

\[ \rho = 0.218197; \text{p-value} < 2.2e-16; 95\% \text{ conf. interval} : [0.177115, 0.25852] \]

B.6.3. Android

\[ \rho = 0.0836727; \text{p-value} < 2.2e-16; 95\% \text{ conf. interval} : [0.079258, 0.088093] \]
$\rho = 0.024535$; p-value $< 2.2e-16$; 95% conf. interval: $[0.020089, 0.02898]$
The following survey questionnaire for our study on industrial variability modeling was designed using SurveyGizmo¹ and is currently available online at our project website².

---

**Survey Questionnaire**

**Do you consider variability modeling useful?**

<table>
<thead>
<tr>
<th>Definitely yes</th>
<th>Yes</th>
<th>Neutral</th>
<th>No</th>
<th>Definitely no</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/definitely-yes" alt="Definitely yes" /></td>
<td><img src="https://example.com/yes" alt="Yes" /></td>
<td><img src="https://example.com/neutral" alt="Neutral" /></td>
<td><img src="https://example.com/no" alt="No" /></td>
<td><img src="https://example.com/definitely-no" alt="Definitely no" /></td>
</tr>
</tbody>
</table>

**Which of the following uses of variability modeling are most valuable in your experience?**

Please check all that apply.

- Management of existing variability
- Derivation of products
- Planning of variability
- Product configuration
- Domain modeling
- Marketing feature scoping
- Design/Architecture
- Software deployment
- Requirements specification
- QA/Testing
- Documentation
- Other

---

¹[http://www.surveygizmo.com](http://www.surveygizmo.com)

²[http://gsd.uwaterloo.ca/industrial-variability-modeling](http://gsd.uwaterloo.ca/industrial-variability-modeling)
How have you modeled variability?
Please check all that apply.

- Separate variability models
- Annotation of existing implementation artifacts
- Other:

Which notations have you used to model variability?
Please check all notations that apply.

- Feature model
- Decision model
- UML-based representation
- Goal model
- Domain-specific language (DSL)
- Architecture description language (ADL)
- Aspect-oriented language
- Frame-based representation
- Product matrix
- Spreadsheet
- Configuration facilities of a component framework (e.g. Spring, EJB, OSGI)
- Key/value pairs (e.g. in xml- or text-based configuration or properties files)
- Free-text description
- Other:

Which tools have you used to model variability?
Please check all tools that apply.

- AHEAD Tool Suite from University of Texas
- DOPLER Tool Suite from University of Linz
- dslvariantmanagement (open source)
- FeatureIDE from University of Magdeburg
- GEA RS from BigLever Software
- mbeddr.com (open source)
- Product Configurator from Camos
- Product Modeler from Configit
- Pure::variants from Pure::Systems
- Oracle configurator/modeller
- Siebel configurator from Oracle
- XFeature from P&P Software
- Other open source tools:
- Other commercial tools:
- Home-grown domain-specific tools:
Which of these "units of variability" do your variability models use?
Please check all units that apply.
- Features
- Decisions
- Configuration options
- Variation points
- Calibration parameters
- Other: 

How many units of variability (as specified above) do your models have?
Please make one selection per row.

<table>
<thead>
<tr>
<th>Units of Variability</th>
<th>No model</th>
<th>1 model</th>
<th>2-5 models</th>
<th>&gt; 5 models</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>51-100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101-1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1001-10000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&gt; 10001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Do some of your models have explicitly-modelled feature dependencies (e.g. requires, excludes)?
Please select the percentage of features that have such dependencies (in average).

<table>
<thead>
<tr>
<th>Percentage</th>
<th>0-25%</th>
<th>26-50%</th>
<th>51-75%</th>
<th>76-100%</th>
<th>Don't know</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Your models represent the variability contained in which implementation artifacts?
Please check all artifact types that apply.
- Requirements
- Architecture/design
- Platform
- Components/modules
- Libraries
- Source code (static variability)
- Running product (dynamic variability)
- Test cases
- Documentation
- Other: 

Save and continue survey later
**C. Survey Questionnaire**

Have you experienced complexity problems with variability modeling? If yes, where?
Please check all areas where problems occurred.

- Visualization of models
- Dependency management (e.g. explosion of dependencies)
- Configuration process (e.g. with conflicts during configuration)
- Model evolution
- Traceability
- Other: 

What mechanisms have you employed to combat complexity in variability models?
Please check all mechanisms that apply.

- Decomposition into multiple models
- Hierarchical organization of multiple models
- Some notion of encapsulation/interfaces between multiple models
- Abstraction / simplification of variability (hard restrictions on the level of granularity for representing variations)
- Visualization of models
- View-based editing and visualization
- Automated reasoning tools (e.g. to check consistency, resolve configuration conflicts or propagate choices)
- Other:

To help us setting your answers in context, it would be very helpful if you could give us some information about the introduction of your product lines and their domains.

Which of the following strategies to introduce a product line have you used?
Please check all that apply.

- Product line was developed before any product was derived (pro-active).
- A single product was evolved into a product line (re-active).
- Multiple existing products were re-engineered into a product line (refactorive).
- Any combination of the strategies above
- Other:

What are the application domains of your product lines?
(e.g. automotive, telecommunication, medical...)

- 

178
It would also be very helpful if you could tell us about your roles and experience in product line projects to set your previous answers in context.

What have been your roles in product line projects?

Check all that apply.

- [ ] Developer
- [ ] Modeler
- [ ] Team Leader
- [ ] Project Manager
- [ ] Domain Expert
- [ ] Researcher
- [ ] Product Manager
- [ ] Marketing Expert
- [ ] Other: 

How many years of industrial experience do you have in software product line development?

- [ ] <1 year
- [ ] 1-2 years
- [ ] 3-5 years
- [ ] 5-10 years
- [ ] >10 years
Bibliography


Bibliography


Bibliography


[DRGN07] Deepak Dhungana, Rick Rabiser, Paul Grünbacher, and Thomas Neu-


Bibliography


Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. Software Product Lines in Action: The Best Industrial Practice in Product Line


[SB10] Steven She and Thorsten Berger. Formal semantics of the Kconfig lan-
guage. Technical Note. Available at http://eng.uwaterloo.ca/~shshe/

[SBKM09] Juha Savolainen, Jan Bosch, Juha Kuusela, and Tomi Männistö. Default
values for improved product line management. In Proceedings of the 13th

[SC92] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability
experience with c news. In Proceedings of the Usenix Summer 1992

[SC05] Michael Stonebraker and Ugur Cetintemel. "one size fits all": An idea
whose time has come and gone. In Proceedings of the 21st International
Conference on Data Engineering, ICDE’05, 2005.

[Sch] M.I. Schwartzbach. Lecture notes on static analysis. Basic Research in
Computer Science, University of Aarhus, Denmark. Available at http://
www.brics.dk/~mis.

[Sch87] David A. Schmidt. Denotational Semantics: A Methodology for
0205089747.

[Sch10] Klaus Schmid. Variability modeling for distributed development - a com-
parison with established practice. In Proceedings of the 14th International

[SD07] Marco Sinnema and Sybren Deelstra. Classifying variability modeling

[SDNB04] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. Covamof:
A framework for modeling variability in software product families. In
Robert Nord, editor, Software Product Lines, volume 3154 of Lecture
Notes in Computer Science, pages 25–27. Springer Berlin / Heidelberg,

[SG09] Reinhard Stoiber and Martin Glinz. Modeling and managing tacit
product line requirements knowledge. In Proceedings of the 2009
Second International Workshop on Managing Requirements Knowledge,
MARK’09, 2009.

[SGB+12] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and
Antonio Ruiz-Cortés. Betty: benchmarking and testing on the automated
analysis of feature models. In Proceedings of the Sixth International
Workshop on Variability Modeling of Software-Intensive Systems, Va-
Bibliography


Bibliography


Bibliography


# List of Figures

1.1. Automotive example: car configurator ........................................ 3
1.2. The Google Play Store as the center of the Android ecosystem .... 4
1.3. Open innovation with software ecosystems ................................. 5
1.4. Empirical journey from software product lines to software ecosystems 9

2.1. Domain and application engineering in SPLE ............................ 18
2.2. BAPO: concerns affecting SPLE ............................................ 19
2.3. Problem and solution space .................................................. 20
2.4. Variability-enabled architectures of Linux, eCos, and FreeBSD .... 21
2.5. Feature specification and code mapping in FreeBSD ................... 22
2.6. Feature model of the JFFS2 filesystem (excerpt) ....................... 26
2.7. Multiple groups under one feature, not allowed in our syntax ..... 29
2.8. Metamodel of feature models .............................................. 30
2.9. Feature model genealogy .................................................... 31
2.10. Feature model repository S.P.L.O.T. .................................... 33
2.11. Excerpt of a Debian manifest file ...................................... 35

4.1. Formal semantics development ............................................ 48
4.2. Configurators of CDL and Kconfig ...................................... 49
4.3. Model excerpts expressed in CDL and Kconfig .......................... 51
4.4. Feature symbols referenced in code .................................... 52
4.5. XOR group in ConfigTool .................................................. 57
4.6. Kconfig feature excluding its parent .................................... 68

5.1. Analysis infrastructure: CDLTools and LVAT ........................... 80
5.2. Feature representation ..................................................... 89
5.3. Summarized Freetz and Linux hierarchies ............................... 92
5.4. Summarized eCos model hierarchy ...................................... 94
5.5. eCos architecture .......................................................... 95
5.6. Hierarchy plots of the three smallest models ........................... 98
5.7. Model hierarchy and shape characteristics .............................. 99
5.8. Dependencies per feature .................................................. 103
5.9. Mean feature dependencies per model .................................. 104
5.10. Embedded for loops in a CDL model ................................... 106

6.1. Dependency metamodel ..................................................... 123
6.2. Dependencies per feature or basic unit .................................. 125
List of Figures

7.1. Conceptual framework: overview of ecosystem organization and variability mechanisms ......................................................... 130
7.2. Benefit of applying variability modeling .................................................. 140
7.3. Notations used to specify variability .................................................. 140
7.4. Variability modeling tools used .................................................. 141
7.5. Reported complexity problems .................................................. 142
7.6. Reported strategies to cope with complexity problems ......................... 142

A.1. Presence condition characteristics .................................................. 152
List of Tables

2.1. Intuitive translational semantics for propositional feature models . . . . 27
2.2. Common reasoning tasks on feature models . . . . . . . . . . . 28

4.1. Mapping of concepts between Kconfig, CDL, and feature modeling . . . . 54
4.2. Invariants between configuration spaces . . . . . . . . . . . . . . . . . . 64

5.1. Model analysis case studies . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.2. Themes of features in the models . . . . . . . . . . . . . . . . . . . . . . . 86
5.3. Grouping statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.4. Feature representation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.5. Percentage of features with constraints and CTCR metric . . . . . . . . 101
5.6. Model metrics used for the quantitative analysis . . . . . . . . . . . . . . 108

6.1. Ecosystem domains and organization. . . . . . . . . . . . . . . . . . . . . 113
6.2. Estimated scales and growth rates of ecosystems . . . . . . . . . . . . . . 116
6.3. Variability mechanisms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.4. Dependency mechanisms . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.5. Dependency statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

7.1. Conceptual framework: overview . . . . . . . . . . . . . . . . . . . . . . . 131
7.2. Derived high-level guidelines for language and tool design . . . . . . . . 135
7.3. Scales of variability models . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

B.1. eCos free market packages . . . . . . . . . . . . . . . . . . . . . . . . . . 156
B.2. Ecosystem growth rate estimations . . . . . . . . . . . . . . . . . . . . . . 157
B.3. Format of CSV files with dependency information . . . . . . . . . . . . . . 159
B.4. Android API methods for intent creation and handling . . . . . . . . . . 163
Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtliche Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialien oder erbrachten Dienstleistungen als solche gekennzeichnet.

....................................................
(Ort, Datum)

....................................................
(Unterschrift)