Master Thesis

August 20, 2017

Microservices-Based Feature Models

Using Heimdall for Correctness Checking and Configuration Validation

Oliver Leumann

of Birwinken TG, Switzerland (05-915-368)

supervised by

Prof. Dr. Harald C. Gall

Gerald Schermann

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Microservices-Based Feature Models

Using Heimdall for Correctness Checking and Configuration Validation

Oliver Leumann

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Oliver Leumann, oliver.leumann@uzh.ch

Project period: 21.02.2017 - 21.08.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Prof. Dr. Harald C. Gall for giving me the opportunity to write this master thesis at the software evolution and architecture lab at the University of Zurich. I would also like to thank Dr. Philipp Leitner for introducing me to Gerald Schermann, my advisor for this thesis. My deepest gratitude goes to him for his inputs and guidance in all of our meetings and who put a lot of effort into the countless review sessions in the end phase of the project. Moreover, I would like to thank Emanuell Stöckli, Christian Davatz, Silvan Troxler, and Alexander Mülli who helped me a lot with the general aspects of writing a master thesis (and how to correctly submit it). I would also like to thank my band members from Dawn Driven, Bigu, Mado and Adrian who ensured that I never lost contact with the rock'n'roll part of my life during the project. Further, I would like to thank Romeo Arpagaus, my team leader at Haufe-umantis AG in St.Gallen, who is not only a great inspiration on how to successfully navigate through the most chaotic workdays, but also always supports any education-related initiative. Finally, special recognition goes out to my parents, Päivi and Andy, as well as to my twin brother Benjamin and my sister Elina for always encouraging and supporting me over the course of my studies. Last but not least, I would like to thank my dearest girlfriend Stephanie for motivating and inspiring me.

Abstract

The microservices architectural style is quickly becoming the standard for designing continuously deployed software applications. The concept of small, independently deployable services complements well with today's possibilities that we have with cloud computing and modern DevOps practices and all of these trends allow for faster time-to-market cycles. Not only does this enable to get earlier feedback from customers, but it also facilitates faster detection of new runtime faults, performance regressions, or changes in business metrics. However, none of these trends are silver bullets and building, deploying, and maintaining such systems can become quite complex in environments with a high release frequency. Keeping track of all the currently existing microservices of an application, the possible multiple versions that exist for the services, and the dependency structures between them, can become hard to maintain manually. Service combinations can have compatibility issues due to services that explicitly require (or exclude) specific versions of a certain service and resolving defective service dependencies to ensure valid service configurations can quickly exceed manual maintenance capabilities. In this paper, we map software product line concepts to the microservices domain in order to introduce a formal model for microservices-based feature models. We present Heimdall as a prototypical Node.js based application that allows software and DevOps engineers to define microservices applications and their dependencies as feature models. Automated analysis techniques adopted from the software product line area enables correctness checking of complex microservices-based applications, the derivation and validation of service configurations, and the recommendation of fixes for invalid service configurations. All these methods are based on satisfiability techniques and a quantitative evaluation of the prototype shows that most of the methods can be performed with a promising performance, even for microservices-based feature models with hundreds of different services.

Zusammenfassung

Der Microservices Architekturstil wird immer mehr zum Standard beim Entwerfen von kontinuerlich eingesetzten Software Anwendungen. Das Konzept kleiner, unabhängig voneinander veröffentlichbarer Services ergänzt sich hervorragend mit den heutigen Mittel die uns Dank Cloud Computing und modernen DevOps Praktiken zur Verfügung stehen und all diese Trends ermöglichen schnellere Produkteinführungszyklen. Damit können nicht nur früher Rückmeldungen von Kunden entgegengenommen werden, sondern auch neue Laufzeitfehler, Performanceeinbussen und Änderungen in Geschäftskennzahlen können schneller festgestellt werden. Keiner dieser Trends ist jedoch eine Wunderwaffe und das Erstellen, Einsetzen und Warten solcher Systeme kann schnell sehr komplex werden, vorallem in Umgebungen mit regelmässigen Releases. Das Bewahren des Überblicks über all die momentan existierenden Microservices einer Anwendung, die verschiedenen Versionen der Services und deren Abhängigkeitsstrukturen untereinander ist nicht einfach und schwer durch manuelle Wartung zu bewältigen. Service Kombinationen können Kompatibilitätsprobleme aufweisen wenn Services explizit eine spezifische Versionen eines bestimmten Services benötigen (oder ausschliessen) und das Auflösen von fehlerhaften Service Konfigurationen kann schnell das Vermögen Manueller Wartung überschreiten. In dieser Abhandlung bilden wir Konzepte von Software Product Lines auf die Domäne von Microservices ab um ein Formales Model von Microservice-basierten Feature Modellen einzuführen. Wir präsentieren Heimdall als Node.js-basierte Prototyp-Anwendung welche Software und DevOps Ingenieure dazu befähigt Microservices Anwendungen und ihre Abhängigkeiten als Feature Modelle zu definieren. Automatisierte Analyse Techniken, welche vom Software Product Line Bereich übernommen werden, ermöglichen das Überprüfen der Korrektheit von komplexen Microservices-basierten Anwendungen, das Ableiten und Validieren von Service Konfigurationen und das Empfehlen von möglichen Lösungen für ungültige Konfigurationen. All diese Methoden basieren auf Satisfiability Techniken und eine quantitative Evaluierung des Prototypen zeigt auf, dass die meisten dieser Methoden mit einer vielversprechenden Performance ausgeführt werden könnnen, auch für Microservices-basierte Feature Modelle mit hunderten von verschiedenen Services.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

The notion of microservices [New15a] solely exists since a few years, but nonetheless this particular service-oriented architectural style has been able to set a new milestone in the software architecture domain. It describes a way to build software applications as a set of small, independently releaseable services which are using lightweight communication methods and each of them is running in its own process [FL14]. The microservices architectural style is quickly becoming the standard for designing continuously deployed software applications [ZBCS16] since the concept of these small independent services complements well with today's possibilities that we have with cloud computing [AFG+10] and modern DevOps practices [HF10, Wol17]. One common denominator that these trends share is that they allow for faster releases of new features [SSLG16]. Instead of waiting several weeks, months or even years for new releases, new code can be deployed to production in intervals of days or even hours. Probably the most anticipated advantages that companies desire from these faster time-to-market cycles is to have earlier customer feedback as well as better and faster detection of new runtime faults, performance regressions, or changes in business metrics (e.g. conversion rate). For this purpose, live experimentation practices such as canary releases, gradual rollouts, A/B testing, or dark launches can be applied to gather the desired measurements.

Unfortunately, despite all the advantages and promising applications that microservices, cloud computing, and modern DevOps practices can offer, none of them are silver bullets. There exist a lot of pitfalls that may come up, since new layers of complexity are quite certain to be added when designing, building, maintaining and deploying such systems [New15a]. Especially in environments with a drastically increased release frequency, it can become quite cumbersome to keep track of all the currently existing microservices for which possibly several versions might be available [SSLG16]. Furthermore, not all service combinations are compatible since some microservices might explicitly require (or even exclude) certain microservices of a specific version. Manually managing the dependencies between the services of microservices-based applications might be feasible for a small number of services, but the complexity increases drastically with every service that gets added to an application. The number of potential possible combinations of microservices with multiple existing versions can quickly exceed the capabilities of software and DevOps engineers to manually ensure valid global service configurations.

1.1 Motivation and Research Questions

Our approach to deal with this problem relies on the use of software product line concepts [Nor02], especially on feature models [Bat05] which are utilized to model variability in product lines. We map these concepts to the domain of microservice-based applications in order to model the variability of microservices, their corresponding versions and the different constraints

between them with the help of feature models. We continue by applying already existing practices to translate such feature models to propositional formulas. The set of translation rules that is required to do this defines the formal model of microservices-based feature models. Having propositional formulas at hand eventually enables the usage of off-the-shelf satisfiability solvers and these solvers allow us to debug feature models and validate feature configurations for these models [MWC09]. Therefore, the first research question is formulated as follows:

Research Question 1

How can software product line concepts be mapped to the domain of microservice applications to enable automated analysis of microservices-based feature models?

Automated analysis of feature models comprises correctness checking and configuration support and both of these concepts can also be applied in the domain of microservices in order to validate the modelled dependencies of microservice applications. Eventually, the formal model and the automated analysis concepts are used as a basis for the HEIMDALL application. The prototype system allows software and DevOps engineers to define feature models for microservices-based applications and to perform automated correctness checking on these models. It also enables configuration support to interactively describe valid service configurations. Both of these automated analysis concepts are realized with the help of an existing satisfiability solver. On the one hand, microservices-based feature models can be validated to verify a model's consistency, to track inconsistency causing elements if any exist, and to detect elements that are not applicable due to specific constraints of the model. On the other hand, the prototype can automatically validate service configurations against their underlying feature models and the solutions provided by the solver are then used to determine whether the service configurations are either legal, incomplete or illegal. In case of illegal configurations, the tool is capable of giving hints on how to resolve non-valid solutions.

Research Question 2

How can we build tooling that is capable of both validating given service configurations and recommending fixes for invalid service configurations based on satisfiability-solution techniques?

1.2 Structure of the Thesis

The remainder of this work is structured as follows. Chapter 2 provides information on microservices, software product lines, feature models and automated analysis of such models. In Chapter 3 we briefly point out related previous work. In Chapter 4 we map the concepts of software product lines to the microservices domain in order to gather a formal model for specifying microservices-based applications and the dependencies between services in feature models. In Chapter 5 we introduce the prototype application and depict the implementation of the automated analysis concepts. After that, chapter 6 evaluates the performance of the automated analysis features and finally, Chapter 7 concludes the paper by summarizing the main learnings and highlights possible future work.

2

Background

Before presenting the reader to the actual mapping of software product line concepts to the microservices domain, we first briefly summarize the core points of those topics that the reader needs to understand to follow the paper. In the first section we introduce the main characteristics of microservices, their relation to DevOps practices, the role of service versioning, and the benefits of using live experimentation techniques. Section 2.2 shows software product line fundamentals and in Section 2.3 we dive into some essential details of feature models. Finally, Section 2.4 presents the translation of feature models to propositional formulas and how satisfiability solvers use them to validate feature models.

2.1 From Microservices to Live Experimentation

The term *microservices* [New15a] is still relatively new. As far it is known, it has been first recorded in may 2011 at a software architect workshop in Venice [FL14] and the participants used it to describe some common architectural styles that they had been observing lately when designing software applications. In the years that followed, a drastically increasing amount of articles and literature have been published to present definitions, principles, recipes and best practices [New15a], patterns and applications [Kra14] as well as implementations of microservices-based applications as described by le et al. [LNS+15]. Probably one of the most famous and still dense definitions has been given by Sam Newman in his popular book "Building Microservices" from 2015 [New15a] and has been carefully extended in his talks [New15b], defining microservices being small autonomous services that work together and that are modelled around a particular business domain. His definition of microservices as well as the principles that he introduced exist nowadays in multiple forms from different authors [DGL+16] [NMMA16], but it is again Sam Newman who describes some important key points about microservices in a very compact way:

> *"They are separate processes, they communicate over network ports, but they are all actually about independent evolution, they are about being able to make a change and deploy them into production by themselves."*
>
> – Sam Newman (2015), [New15b]

2.1.1 Characteristics of Microservices and DevOps

The benefits that the concepts of microservices bring along the way are numerous and varying when applied correctly and with care [New15a]. Technology heterogeneity, better resilience,

independent scalability, enhanced maintainability, composability and replaceability as well as ease of deployment are often listed when advantages of microservices are covered. These benefits complement well with modern DevOps practices such as continuous delivery and deployment [HF10, Wol17]. Microservices are the first architecture developed in the post-continuous delivery era and are essentially meant to be used in environments with fully automated stages in the delivery pipeline [DGL+16]. On the one hand, the enhanced maintainability, composability and replaceability of small services allows performing changes easier and faster on the corresponding code bases of the individual microservices [New15a]. On the other hand, the combination of microservices with fully automated delivery pipeline enables to shorten the time span between committed changes to code repositories and actually deploying these new microservice versions to production [BWZ15]. All in all, microservices and the beforementioned DevOps practices together allow developers to drastically shorten the whole release cycle, with new versions of individual microservices being released with a much higher frequency than ever before.

2.1.2 Service Versioning

The higher the release frequency, the more crucial becomes the managing of existing microservice APIs and available service versions. Newman [New15a] has highlighted a few principles that give some initial aid to cope with the increased complexity of the version landscape of a microservices-based application; Defer breaking changes for as long as possible, catch them as early as possible, use semantic versioning and enable coexistence of endpoints or service versions. The former two are quite self-explanatory whereas the latter two might need some more detailed explanation.

Semantic Versioning. Semantic Versioning[1] (SemVer) is a version notation specification that dictates how version numbers of software packages are assigned and incremented [PW17]. It can be used for immediate judgment whether a consuming service should break or not because of changes to the API of a required producer [New15a]. Each service number is notated in the form of *MAJOR.MINOR.PATCH* whereas respective changes to the three different parts reflect the severity that updates to a producer mean for consuming services. Usually a change to *PATCH* states that bug fixes have been made to existing functionality and changes to *MINOR* indicate that even though new functionality has been added, backward compatibility is preserved. If the latter is not guaranteed, the *MAJOR* part is incremented. One popular example of software using SemVer is NPM[2]. NPM is the most famous package manager for JavaScript and the world's largest software registry [TM17]. It uses SemVer to resolve dependencies between the different JavaScript packages that are used by a software application.

Coexistence of Endpoints or Service Versions. The main problem to solve here is to avoid forcing consumers to upgrade in lock-step when breaking changes are introduced by a producer. The core idea is that both, the old and the new interface, coexist for a certain period in time. Two main approaches exist; Either both endpoints are incorporated in the same running service or the different interfaces are available through different versions on concurrently existing instances of the particular service. As soon as the old interface is not required anymore the interface can be either removed by a new release of the service or, in case of the second approach, by removing the instance or process that is running old version.

[1] http://semver.org/
[2] https://www.npmjs.com/

2.1.3 Live Experimentation Techniques

Shorter release cycles do not only allow for faster innovation, they also allow for earlier customer feedback and insights on how new releases of new service versions are received [SSLG16]. The microservices architectural style already makes it possible to run multiple instances per service at the same time, but newly introduced live experimentation methods try to adopt the previously mentioned co-existence of several different versions of the same service. These live experimentation techniques have the advantage to first introduce new service versions only to a subset of customers (e.g. canary release) or to introduce and compare two different new versions in parallel (A/B testing), before rolling out new changes to the entire customer base. The eventual goal of having earlier feedback about newly launched service versions is to have detailed metrics about potential growth in runtime faults, declining performance, or changes in business metrics. These metrics can then be used to evaluate the impact of specific releases and in case of possible deterioration, suitable fixes need to be scheduled, such as rollbacks or hotfixes.

2.2 Software Product Line Tenets

In order to cope with the prevalently customer-driven software domain, many software companies try to satisfy the diverse customer needs by creating so called software product lines (SPLs) [Nor02]. Generally, a product line contains a set of similar products that reuse and share some common features and functions in order to achieve economics of production. Product lines are not new in manufacturing and applied by popular companies like Ford, Dell, Boeing, and McDonald's, but adapting these principles and practices to the software domain is relatively new. The idea of building software products from common artifacts aims to sufficiently meet today's demands of software mass customization.

> *"A software product line is a set of software-intensive systems that share a common, managed feature set satisfying a particular market segment's specific needs or mission and that are developed from a common set of core assets in a prescribed way."*
>
> – Linda Northrop (2002), [Nor02]

2.2.1 Core Assets

The so called core assets form the foundation of SPLs and represent the common artifacts that the different systems share with each other [Nor02]. In the software domain, typical examples for reusable core assets are [McG04]:

- Frameworks
- Libraries
- Components
- Tools
- Development or execution platforms

For each such asset, a process, guide or plan is attached that defines how it can be optimally used for building a desired product. Additionally, core assets can also include domain models, requirements, documentations, specifications, test plans, performance models, budgets and other complementary aspects that help describing how the asset can be used for a product [Nor02]. Generally, there exist two ways to acquire core assets [McG04]. The first option is that existing assets from other products are copied and if necessary modified and extended to the requirements

of the new product. However, in case that the requirements can not be tackled with existing means, a new core asset needs to be build up from scratch. The latter is usually more expensive and extensively building brand new assets thus needs to be avoided. After all, maintaining products that only share a little percentage of the product line's asset base outweighs the advantages of SPLs and fundamentally contradicts against its core idea. On the other hand, if managed correctly, companies that use SPL methods to create and maintain software will be able to dramatically reduce costs, improve software quality and drastically speed up the time to market for new applications.

2.2.2 Shared Architecture of Core Assets

In a specific product line, each core asset shares an architecture that all products have in common [McG04]. Typically, the architecture used for the product line is built from the quality-attribute requirements, e.g. performance, scalability, and security. The quality attributes need to be identified when establishing the new product line. Changes to the architecture are allowed since new products may have different requirements than the products already integrated in the product line. An architecture change might be needed for example if a new product needs to be scalable to millions of users instead of the few thousands that are currently supported by the existing products. Yet it must be considered that the new product might be not a candidate to be included in the product line in case the architectural adjustments requires too rigorous changes.

2.3 Feature Models

In the previous section we have briefly introduced SPLs and also described the importance of reusable core assets. However, in order to be able to develop such reusable assets, commonalities as well as variabilities of the different products need to be identified [LKL02]. A central technique to put SPL engineering into practice is variability modelling [PBvDL05] and feature modelling has become a very popular modelling notation to serve this particular purpose [MWC09].

> *"Feature modeling is the activity of identifying externally visible characteristics of products in a domain and organizing them into a model called a feature model"*
>
> – Lee et al. (2002), [LKL02]

The notion of features is used to represent the mutual or differing externally visible characteristics that differentiate the products from each other from a stakeholders perspective [LKL02]. The term "feature" is chosen by purpose since it is the general case that engineers and customers tend to speak of product characteristics with respect to features that a particular product has and/or delivers. A feature configuration is a particular set of features whereas a product of the SPL comprises a feature only if the feature is in its feature configuration [Bat05]. A feature configuration can be either compliant to the constraints of a feature model and represents a legal product or it violates the constraints and is therefore illegal [MWC09].

2.3.1 Structural Relationships

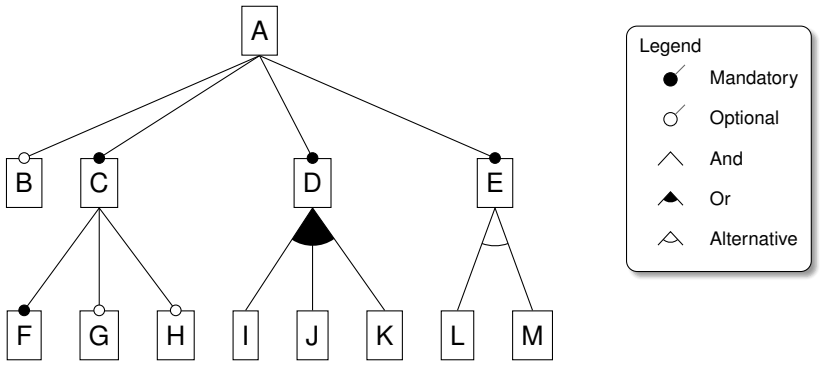
A feature model is a hierarchically arranged set of features and feature diagrams can be used to graphically represent these feature models and to capture structural or conceptual relationships among the different features of a product line. The starting point of the hierarchy is given by the so called root feature which represents the domain concept being modelled, the actual SPL.

The relationships of parent features to child features can be categorized into five different types [LKL02] [Bat05] as shown in Table 2.1.

Categories	Description
Mandatory	The feature is required; This is needed to model common features among different products of a product line.
Optional	The feature is optional; This allows to model variability for specific single features among the products.
And	The subfeatures must be selected with respect to the mandatory or optional indicators; This is used to model a set of mandatory and/or optional subfeatures.
Or	At least one of the subfeatures is required; This enables to model a set of optional subfeatures where at least one needs to be selected.
Alternative	Exactly one of the subfeatures needs to be selected; This is useful to model mutually exclusive subfeatures and thus needed to model features that can substitute each other.

Table 2.1: Types of Structural Relationships Between Features

Figure 2.1 shows an example of a feature model representing a graphical hierarchy of features with relationships between the features according to the five types described in Table 2.1. In this hierarchical graph, features are represented by nodes and the relationships between features are represented by edges. The feature *A* is the root feature and comprises the subfeatures *B*, *C*, *D*, and *E* whereas the first one is an optional feature and the latter three features are mandatory among all products. Feature *C* requires feature *F* but subfeatures *G* and *H* are optional for the products of this exemplary SPL. Feature *D* requires at least one of the features *I*, *J*, and *K* to be present whereas feature *E* requires that exactly only one of its subfeatures, either *L* or *M*, exists in each of the products of the current model.



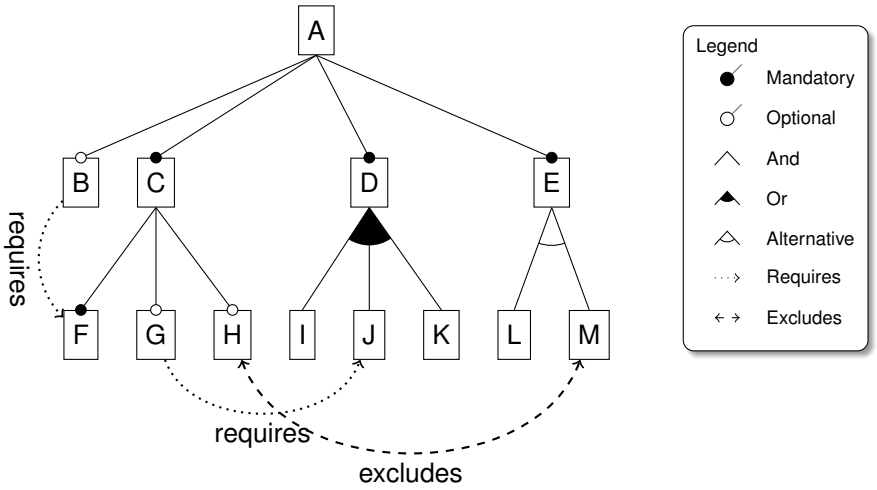
$$(B \rightarrow F) \ \wedge \ (G \rightarrow J) \ \wedge \ \neg(H \wedge M) \ \wedge \ (G \wedge H \rightarrow \neg L)$$

Figure 2.1: Constraints in Feature Models

2.3.2 Cross-Tree Constraints

In order to further refine the constraints among the features [MWC09], additional complementary constraints can be defined to the basic structural relationships between features in feature models. These additional constraints are called *cross-tree constraints* since they are not restricted to being solely used for modelling parent-child relations, but they also allow the modelling of arbitrary constraints between any nodes of a model. Eventually, the set of all relations of a feature model is defined by conjoining all the relations of its hierarchical graph of features with the conjunction of its cross-tree constrains. Notice that parent-child relations themselves do not reveal inconsistencies. Only the usage of cross-tree constraints may introduce contradictions [TBC06] because of badly defined relationships among features. Cross-tree constraints make it necessary to check for correctness of feature models, which is explained later on.

As shown in Figure 2.1, cross-tree constraints are typically listed at the bottom of the actual feature diagram. In this particular example we have the definitions of four cross-tree constraints. The first constraint $B \rightarrow F$ depicts the fact that the optional feature B requires feature F and the second constraint $G \rightarrow J$ illustrates that G requires J. The third constraint $\neg(H \wedge M)$ states that H and M exclude each other and the fourth constraint $G \wedge H \rightarrow \neg L$ defines that whenever G and H get selected, feature L is prohibited. The first three cross-tree constraints are quite common in feature modelling, thus specific graphical notations are available to represent this constraints [BSRC10]. Figure 2.2 shows this alternative graphical notation on the same model as seen in Figure 2.1. The *requires* constraint is represented as a dotted arrow that is drawn from the feature that requires a feature to the feature that it actually requires. The *excludes* constraint is visualized as a dashed line with arrows pointing to the features that exclude each other. The constraint $G \wedge H \rightarrow \neg L$ is still written out as a Boolean formula since no appropriate graphical notation exists for it.



$$(G \wedge H \rightarrow \neg L)$$

Figure 2.2: Modelling Requires and Excludes Constraints in Feature Models

2.3.3 Extended Feature Models

Feature Models have also been extended in such a way that *attributes* can be assigned to features [BMAC05]. These attributes can hold values defined by their *attribute domain* and *extra-functional features* can be specified to establish relations between one or more attributes of a feature. This enables to model complex constraints, e.g. if attribute A of feature X is higher than a value V, then feature B can not be part of the product. Such models are called *extended, advanced, or attributed feature models* [BSRC10], but they are not required for this thesis and therefore won't be discussed any further.

2.4 Automated Analysis of Feature Models

When modeling SPLs using feature models, it is important to check for the correctness of the model [TBC06]. As previously indicated, errors may be introduced because of badly defined constraints among features. Also during the product derivation process [MWC09], illegal combinations of features are easily possible. Often, feature models are becoming quickly big enough to avoid a manual handling of them [TBC06], therefore an automation of analyzing feature models is necessary. In order to be able to automatically debug feature models and to validate whether feature configurations comply with the defined constraints of a given feature model, specific translation rules have been introduced to reduce a feature model to its propositional formula representation [BBRC06]. This has opened the possibility to make use of logic-based systems, including off-the-shelf tools such as satisfiability (SAT) solvers [MWC09] whose underlying representational formalism is propositional logic [GKSS08]. Essentially, these tools are capable of providing generic combinatorial reasoning, but their maximum potential is exploited when using them in domains that are not normally viewed as propositional reasoning tasks, in this case, analyzing feature models.

2.4.1 Categories in Feature Model Analysis

Existing analysis of feature models fall into two main categories: correctness checking and configuration support [MWC09]. The former focuses solely on debugging a feature model itself, whereas the latter involves to additionally incorporate a given feature configuration to the feature model analysis.

Correctness Checking. Correctness checking is used to analyze a feature model regarding its consistency or if dead features exist. A feature model is regarded as inconsistent if no single valid feature configuration can be established. Dead features on the other hand, are features that do not belong to any legal product. Such features can not be part of a feature configuration without violating constraints of the feature model. Both of these problem types can be resolved by applying SAT solvers correspondingly.

Configuration Support. Configuration Support represents the second category of automatic analysis with SAT solvers and it is used to support the derivation process of products. Thus, SAT solvers are not only able to compute whether a certain feature configuration is legal or illegal, but they also support two distinct types of guided product configuration techniques. The first type is termed *interactive* configuration where the configuration system guides users with consistent choices during the decision process by validating and propagating the user's decision. Eventually, this ensures that the user always has a legal product configured at the end of a product configuration. The second product configuration type is called *offline* or *batch* configuration

and it automatically tries to complete a partial configuration without requiring any intermediate interaction from the user.

2.4.2 Translation to Propositional Formulas

Translating feature models to propositional formulas have been subject of thorough research [Man02] [Bat05] [CW07] and an accumulated list of translation rules is shown in Table 2.2, as derived from Mendonca et al. [MWC09] and Benavides et al. [BSRC10]. Formula variables represent the features of the model whereas the *root* feature is defined by a simple formula r. The second rule depicts that every *optional* subfeature is defined with an implication to its parent feature and the third table row delineates that a *mandatory* feature is additionally implied by its parent. Benavides et al. [BSRC10] define a *mandatory* feature simply with a logical equivalence operator between the feature and its parent since implications that go in both directions between two variables can be replaced by a logical equivalence. The two rules that follow describe the *or* and *alternative* structural relationships by an implication from the parent feature to the respective cardinality relation [MWC09]. The $[1..n]$ and $[1..1]$ cardinalities of these grouped features simply define the respective ratio between the parent and the subfeatures, i.e. *at least one* subfeature for the *or* grouped features and *exactly one* subfeature for the *alternative* grouped features. The sixth table row depicts the fact that *cross-tree constraints* are generally already Boolean formulas. In Figure 2.2 however, we have shown that specific notations exist for the two popular cross-tree constraints, *requires* and *excludes*, and thus the two last rows of Table 2.2 show the Boolean formulas for the corresponding constraints. The Boolean formula representation of the `excludes` constraint introduced in Figure 2.1 is actually a shorthand notation for one feature prohibiting the other and vice versa, which is also appropriately depicted in the last row of Table 2.2.

Relation	**Feature model context**	**Corresponding formula**
root	r is the root feature	r
optional	c is subfeature of parent p	$c \rightarrow p$
mandatory	m is subfeature of p p is parent feature of m shorthand: m is mandatory subfeature of p	$m \rightarrow p$ $p \rightarrow m$ $m \leftrightarrow p$
or	p is the parent of [1..n] grouped features $g_1, ..., g_n$	$p \rightarrow (g_1 \vee ... \vee g_n)$
alternative	p is parent of [1..1] grouped features $g_1, ..., g_n$	$p \rightarrow 1\text{-}of\text{-}n(g_1, ..., g_n)$
constraints	cross-tree constraints	already Boolean formulas
requires	feature a requires feature b	$a \rightarrow b$
excludes	a prohibites b b prohibites a shorthand: a and b exclude each other	$a \rightarrow \neg b$ $b \rightarrow \neg a$ $\neg(a \wedge b)$

Table 2.2: Feature Models as Boolean Formulas

In order to get a full propositional formula representation for the configuration possibilities of a feature model, the formulas for the corresponding syntactic elements need to be conjoined together with the conjunction of all the cross-tree constraints [BSRC10]. A SAT solver can then take the propositional formula and determine if the formula is satisfiable, i.e. there is a variable

assignment that makes the formula evaluate to true. One predetermined constraint for such assignments is that the variable representing the root gets assigned with *true*, i.e. $r \leftrightarrow true$.

Figure 2.3 represents the appropriate propositional formula of the feature model shown previously in Figure 2.1. The first line (1) defines feature A as the root element and the second line (2) represents its optional feature B as well as its mandatory subfeatures C, D and E. The third line (3) states that regarding feature C, only the subfeature F is mandatory. The fourth line (4) depicts that at least one of feature D's subfeatures needs to be selected and line (5) defines that only either feature L or feature M can be selected as a subfeature for feature E. Finally, the last line (6) demonstrates that cross-tree constraints can be adopted from the feature model either without any changes, or in case of graphical notations for *excludes* and *requires* constraints, with the appropriate translation formulas.

$$
\begin{aligned}
(A) \ \wedge & && (1)\\
(B \rightarrow A) \ \wedge \ (C \leftrightarrow A) \wedge \ (D \leftrightarrow A) \wedge \ (E \leftrightarrow A) \ \wedge & && (2)\\
(F \leftrightarrow C) \ \wedge \ (G \rightarrow C) \wedge \ (H \rightarrow C) \ \wedge & && (3)\\
(I \rightarrow D) \wedge \ (J \rightarrow D) \wedge \ (K \rightarrow D) \ \wedge \ (D \rightarrow (H \vee J \vee K)) \ \wedge & && (4)\\
(K \rightarrow D) \wedge \ (L \rightarrow D) \ \wedge \ (D \rightarrow (K \wedge \neg L) \vee (\neg K \wedge L)) \ \wedge & && (5)\\
(B \rightarrow F) \wedge \ (G \rightarrow J) \wedge \ \neg(H \wedge M) \ \wedge \ (G \wedge H \rightarrow \neg L) & && (6)
\end{aligned}
$$

Figure 2.3: Feature Model as Propositional Formula

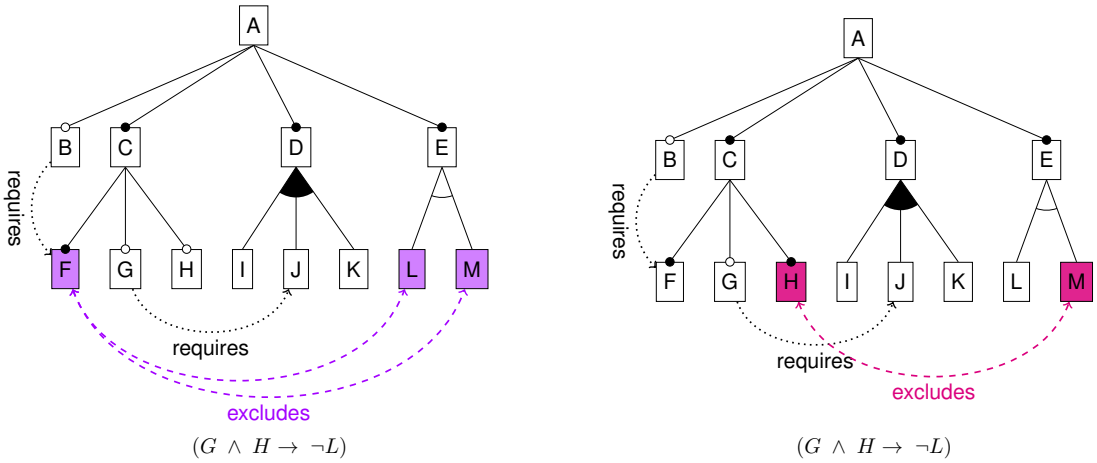
2.4.3 Correctness Checking of Feature Models

In Section 2.4.1 we have introduced the notion of *correctness checking* and that basically two different aspects of feature models can be analyzed with the help of a SAT solver: the *consistency* of Feature models and *dead* features. In Figure 2.4 we show two graphical examples where a SAT solver would detect flaws in feature models in case correctness checking would have been performed on the respective propositional formulas of the feature models.

Inconsistent Feature Models. Figure 2.4(a) shows an example of an inconsistent feature model by slightly adjusting the feature diagram introduced by Figure 2.2. The *excludes* constraint between the features H and M has been removed, but two new *excludes* constraints have been added; One between feature F and feature L, the other between feature F and feature M. A SAT solver is able to detect that this feature model is inconsistent since no legal feature configuration can be computed for it. On the one hand, the feature F is mandatory for each feature configuration of the feature model. On the other hand, the *alternative* relation between Feature E and its child features makes it mandatory that either feature L or feature M are also part of the configuration, but both conflict with the feature A because of the newly added *excludes* constraints. Figure 2.4(a) represents the conflicting elements with purple color.

Dead Features. In contrast to inconsistent feature models, Figure 2.4(b) shows an example with a *dead* feature, again by slightly adjusting the exemplary feature model from Figure 2.2. This time, the feature H is not *optional* anymore, but has been changed to be *mandatory*. Due to the *excludes* constraint between the features H and M, feature M can never be selected for a legal feature configuration. Therefore feature M can not exist in any product and represents a *dead* feature.

Figure 2.4(b) highlights the *dead* feature M and the corresponding *excludes* constraint and the feature H that cause this flaw in the feature model with pink color.



(a) Inconsistent Feature Model

(b) Feature Model with Dead Feature

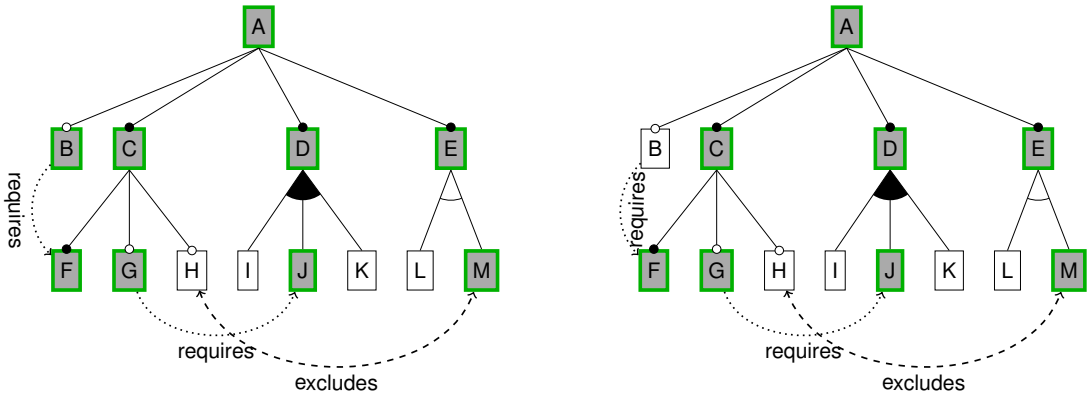
Figure 2.4: Correctness Checking of Feature Models

2.4.4 Validating Feature Configurations

As previously mentioned, a feature configuration is a set of selected features that is used to describe a product in a product line [MWC09]. In terms of the propositional formula representation of feature models, a feature configuration represents the set of variables that get assigned with a Boolean true value. In order to validate whether a given configuration is legal, a SAT solver requires the propositional formula of the feature model as well as the set of variables that have been selected. Thus it is recommended to ensure that the feature model is consistent before performing any configuration validation on it, else the validation can never be successful. The possible results returned by a SAT solver evaluation are implementation specific, but usually some indicators are returned that declare whether a feature configuration is either legal or illegal.

Legal Feature Configurations. In Figure 2.5 we show two examples of legal feature configurations. Figure 2.5(a) shows that the features $\{A, B, C, D, E, F, G, J, M\}$ have been selected, which are highlighted accordingly as gray boxes in the feature model. A SAT solver that validates the given feature configuration against the propositional formula would confirm the configuration to be legal since no constraints are violated by the selected features, thus these features have been accordingly visualized with green borders. Figure 2.5(b) shows that we would still obtain a legal configuration if we would remove feature B from the selection. Feature B is optional and thus no constraints of the feature model are harmed.

Illegal Feature Configurations. Figure 2.6 introduces two examples of feature configurations that are illegal. The first example Figure 2.6(a) shows the impact of selecting feature H instead of feature G, leading to a feature configuration that comprises the features $\{A, C, D, E, F, H, J, M\}$. Since the features H and M conflict each other, the SAT solver would mark the configuration as illegal. Figure 2.6(b) also shows an illegal configuration, but this time too few features have

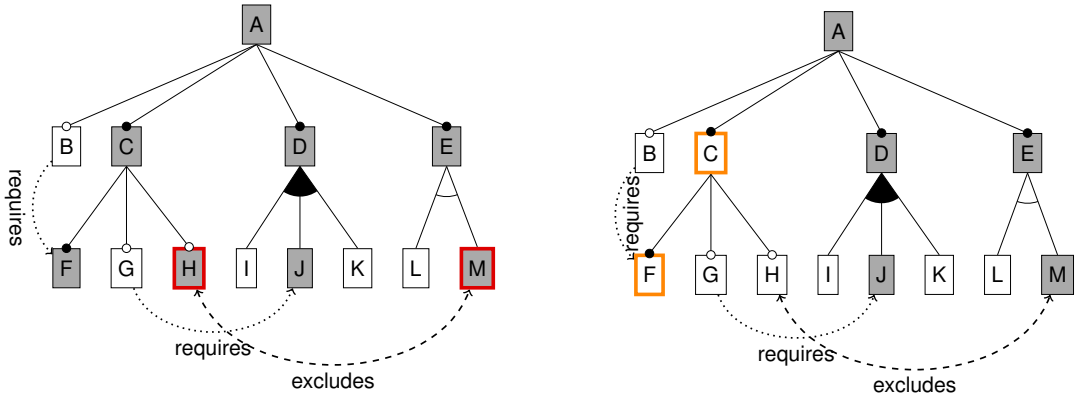


$(\mathrm{G} \wedge H \rightarrow \neg L)$ $(\mathrm{G} \wedge H \rightarrow \neg L)$

(a) (b)

Figure 2.5: Legal Feature Configurations

been selected. The relationship constraints of the feature model requires features C and F to be incorporated into all software products of the given product line. Depending on the implementation and use of SAT solvers, it is possible to evaluate missing features, thus we mark the missing features of this illegal configuration with orange instead of red. Later in Chapter 5 we will see an actual implementation of such a mechanism where configurations with missing features are called *incomplete feature configurations*.



$(G \wedge \mathrm{H} \rightarrow \neg L)$ $(G \wedge H \rightarrow \neg L)$

(a) (b)

Figure 2.6: Illegal and Incomplete Feature Configurations

3

Related Work

As far as we know, using SPL techniques to model and manage dependencies in microservices-based applications haven't been subject of research so far and thus no directly related work exists for this particular topic. Section 3.1 shows some general research that has been recently conducted in the domain of microservices and in Section 3.2 we introduce previous work which have tackled challenges that are related to the task of modelling and managing microservice dependencies. In Section 3.3 we highlight some research that has also adopted SPL practices to overcome challenges in current trends of the software engineering landscape and in Section 3.4 important research regarding automated analysis of feature models is presented.

3.1 Current Research in Microservices

Literature that presents basic definitions, principles, and practices of microservices [Kra14] [New15a] [NMMA16] is likely to remain quite valid for a longer period of time, but research that is trying to facilitate the diverse challenges of microservices is steadily adapting to the increasing findings in the microservice area. Two surveys have been recently published [DGL+16] [DFML17] that try to reflect the current state of research in the microservice domain.

Dragoni et al. [DGL+16] present the past, current state, and future challenges of microservices-based applications. With respect to the past, they emphasize that the microservice architecture is an immediate successor of service-oriented architecture (SOA). Well known names have underpinned this, such as Cockraft [NMMA16] talking about fine grained SOA or Newman [New15b], claiming that the microservices architecture is nothing more and nothing less than an opinionated form of SOA. Regarding the current state, the basic characteristics and benefits of microservices are summarized and the impact on quality attributes, such as availability, reliability, maintainability, performance, security and testability are briefly recapped [DGL+16]. The outlook on future challenges focuses on the problem of microservices being distributed systems and therefore inherently harder to program than monoliths. Two key areas are identified, dependability and security. Dragoni et al. highlight research that address issues in these two topics in order to answer questions such as, how can changes to a producing service with side-effects on the consuming services be managed and how can attacks be prevented that exploit network communications. We hope that the findings of this thesis help to enhance practices of the former aspect.

Francesco et al. [DFML17] apply a systematic mapping study methodology to identify, classify, and evaluate the current state of the art of research on microservices. Publication trends, focus of research, and potential for industrial adoption are considered and used for classification purposes. One example is the research-contribution category that lists the frequency distribution of the outcomes of the investigated papers. Since the microservices trend emerged from the industrial field, "application" leads the ranking ladder of research outcomes. Another example cat-

egory is the analysis of research strategies whereas "solution proposal" is the clear winner. This has been explained with microservices being in its infancy and not consolidated in any standards, tempting many researchers to propose their own solutions for recurrent or specific problems. One further interesting category deals with the most frequent problems that are targeted by current research. The leading terms have been identified to be "complexity", "low flexibility", "resources management", and "service composition". Francesco et al. argue that this results confirm that on the one hand microservices can help in achieving a good level of flexibility but on the other hand may bring higher complexity, mainly because they imply a high number of distributed services to operate. "Service decomposition" addresses the problem of breaking down existing monolithic systems into the prominent small services of microservices-based applications. This is of course a natural consequence of having in the past few years a lot of software applications becoming huge unmaintainable legacy software structures and thus the software industry is trying to resolve this issue.

Actually, a lot of work [AAE16], [KMK16], [LTV16] has been presented in the past years to facilitate breaking down existing monolithic systems in order to enjoy the benefits that come along with the microservices architectural style. Alshuqayran et al. [AAE16] have focused on identifying challenges, the architectural diagrams/views and quality attributes related to microservices systems. Kecskemeti et al. [KMK16] have introduced a methodology for decomposing monolithic services to multiple microservices whereas the methodology applies several outcomes of a real-life project called ENTICE. Similarly, Levcovitz et al. [LTV16] have described a technique to identify and define good candidates to become microservices on a monolithic enterprise system. Despite the hype and popularity of microservices, a multitude of software systems still endure in their monolithic shape. The challenge stays real and recent research persistently tries to further facilitate the process of transforming these big software systems into microservices-based applications [DDLM17], [GT17], [MCL17]. Dragoni et al. [DDLM17] use a real world case study in order to demonstrate how scalability is positively affected by reimplementing a monolithic architecture into microservices using specific techniques. In a similar manner, Gouigoux and Tamzalit [GT17] present technical lessons learned from a migration of a real world monolith to microservices by addressing the granularity, deployment, and the orchestration of microservices. One further very recent, but especially notable and noteworthy research has been introduced by Mazlami et al. [MCL17]. They present a formal microservice extraction model to allow algorithmic recommendation of microservice candidates. Thus, instead of solely relying on informal migration patterns techniques that already exist, a foundation for automated support tools is given and a web-based prototype is provided to demonstrate adequate performance evaluations. Due to the formal model, they are the first to provide a semi-automated approach that covers recommendation of microservices from the monolithic codebase without the need of heavy user input.

3.2 Modelling and Managing Service Dependencies

A lot of research has been carried out for deploying and operationally managing microservices, but there has rarely been strong focus on modelling the dependencies between the services and, above all, to introduce automated analysis of service dependencies before deploying services. We recap on three carefully selected research papers that in some way or the other, try to overcome aspects of this particular problem area.

Since the microservice architecture is an immediate successor of SOA, managing services and their dependencies was already of potential interest for the latter architectural style. One interesting work has been delivered by Ensel et al. [EK02] who describe an approach for managing service dependencies with the combined help of several XML technologies, such as XML, XPath,

and RDF. By relying on these general-purpose technologies it is possible to represent dependency graphs in such a way that they can be parsed by common off the shelf XML parsers. No previous work has dealt with describing dependency information in a uniform way before, so this has been the first time that management systems in general could make use of it. Since no formal model has been provided, this approach is limited to manage dependencies without the possibility for automated analysis and it mainly focused on querying and visualizing the services and their dependencies. Also, even though the use of XML technologies has its advantages, it already prescribes a specific part of the technology stack.

Uhle [UT14] presents in his thesis a way to model the dependability of microservice architectures by using dependency graphs that are transformed to fault trees. For the dependency graphs, vertices represent applications and directed edges indicate the dependencies. As part of the thesis, multiple methods have been introduced on how to construct dependency graphs from a deployed microservice architecture, i.e. manual creation, from service interface modules, from deployment configuration, or from network connections. Dependency graphs have been defined to be directed, acyclic, and rooted in order to be able to be transformed to a fault tree. The main idea is that the fault trees are then used to model failure propagation of microservice architectures in order to obtain an overview of which parts of the architecture are more crucial than other ones. The thesis is limited to already deployed systems and focuses mainly on computing failure propagation probabilities of services. Automated consistency checking and correctness validations of dependencies are neglected since the approach has not been developed from the perspective to ensure valid service configurations before deploying them. A further limitation of this work is that dependency graphs have been investigated on the granularity level of applications and their services, but particular versions of microservices are not taken into account. Nevertheless, the different methods for constructing dependency graphs are also applicable to the prototype system of our thesis. Since there is plenty of research available in this area, we will not further emphasize this aspect in our work.

Schermann et al. [SSLG16] introduced a formal model for multi-phase live experimentation and presented a prototype implementation that allows release engineers to define and automatically enact complex live experimentation strategies on microservices-based applications. Their concept is based on a traffic routing mechanism using lightweight proxy components. The prototype is non-intrusive and thus, it does neither need any feature toggles nor any other code-level changes. However, one key requirement is that requests are correctly forwarded between the various service instances and versions. Thus, the approach relies on the coexistence of service versions as described in Section 2.1.2 and for that the formal model involves modelling the different versions of the available microservices. Release strategies are defined in a domain specific-language and include the configuration of which metrics and thresholds should be considered to help deciding whether a release is successful or needs to be rolled back. The approach of this paper drastically helps to automate experimentation in the scope of microservices-based applications and enables version controlling, sharing, and reusing strategies between changes or even teams. Some of the goals are related to the goals of this thesis. On the one hand, the approach of Schermann et al. deals, among other things, with the problem of automatically validating whether new deployments of services are successful or not. On the other hand, the current thesis has as one of its goals to automatically validate service dependencies in order to ensure that a given service configuration is valid in the first place before it is rolled out in form of an experiment.

3.3 SPLs in Related Application Areas

Instead of using the SPL techniques to model and manage service dependencies, recent research has been conducted by Sousa et al. [SRD16c] [SRD16a] to apply SPL practices in the cloud com-

puting area. Even though the problem domain is different then from the one that we address here in this thesis, it nicely illustrates that SPL practices can be reused and adjusted in multiple ways to overcome challenges in a multitude of application areas.

Sousa et al. [SRD16c] have proposed an approach to automatically build a multi-cloud environment for microservices-based applications while still reasonably handling the heterogeneity and variability of cloud providers and the multi-cloud requirements of microservices-based applications. In detail, the heterogeneity of providers needs to be taken into account in order to deal with the different levels of functionality abstractions that cloud providers offer, such as IaaS, PaaS, and SaaS. Tackling cloud provider variability helps to deal with complex constraints regarding a cloud provider's configurations, e.g. which functionality is provided to which regions and for what price plan. Finally, the multi-cloud requirements need to be addressed in order to cope with the profoundly differing requirements of microservices as well as to consolidate private and public clouds. Sousa et al. rely on the idea to use feature models (FMs) to define the features that are offered by a provider and how these features are related. The application requirements are then matched against these FMs and an automated analysis of these FMs is used to find a valid and complete configuration for a specified set of cloud providers.

In order to be able to model cloud provider variability as FMs, Sousa et al. also needed to extend FMs with relative cardinalities [SRD16b]. Feature cardinalities have been already introduced before [CBUE02] in order to model features that can be selected multiple times for a specific product. However, the approach has been limited in the way that it only allowed feature cardinalities in relation to the parent feature or the global product configuration. Sousa et al. [SRD16b] generalized the existing interpretations of feature cardinalities in such a way that cardinalities can be applied to features relative to any other feature in the feature model.

One of the most recent work by Sousa et al. [SRD17] carefully examines the problematic nature when requirements of a cloud system change and the complex constraints that this change often exhibit. Their goal is to guarantee a safe transition from a valid initial configuration to a target configuration while satisfying any constraints that exist for the transitions. Their previous approach of automatically setting up multi-cloud environments for microservices applications is extended with variability modelling for a dynamic SPL approach by extending FMs with temporal constraints and reconfiguration operations. Temporal constraints are required to define aspects such as when choosing a feature makes another feature unavailable for all further prospective configurations. The reconfiguration operations are needed to allow modelling of multiple reconfiguration paths and their impact on reconfiguration time, costs, and performance.

3.4 Automated Analysis of Feature Models

There exist numerous works in the literature that propose the usage of propositional formulas for the automated analysis of feature models. Benavides et al. [BSRC10] have made a neat summary of the evolution regarding the connection of propositional formulas with feature models. In doing so, Mannion et al. [Man02, MC03] have been identified to be the first that connected propositional formulas with feature models and Batory [Bat05] has been depicted as the first one to use a SAT solver for analysing feature models. Benavides et al. [BSRC10] also have made a thorough listing of the most innovative contributions with respect to automated analysis of feature models. For example, Yan et al. [YZZM09] have been mentioned for proposing an optimization method to reduce the size of the logic representation of feature models by removing irrelevant constraints. The work by Benavides et al. [BSRC10] also comprises a comprehensive list of analysis operations on feature models and a multitude of literature is referenced that tackle one or more of these analysis operations.

With respect to correctness checking as part of the automated analysis, there are some works

in the literature that analyze approaches and possible optimizations of this aspect. Trinidad et al. [TBC06] analyze the detection of dead features based on previous work by Benavides et al. [BSRC10] and propose a way to detect relationships that cause dead features to appear. Trinidad et al. [TBD+08] also introduce further automated analysis methods that help to detect contradictions in feature models as well as so called *full-mandatory* features. The latter are features that actually have not been explicitly defined to be mandatory, but cross-tree constraints implicitly cause them to be present in possible feature model configurations. *Full-mandatory* features have also named to be *false optional* features by Rincón et al. [RGMS14] who propose an ontological rule-based approach for analyzing dead and false optional features in feature models. For that they formalize first-order logic rules to identify dead features and false optional features and these formal rules further enable to identify the relationships that cause the respective issues. Hemakumar [Hem08] focuses on finding contradictions statically using model checking and an incremental consistency algorithm and Wang et al. [WXH+10] introduce a dynamic-priority based approach to fix inconsistencies in feature models. Mendonça [Men09] provides explanations for the space and time (in)tractability of techniques for reasoning on feature models and additionally enhances the algorithmic performance of these techniques.

Configuration support for feature models has also been subject in a multitude of works and different proposals have been made to tackle and optimize this process. We briefly introduce a few selected research papers that depict different aspects of it. Mendonça et al. [MBC09] introduce *S.P.L.O.T.*, a Web-based reasoning and configuration system for SPLs. Their system provides efficient reasoning and interactive configuration services to SPL researchers and practitioners. The interactive configuration support automatically validates every single configuration decision to enforce their consistency. This results in a backtrack-free configuration process directly benefiting users that are never forced to revisit past decisions. Botterweck et al. [BSP09] depict interactive techniques to support the configuration of complex feature models. These techniques include visual interaction with a formal reasoning engine, visual representation of multiple interrelated hierarchies, indicators for configuration progress, and filtering of visible nodes. Barreiros et al. [BM11] propose the use of soft constraints and categorize possible semantics for such constraints. Soft constraints can help to improve configuration support by highlighting more common configuration options.

Even though all of these concepts might be used in a beneficial manner for microservices-based feature models, adopting them is out of this scope of the work and mainly rudimentary concepts are used to perform the different aspects of automated feature model analysis.

4

Applying SPL Techniques on Microservice Based Applications

In this chapter we present how the principles of microservices-based applications fit into the domain of SPLs. Establishing a proper mapping between this particular architectural style and the software development paradigm makes it possible to reuse techniques of SPLs, such as feature models, for the microservices area. Modelling a microservice application and its dependencies as a feature model facilitates the translation of this feature model to a propositional formula. This in turn allows us to use SAT solvers for automatically verifying the correctness of the feature model as well as for validating specific microservice configurations.

4.1 Mapping SPL Artifacts to Microservice Apps

In Section 2.1 we have briefly introduced the main concepts of microservices-based applications and in Section 2.2 we have briefly introduced some SPL tenets where core assets form the basis of SPLs. Comparing the principles of both domains shows that microservices align well with the ideas of SPL. For the core assets in SPLs we have emphasized that all products of a product line share a certain architecture that they have in common [McG04]. The microservices architectural style fits perfectly as an application of such an architecture since it is shared among all microservices, the actual core assets, in a microservices-based application. Microservices are specific kind of components, units of software that are independently replaceable and upgradeable [FL14] and in Section 2.2 we have already listed that *components* are a typical example of core assets. This comparison is further underpinned by the idea of reusability that microservices and core assets have in common. By trying to further derive appropriately mapped artifacts, a microservice-based application proves to be a product in SPLs, since microservices are used to develop microservice applications and core assets are used equally to develop products. This association leads to the final mapping of having a set of microservices-based applications being the counterpart for a generic SPL. This allows us to model the versatile service landscape that exists in environments which apply live experimentation techniques. Only with that we can model microservices-based applications to potentially exist in different combinations of available microservices and multiple versions of microservices. Furthermore, a set of microservices-based applications can have its applications reuse and share services amongst each other which reflects the typical characteristic of SPLs, where core assets are reused in the products of the product line. Table 4.1 shows the mapping of artifacts from SPLs and microservices.

SPLs	Microservices
Software product line	Set of microservices-based applications
Product	Microservices-based application
Core Asset	Microservice

Table 4.1: Mapping Artifacts of SPLs to the Microservices Domain

4.2 Microservice Applications as Feature Models

After thorough analysis of microservices in general, and by adopting the existing concepts of feature models from Section 2.3 as well as the artifact-mapping of Section 4.1, we have derived the building blocks that are required to define a feature model which represents our desired software product line, a set of microservices-based applications. We can distinguish between practices that are used to model the structural relationships of microservices-based applications (i.e. applications, services, and service versions) and practices to model the constraints among the services of such applications (e.g. one service depends on another service).

4.2.1 Features and Structural Relationships

Regarding the features and the structural relation between the features in feature models of microservices applications, we have identified the following building-blocks.

Microservices-based applications. In a feature model that represents microservices-based applications, the *root node* plainly illustrates the domain concept of microservice applications being configurable for the current product line. The feature configurations that are enabled by such models represent each a particularly compiled set of microservices with specific constraints between services. Eventually the feature model and a corresponding service configuration are used together to be validated with the help of a SAT solver in order to get a clear assertion about the validity of the configuration. Essentially, as introduced in Section 2.4.4, valid feature configurations constitute the products of our microservice applications product line.

Microservices. The immediate child features of the feature model's root feature are modelled as an *and* relation and depict the microservices that are available for the current model. Typically, services are defined as *optional* features since it should be usually allowed to select arbitrary subsets of the microservices that are available for a specific product line. Eventually, this offers more flexibility when developing microservices-based products. But in case a service is absolutely required to be comprised in each product of a specific product line, it can be alternatively modelled as a *mandatory* subfeature of the root feature.

Versions. Versions are modelled as *alternative* subfeatures of services since the goal is to eventually obtain feature configurations that explicitly state which distinct versions of which microservices should be used for a particular microservices-based application. From a practical point of view this means that we do not need to enable feature configurations representing runtime environments where potentially multiple different versions of a microservice could exist in parallel. We're just interested in modelling and validating the dependencies of microservices at build-time and for this purpose exactly one distinct version needs to be selectable for each service in a specific service configuration.

Based on the identified characteristics, we derived a formal representation for the parent-child feature relations of microservices-based feature models. To begin with, a feature model for microservices applications consists of a finite set S of n services $s_1, s_2, \ldots, s_n \in S$:

$$Microservices\text{-}BasedApplications \qquad S = \{s_1, s_2, \ldots, s_n\}$$

Services s_x themselves are available in different versions, thus every service s_x leads to a finite set V of n versions $v_{x1}, v_{x2}, \ldots, v_{xn}$ for each service $s_x \in S$:

$$Versions \qquad V(s_x) = \{v_1, v_2, \ldots, v_n\}$$

With the notion of microservice-based applications, microservices, versions, and the relationships between these features, we already have the main building blocks that allow us to model feature models of microservice applications. The structure of these models yield a certain symmetry and some neat properties. Such feature models of microservices-based applications have not only in common that just three levels exist in the feature hierarchy, but also that every path from the root to an arbitrary leaf feature has the exact same depth. Each level represents an abstraction; The topmost level illustrates the domain concept of modelling a set of microservice applications, the second level lists all the available services and the lowest level shows the versions that are available for every single service.

From the overall available types of parent-child relationships in the domain of feature models, only the *or-group* relation is not relevant and thus not used for modelling microservice applications; Neither for microservices nor for versions exist practical implementations of having an "at least one" dependency that is reasonable for parent-child relationships.

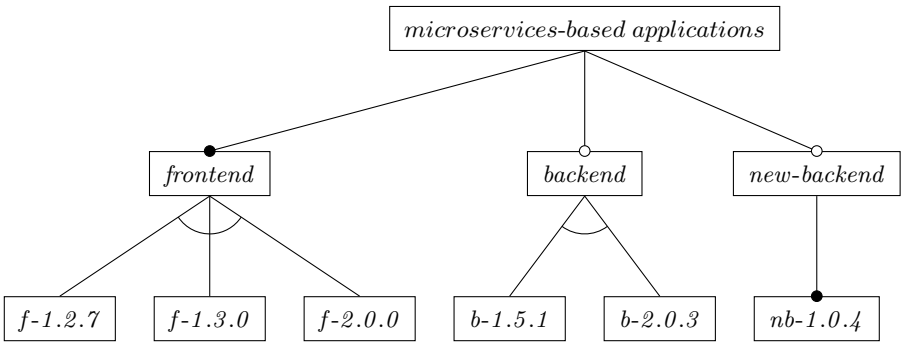


Figure 4.1: Structural Composition in Feature Models of Microservices-Based Applications

Figure 4.1 shows an example of a feature model for microservices-based applications that comprises three different services *frontend*, *backend*, and *new-backend*. The microservice *frontend* has been marked as a mandatory feature to indicate that it needs to exist in all possible service configurations. Furthermore, there exist three different versions for service *frontend*, two for *backend*, and one for *new-backend*, whereas the labels of the version features are prefixed with the corresponding initials of their service names to clearly distinguish the versions of different microservices from each other.

4.2.2 Cross-Tree Constraints

With respect to cross-tree constraints among the features in feature models of microservices applications, we have identified three different constraint types that are required to sufficiently model the possible dependencies between microservices.

Requires. In Section 2.3 we have already introduced the *requires* constraint that evidently indicates that a particular feature requires a specific other feature. Such constraints also typically exist for microservices-based applications and thus it makes sense to reuse them in this current context to represent services that depend on the existence of another service.

Excludes. Similarly to the *requires* constraint, the *excludes* constraint has been also already introduced in section 2.3. Apparently these constraints are used to model features that are not allowed to coexist for the same product. Even though this constraint is less common that the *requires* constraint, they can be useful to highlight microservices that conflict with each other, e.g. microservices obtaining the same network ports.

Alternative. Even though cross-tree constraints give the possibility to model very sophisticated relationships between any features in the model, we only need one further possible constraint that makes sense in the domain of microservice-applications. The *alternative* parent-child relationship, as introduced in Section 2.2, is adopted as a cross-tree constraint to illustrate that a feature specifically depends on one feature from a particular set of different possible options. This is useful to represent microservices that have a specific dependency which can be potentially satisfied by more than one service, but only exactly one of them is required to fulfill the job. *Excludes* constraints can potentially appear in conjunction with *alternative* constraints in order to explicitly define that specific microservice alternatives conflict with each other. Such structures are especially needed in case an *alternative* constraint originates from a non-mandatory microservice. If such a service is not selected for the service configuration, the *excludes* constraints still ensures that the conflicting alternatives can not be selected simultaneously for the current configuration.

Since dependencies between services can change due to version upgrades, we decided to model constraints between the features that represent versions instead of modelling them between microservices. Of course one could argue that defining constraints between the microservices itself should be enabled too, since constraints between services might not alter that frequently. But eventually it would only add unnecessary complexity to the feature model and actually, it can be individually resolved in the corresponding implementations of such feature models in case such a property is desired at all.

Based on the identified characteristics, we derived a formal representation for the cross-tree constraints of microservices-based feature models. Constraints c_x always originate from specific versions, thus every version v_x leads to a finite set C of n constraints $c_{x1}, c_{x2}, \ldots, c_{xn}$ for each version $v_x \in V$:

$$Constraints \qquad C(v_x) = \{c_1, c_2, \ldots, c_n\}$$

In Figure 4.2 we have extended the feature model of Figure 4.1 with some constraints between the individual versions of microservices. This feature model, or slight alterations of it, will be used throughout the remainder of the thesis as a running example. Microservice version $f\text{-}1.2.7$

requires microservice version $b\text{-}1.5.1$ and microservice version $f\text{-}2.0.0$ requires microservice version $nb\text{-}1.0.4$. For each version of microservice $backend$, i.e. $b\text{-}1.5.1$ and $b\text{-}2.0.3$, excludes constraints to $nb\text{-}1.0.4$ are modelled to indicate that these specific service versions would conflict with each other. Finally, an *alternative* constraint is defined at the bottom of the feature hierarchy, indicating that $f\text{-}1.3.0$ depends either on $b\text{-}2.0.3$ or on $nb\text{-}1.0.4$. The Boolean formula shows the small difference of *alternative* cross-tree constraints compared to the *alternative* parent-child relations of feature models. The latter requires that for every subfeature an implication is defined from the subfeature to its parent, but the former has no such parent-child relation since the constraints are drawn between the versions of services. The features are on the same level and therefore no implications need to be defined from the version options to the version from which the *alternative* cross-tree constraint actually originates.

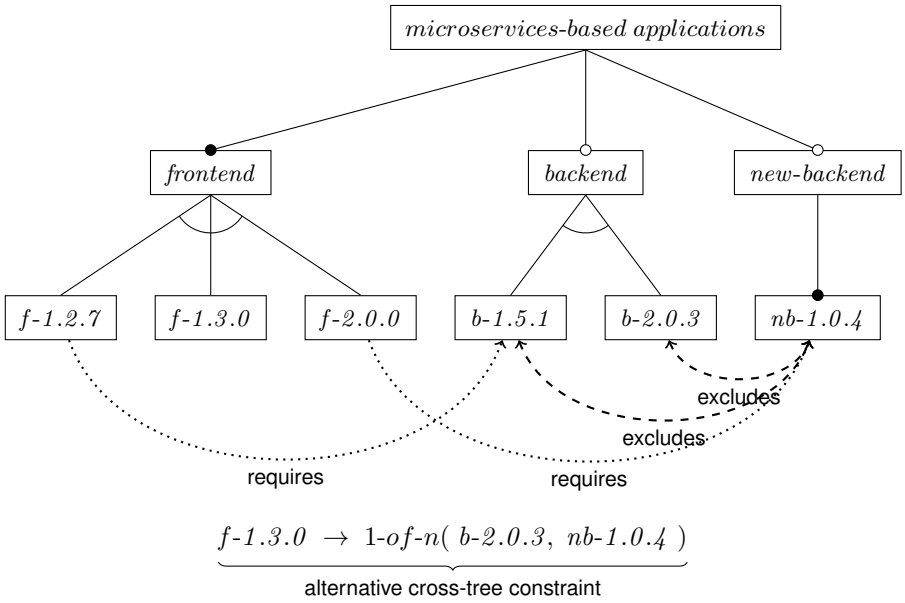


Figure 4.2: Constraints in Feature Models of Microservices-Based Applications

4.3 Automated Analysis of Microservice Apps

The proper mapping of feature models to the microservices domain makes it possible to adopt the practices and techniques that have been introduced in Section 2.4 and to apply them appropriately for microservices-based feature models.

4.3.1 Microservice Apps as Propositional Formulas

We adopt the Boolean formulas discussed in Section 2.4 with respect to the concepts of feature models for microservices-based applications from Section 4.2. Table 4.2 shows the Boolean formulas that need to be defined for the three different types of features and the three different cross-tree constraints in such feature models. The *root* feature can be adopted without any changes and thus the feature is represented by a simple formula a. An *optional* microservice s is modelled as a being

the parent of s, thus the formula is $s \rightarrow a$. A *mandatory* microservice needs the implication also in the other direction, thus we additionally define $a \rightarrow s$. As already highlighted in Section 2.4.2, implications in both directions can be replaced by the logical equivalence operator, so we can define a *mandatory* microservice also simply just with the single formula $a \leftrightarrow s$. For representing a version we use the corresponding Boolean formulas of the *alternative* relationship. Thus, for every version $v_1, ..., v_n$ available for a microservice s, we need to state an implication from the version to the microservice, i.e. $v_1 \rightarrow s \wedge, ..., \wedge v_n \rightarrow s$. Furthermore, the parent to child relationship from service s to the [1..1] grouped features $v_1, ..., v_n$ is defined with $s \rightarrow 1\text{-}of\text{-}n(v_1, ..., v_n)$ in order to complete the definition for the *alternative* relationship from microservices to versions. As described in Section 4.2.1, the *or* relation between features and subfeatures is not feasible in the microservices domain and therefore we also can not reuse the corresponding translation rule in a meaningful manner. The Boolean formulas for representing cross-tree constraints of feature models for microservice applications are given in the same way as by general feature models; The constraints are already formulated as Boolean formulas at the bottom of the models. The formulas for defining the *requires* and *excludes* constraints have been already presented before in Section 2.4.2 and also the *alternative* cross-tree constraint has been already defined in Section 4.2.2, thus the constraints are shown without further ado by the three last rows of Table 4.2.

Relation	**Feature model context**	**Boolean formula**
Microservice apps	Domain concept of modelling microservice applications a is the root feature	a
Optional service	s is subfeature of application a	$s \rightarrow a$
Mandatory service	s is subfeature of a a is parent feature of s shorthand: s is mandatory service of a	$s \rightarrow a$ $a \rightarrow s$ $s \leftrightarrow a$
Version	The version v_1 is a subfeature of s The version v_n is a subfeature of s s is parent of [1..1] grouped features $v_1, ..., v_n$	$v_1 \rightarrow s$ $v_n \rightarrow s$ $s \rightarrow 1\text{-}of\text{-}n(v_1, ..., v_n)$
Requires	v_1 of s_1 requires v_1 of s_2	$s_{1v_1} \rightarrow s_{2v_1}$
Excludes	v_1 of s_1 prohibits v_1 of s_2 v_1 of s_2 prohibits v_1 of s_1 shorthand: v_1 of s_1 and v_1 of s_2 exclude each other	$s_{1v_1} \rightarrow \neg s_{2v_1}$ $s_{2v_1} \rightarrow \neg s_{1v_1}$ $\neg(s_{1v_1} \wedge s_{2v_1})$
Alternative	v_1 of s_1 is parent of [1..1] grouped features $s_{2v_1}, ..., s_{mv_n}$	$s_{1v_1} \rightarrow 1\text{-}of\text{-}n(s_{2v_1}, ..., s_{mv_n})$

Table 4.2: Translation Rules for Microservices-Based Feature Models

In Section 2.4 we have already defined that by conjoining the Boolean formulas for all elements of a feature model, we get the corresponding propositional formula. Figure 4.3 shows the appropriate propositional formula of the feature model shown in Figure 4.2. The first line (1) defines modelling of microservices-based applications (MBA) as the root element. The second line (2) describes the services comprised in the MBA whereas an equivalence operator is used to appropriately define *frontend* as a mandatory service. Line (3) represents the *alternative* relationship between service s_1 and its three available versions $f\text{-}1.2.7$, $f\text{-}1.3.0$ and $f\text{-}2.0.0$. Line (4) shows

the same relationship between service *backend* and its versions *b-1.5.1* and *b-2.0.3* and similarly line (5) shows the same for service *new-backend* for which only one version *nb-1.0.4* exists. The remainder of the propositional formula, lines (6) to (10), represents the different cross-tree constraints that have been captured in the feature model. Line (6) defines a *requires* constraint from microservice version *f-1.2.7* to *b-1.5.1* and line (7) does it likewise from *f-2.0.0* to *nb-1.0.4*. Line (8) shows the *excludes* constraint between the service versions *b-1.5.1* and *nb-1.0.4* and line (9) does the same for *b-2.0.3* and *nb-1.0.4*. Finally, line (10) defines the *alternative* cross-tree dependency from service *f-1.3.0* to the two possible options *b-2.0.3* or *nb-1.0.4*.

$$
\begin{aligned}
&(mba) \wedge && (1)\\
&(frontend \leftrightarrow mba) \wedge (backend \rightarrow mba) \wedge (new\text{-}backend \rightarrow mba) \wedge && (2)\\
&(f\text{-}1.2.7 \rightarrow frontend) \wedge (f\text{-}1.3.0 \rightarrow frontend) \wedge (f\text{-}2.0.0 \rightarrow frontend) \wedge\\
&\quad (frontend \rightarrow (f\text{-}1.2.7 \wedge \neg f\text{-}1.3.0 \wedge \neg f\text{-}2.0.0)\\
&\qquad \vee (\neg f\text{-}1.2.7 \wedge f\text{-}1.3.0 \wedge \neg f\text{-}2.0.0)\\
&\qquad \vee (\neg f\text{-}1.2.7 \wedge \neg f\text{-}1.3.0 \wedge f\text{-}2.0.0)) \wedge && (3)\\
&(b\text{-}1.5.1 \rightarrow backend) \wedge (b\text{-}2.0.3 \rightarrow backend) \wedge\\
&\quad (backend \rightarrow (b\text{-}1.5.1 \wedge \neg b\text{-}2.0.3) \vee (\neg b\text{-}1.5.1 \wedge b\text{-}2.0.3)) \wedge && (4)\\
&(nb\text{-}1.0.4 \rightarrow new\text{-}backend) \wedge (new\text{-}backend \rightarrow nb\text{-}1.0.4) \wedge && (5)\\
&(f\text{-}1.2.7 \rightarrow b\text{-}1.5.1) \wedge && (6)\\
&(f\text{-}2.0.0 \rightarrow nb\text{-}1.0.4) \wedge && (7)\\
&\neg(b\text{-}1.5.1 \wedge nb\text{-}1.0.4) \wedge && (8)\\
&\neg(b\text{-}2.0.3 \wedge nb\text{-}1.0.4) \wedge && (9)\\
&(f\text{-}1.3.0 \rightarrow (b\text{-}2.0.3 \wedge \neg nb\text{-}1.0.4) \vee (\neg b\text{-}2.0.3 \wedge nb\text{-}1.0.4)) && (10)
\end{aligned}
$$

Figure 4.3: Feature Model as Propositional Formula

4.3.2 Correctness Checking and Configuration Validation

Adopting feature models for modelling product lines of microservices-based applications and appropriately reusing the demanded translations rules makes it possible to apply automated analysis of feature models in the microservices domain. Correctness checking as well as validating feature configurations of microservices-based feature models do not differ from the general feature model approaches introduced in Section 2.4, thus we illustrate the concepts by giving examples that are based on Figure 4.2.

Correctness Checking. Figure 4.4 shows that slight adjustments to the feature model of Figure 4.2 can quickly turn it into an inconsistent model. This time, all microservices are defined to be *mandatory*. This results in conflicts between the versions of the services *backend* and *new-backend*. Both versions *b-1.5.1* and *b-2.0.3* of service *backend* have each an *excludes* constraint defined that conflicts with *nb-1.0.4* which also needs to be comprised in each service configuration of every microservice-based application of the product line. The feature model in Figure 4.4 is clearly in an inconsistent state, whereas the corresponding conflicting items are highlighted appropriately.

Even less adjustments to the original microservices-based feature model of Figure 4.2 are necessary to yield *dead* features. In Figure 4.5 only service *new-backend* has been additionally defined to be mandatory. This results to the problem that neither *backend* nor one of its versions *b-1.5.1*

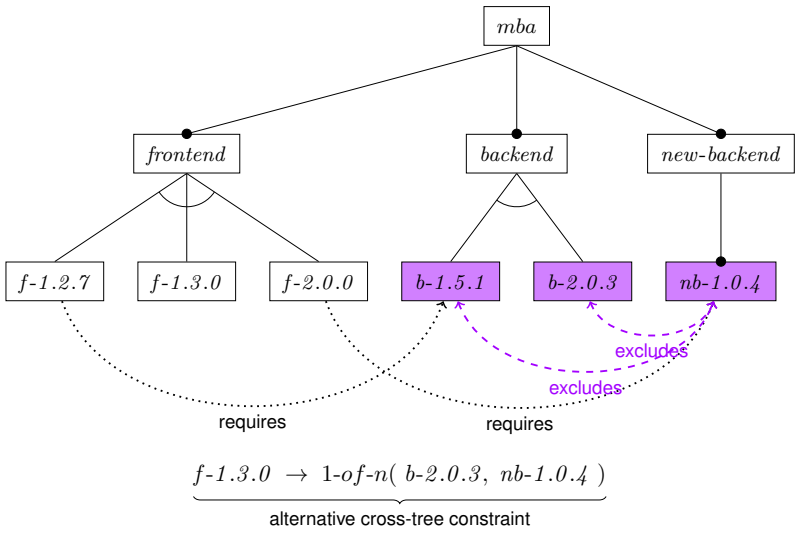


Figure 4.4: Verifying Inconsistencies of Microservices-Based Feature Models

or $b\text{-}2.0.3$ can be selected for valid feature configurations. Since $new\text{-}backend$ is mandatory for all products, and with that also its only existing version $nb\text{-}1.0.4$ the two *excludes* constraints existing in the feature model completely prohibit the selection of $backend$ $b\text{-}1.5.1$ or $b\text{-}2.0.3$ making them *dead* features. This has been correspondingly highlighted in Figure 4.5.

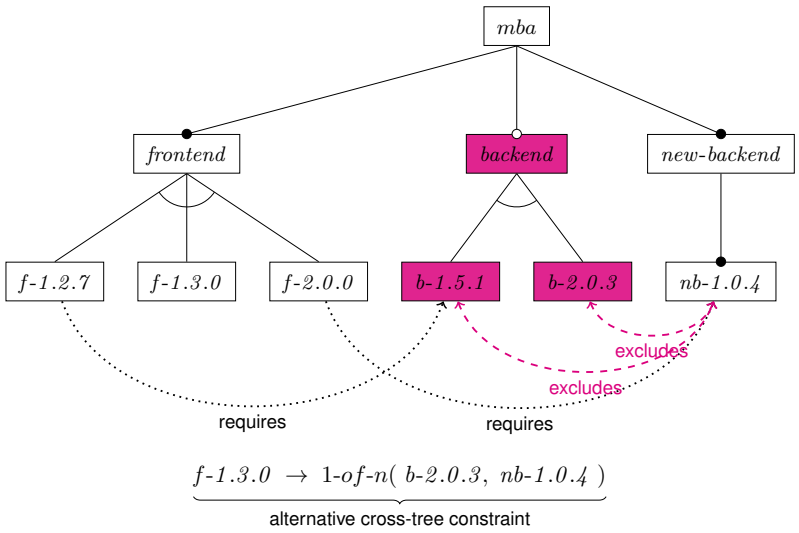
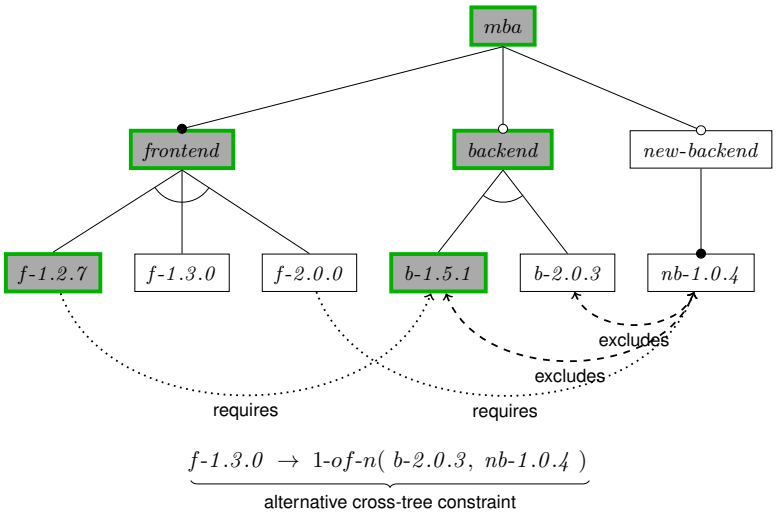


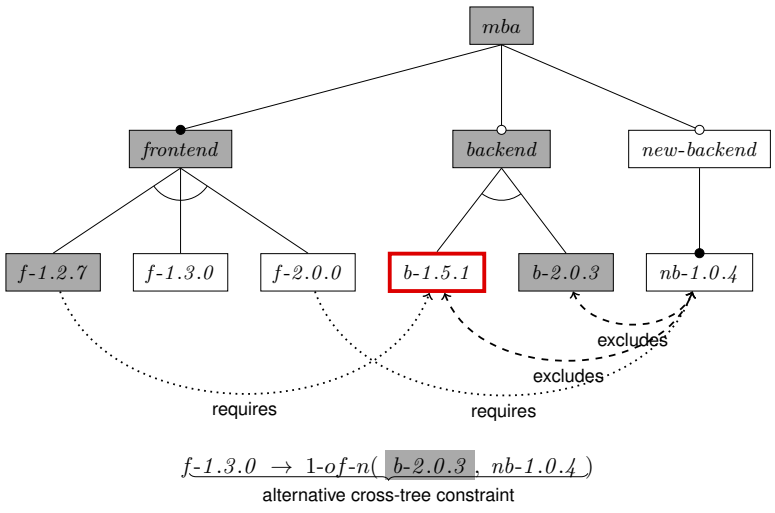
Figure 4.5: Detecting Dead Features in Microservices-Based Feature Models

Validating Service Configurations. In Figure 4.6 we show two examples of feature configurations. Figure 4.6(a) shows that the features $\{mba, frontend, f\text{-}1.2.7, backend, b\text{-}1.5.1\}$ have been selected and thus are highlighted accordingly in the feature model. A SAT solver that validates

the given feature configuration against the propositional formula of Figure 4.3 would confirm the configuration to be legal. This is because no constraints are violated by the selected features and thus these features are appropriately visualized with green borders. In contrast to Figure 4.6(a), Figure 4.6(b) shows an illegal feature configuration. This time, feature $b\text{-}2.0.3$ has been selected instead of $b\text{-}1.5.1$ and this violates the defined *requires* constraint of $f\text{-}1.2.7$ depending on $b\text{-}1.5.1$. Feature $b\text{-}1.5.1$ has been highlighted appropriately to represent the constraint violation.



(a)



(b)

Figure 4.6: Service Configurations of Microservices-Based Applications

5

Implementation

In this section, the HEIMDALL web based application is presented. The system is a prototype implementation of our microservices-specialized feature model and is based on Node.js[1], Vue.js[2], and vis.js[3]. Our prototype targets microservices-based applications, specifically the modelling and validation of dependencies between microservices.

5.1 System Overview

As visualized in Figure 5.1, the HEIMDALL application follows the microservice architectural style and is composed of four distinct services; The frontend, the software product line (SPL), the satisfiability (SAT) solver, and the database (DB) service.

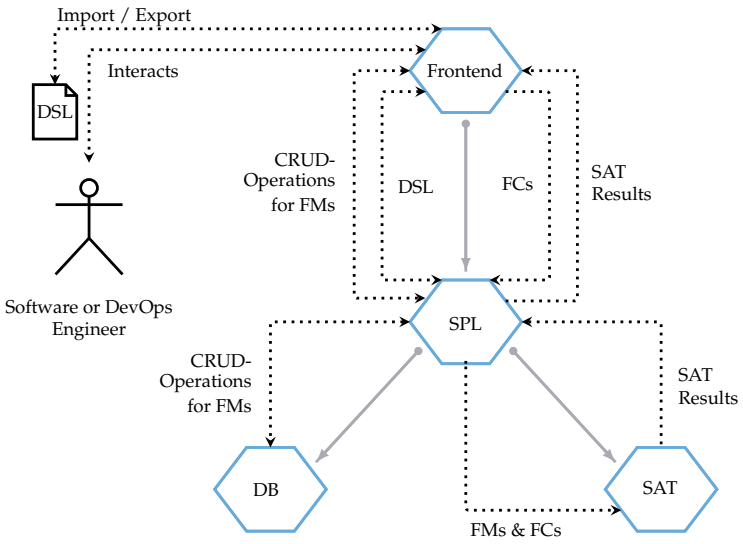


Figure 5.1: High-level Architectural Overview of the HEIMDALL Application

[1]https://nodejs.org
[2]https://vuejs.org
[3]http://visjs.org

In the following, we will briefly discuss the responsibilities, interactions, and dependencies of the microservices.

Frontend & SPL. The frontend service is the single interaction point that software and DevOps engineers can interact with. It offers graphical support to create and maintain feature models, define feature configurations on the basis of feature models, and to validate the feature models (FMs) and the feature configurations (FCs). Furthermore it is also possible to have the feature models defined in a YAML[4]-based domain-specific language (DSL) [MHS05] which enables importing or exporting feature models in that particular format. The frontend service depends on the SPL service which processes the incoming requests from the fronted.

SPL & DB. The SPL service exposes a RESTful API to the Frontend. It validates incoming CRUD-requests for the different elements existing in microservices-based feature models to allow retrieval and manipulation of applications, services, versions, and constraints from the DB service. In case feature models are uploaded in the DSL, they get parsed and appropriately decomposed and populate to the DB. In case of an export request, the process is exactly the same vice-versa.

SPL & SAT. The SPL service does not only depend on the DB service but also on the SAT service. When a request for correctness checking of a feature model is received, the SPL service gathers the corresponding elements of the product line from the DB service, compiles it to a JSON representation of the feature model and sends it to the SAT service. The SAT service translates the JSON representation of the feature model to a propositional formula and validates it with the help of an existing SAT solver. Depending on the result delivered by the SAT solver, the SAT service can return three different response types with respect to the correctness of feature models:

- **Consistent Feature Model:**
  One possible result is that a the feature model is consistent and a corresponding confirmation is returned.
- **Inconsistent Feature Model:**
  In case the result is not consistent, hints on how to resolve the inconsistent feature model are returned.
- **Dead Features:**
  If the result is consistent, check if dead features exist and return a list of these features.

Similarly, when the validation of a feature configuration is requested, familiar actions are performed. the SPL service again gathers the elements of the product line from the DB and compiles it in the same way before sending it together with the feature configuration to the SAT service. The SAT service performs a quite similar translation of the feature model to a propositional formula, but then it validates the given feature configuration against the formula with the help of the SAT solver. The SAT service returns three different types of responses, depending on the result computed by the SAT solver:

- **Legal Configuration:**
  The first possibility is that the SAT service response with a confirmation that the configuration is legal.
- **Incomplete Configuration:**
  In case the configuration is incomplete, a further response type can suggest feature model elements that may lead to a legal configuration.

[4]http://yaml.org

- **Illegal Configuration:**
  The third possibility is, in case the configuration is illegal, that the SAT service first performs a neat backtracking algorithm to resolve which selected features are responsible for the illegal configuration. The SAT solver the returns information about the possible culprits.

5.2 Technology Stack

The HEIMDALL application has been developed mainly in JavaScript utilizing Node.js as the server-side JavaScript runtime and on the frontend side the Vue.js framework as well as the vis.js visualization library have been used.

Backend. Node.js was not only chosen due to its lightweight and efficient architecture, but also due to its low entry level that makes it easy to learn and thus to a perfect candidate for getting prototypes quickly up and running. This gets further emphasized by the fact that Node.js is bundled with NPM, the world's largest software registry. Due to NPM, it is not only relatively easy to find useful third party libraries and tools, but also directly integrating them is a breeze. This was especially useful to find and incorporate an appropriate SAT solver for the HEIMDALL application. Eventually, Logic Solver[5] has been used for that purpose, a Boolean satisfiability solver written in JavaScript that contains MiniSat[6], an industrial-strength SAT solver. In order to have maximal flexibility with respect to data structures, MongoDB[7] is used to store data about the features and constraints of feature models. It is a document database and stores data in flexible, JSON[8]-like documents. The communication between the services of the application is handled through RESTful HTTP APIs on the basis of ExpressJS[9].

Frontend. For the frontend side, Vue.js was chosen as the main framework in order to keep prototyping as smooth as possible. Vue.js is more flexible and less opinionated than AngularJS[10] and with respect to React[11], it offers similar strengths, such as utilization of a virtual DOM and providing reactive and compose-able view components, but has in contrast a more flat learning curve and shorter path to productivity. In order to dynamically visualize the feature diagrams in the browser, vis.js has been used. It is designed to be easy to use and even though it is not as powerful or versatile as other visualization libraries such as D3[12], it is able to handle large amounts of dynamic data and provides appropriate manipulation and interaction possibilities with the data.

Deployment. In order to ease deployment and reproducibility of experiments, we made use of Docker[13], the world's leading software container platform [Inc17]. Although HEIMDALL does not yet consist of many moving parts in its microservices architecture, we have decided to use Docker Compose[14] in order to facilitate defining and running our multi-container application whereas each service is placed in its own container.

[5] https://www.npmjs.com/package/logic-solver
[6] http://minisat.se
[7] https://www.mongodb.com
[8] http://www.json.org
[9] http://expressjs.com
[10] https://angularjs.org
[11] https://facebook.github.io/react
[12] https://d3js.org
[13] https://www.docker.com
[14] https://docs.docker.com/compose

5.3 Data Modelling of Feature Models

HEIMDALL allows creating and manipulating feature models of microservices-based applications. Therefore appropriate elements of this domain need to be modelled in the database. In Section 4.2.1 we have identified the three different types of features available in the feature models, namely *microservices-based applications*, *microservices*, and *versions*. We have formally defined that a *microservices-based application* consists of a finite set of *microservices* and each *microservice* is available in a finite set of *versions*. Additionally, we have shown in Section 4.2.2 the existence of the three different cross-tree constraint types, such as the *requires*, *excludes*, and *alternative* constraints. We have determined that constraints are modelled between the versions of services and since we have formally defined that constraints always originate from a specific version, each version can own a finite set of constraints. Figure 5.2 shows a UML diagram of a schematic database model that represents the distinct elements, attributes, and relations that are required to implement feature models of microservices-based applications. Since MongoDB is used as the actual database technology, data is stored in so called *collections* as opposed to data that is stored in tables in relational database systems.

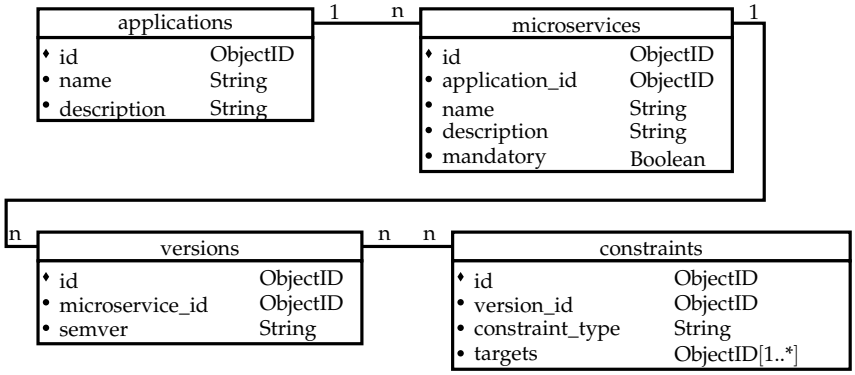


Figure 5.2: Data Model of Feature Model Elements

Applications. HEIMDALL allows managing multiple microservices-focused SPLs, thus the creation and manipulation of multiple feature models needs to be covered. We have decided to represent feature models with their root feature and therefore we store the root features of the feature models, i.e. *microservices-based applications*, in a corresponding collection abbreviated as `applications`. Besides the uniquely identifying attribute `id`, we provide two further attributes; `name` to label the feature with a meaningful name, and `description`, which allows to add some descriptive information about the product line that is modelled with the feature model.

Microservices. Services are appropriately stored in the `microservices` collection, each having a unique `id` as well as a reference attribute `application_id` that links to the root feature that they belong to. This allows us to model the child-parent relationship between the `applications` and the `microservices` with a $[1..n]$ cardinality, which is necessary since we have formally defined that feature models of *microservices-based applications* consist of a finite set of *microservices*. Equally to the root feature, services have the attributes `name` and `description`,

whereas the latter gives the possibility for briefly describing the corresponding service. Finally, the `mandatory` attribute is used to indicate whether a microservice is obligatory or not.

Versions. Versions are stored in the appropriate `versions` collection, with their respective unique *id* as well as the `microservice_id` reference to the microservice that they correspond to. Similarly to the relationship between applications and services, the `microservice_id` reference is used to model the the $[1..n]$ cardinality between a service and its available versions. This covers our formal definition that each *microservice* is available in a finite set of *versions*. A version's `semver` attribute is used to represent it with a meaningful tag.

Constraints. All constraints are stored in the same collection named `constraints`. This elements also have the `id` attribute as a unique identifier and with the help of the `version_id` attribute we can model the formally defined finite set of constraints that can originate from a specific version. As the name indicates, the `constraint_type` attribute is used to represent whether the constraint is of type *requires*, *excludes*, or *alternative*. The `targets` attribute is an array of ids that represent foreign keys pointing back to a specific set of versions. Therefore we have a $[n..n]$ cardinality between the *versions* and the *constraints* collection. The important point here is that `requires` and `excludes` constraints only have one single version identifier in the `targets` attribute, but alternative constraints can have more than just one version identifier in the `targets` array. The latter enables modelling the dependency between a specific microservice version (i.e., the `version_id` attribute) and a set of multiple alternative versions of potentially different services (i.e., the `targets` array) that it can choose from.

5.4 Domain-Specific Language

As previously introduced, we have defined a DSL in order to enable importing and exporting of feature models in the YAML standard. On the one hand, this makes it possible to specify feature models in this particular DSL and import them into the HEIMDALL application, on the other hand, in conjunction with the export functionality, the DSL can be used as a specific data exchange format between HEIMDALL and other systems. The DSL was built as an internal DSL on top of YAML as a host language. YAML is a data serialization language designed to be readable by humans. In the following, we will present the DSL with the help of a sample feature model specification and explain the design decisions behind it. Furthermore, we will also briefly describe what happens when a feature model specification in the DSL is imported to HEIMDALL and how the data is further used in the system.

Introducing the DSL. Listing 5.1 shows a part of the specification that is required to define the feature model from our running example in Figure 4.2. The DSL is a straightforward mapping of the elements and attributes defined in the data model of Section 5.3. YAML's root object represents the root feature of microservices-based feature models. It comprises almost the same attributes as the documents from the applications collection of the data model, as shown with the `name` and `description` keys on line 1 and 2. The `id` property does not need to be mapped to a corresponding key in the DSL (the same is true for the `id` properties of services, versions, and constraints). The `microservices` key on line 3 lists the services that are available for the current feature model. The services have the same properties as in the data model, e.g. lines 4 to 6 represent the mandatory *Frontend* service including a description. Each service has a `versions` key that lists the different obtainable versions. Versions comprise the `semver` key to represent a version's tag. For example *1.2.7*, *1.3.0*, and *2.0.0* on lines 8, 14, and 22 depict the three different versions that exist for the *Frontend* microservice. The `constraints` key of a version lists the

constraints that originate from the corresponding version. For example the `constraints` list on line 9 contains just one single constraint. Equally to the data model, each constraint contains a `constraint_type` property as well as the `targets` key that contains the list of a constraint's targets, e.g. as defined on line 10 and 11 respectively. Equally to the data model, *requires* and *excludes* constraints only have one target whereas *alternative* constraint can have several targets. Since we do not have any uniquely identifying keys for versions of services, a target is comprised of two keys, the `name` that points to a certain other service and the `semver` that specifies which exact version of the service is targeted. For example, lines 12 and 13 define that the corresponding *requires* constraints targets version *1.5.1* of the *Backend* service. The detailed specifications for the *Backend* and the *New-Backend* services are omitted since they do not vary considerably from the introduced concepts so far. The full specification is available as attachment in Appendix A.2.

```
1  name: Microservices-Based Applications
2  description: Feature model defining a sample set of microservices based applications
3  microservices:
4  - name: Frontend
5    description: Sample frontend service
6    mandatory: true
7    versions:
8    - semver: 1.2.7
9      constraints:
10     - constraint_type: requires
11       targets:
12       - microservice: Backend
13         version: 1.5.1
14   - semver: 1.3.0
15     constraints:
16     - constraint_type: alternative
17       targets:
18       - microservice: Backend
19         version: 2.0.3
20       - microservice: New-Backend
21         version: 1.0.4
22   - semver: 2.0.0
23     constraints:
24     - constraint_type: requires
25       targets:
26       - microservice: New-Backend
27         version: 1.0.4
28 - name: Backend
29   # Details omitted
30 - name: New-Backend
31   # Details omitted
```

Listing 5.1: Sample Feature Model Defined in the YAML Based DSL

Importing a Feature Model Specified in the DSL. The import of a feature model which is specified in the DSL is quite straightforward. The specification gets parsed with the help of the js-yaml[15] JavaScript library which helps to return a JavaScript object representation of the feature model. The information about the elements of the feature model is then appropriately used to create the specific data representations of the feature model in the database. As previously indicated, no explicit `id` keys need to be defined by a feature model specification in general since

[15]https://www.npmjs.com/package/js-yaml

the ids will be generated when the feature model elements are created in the database. In case the feature model is required by the frontend service or the SAT service, the SPL service fetches all the corresponding elements of the desired feature model from the database, compiles it to a JavaScript object and sends it to the appropriate service after parsing it to JSON. A small part of the JSON formatted feature model from our running example is illustrated in Listing 5.2, which shows the close similarity of the DSL to the JSON exchange format of feature models between services. One main reason is that YAML 1.2 is a superset of JSON, which is actually just another data serialization format [Eva17].

```
{
  "id": "59805bc0b9572c4514713bec",
  "name": "Microservices-Based Applications",
  "description": "Feature model describing a sample set of microservices based apps",
  "microservices": [
    {
      "id": "59805bc0b9572c4514713bed",
      "application_id": "59805bc0b9572c4514713bec",
      "name": "Frontend",
      "description": "Sample frontend service",
      "mandatory": true,
      "versions": [Array] // Details omitted
    },
    {
      "id": "59805bc0b9572c4514713bee",
      "name": "Backend",
      // Details omitted
    },
    {
      "id": "59805bc0b9572c4514713bef",
      "name": "New-Backend",
      // Details omitted
    }
  ]
}
```

Listing 5.2: Sample Feature Model Defined Formatted in JSON

5.5 Visualization of Feature Models

The implementation of feature diagrams is achieved by using the vis.js visualization library. The library consists of multiple components whereas its network component provides the necessary set of functionalities to draw the hierarchical graph of feature models. In HEIMDALL, we have combined Vue.js and vis.js together in order to implement an editor for manually creating and manipulating feature models of microservice applications. Figure 5.3 shows a screenshot to illustrate how the graphical representation of feature models is realized in HEIMDALL. The example shows the implementation from the running example of Figure 4.2. Even though the implementation follows the characteristics of the original definition of feature diagrams, there exist some subtle differences that need to be highlighted.

Mandatory and Optional Services. One difference is that, due to some technical restrictions of the vis.js library, it is not possible to draw the distinction between mandatory and optional microservices with filled or empty circles. Instead of using filled circles for mandatory services,

the labels of such services are simply prefixed with an *asterisk* whereas the labels of optional services are not.

Alternative Relationship Between Services and Versions. Another change that is caused due to the limitations of the vis.js library, is that no arc is drawn between the linking lines of a service and its versions. Since we have only the *alternative* relationship between every service and its versions, we do not see the requirement to indicate this relation visually in any way.

Alternative Cross-Tree Constraints. Another visual change for feature models is caused by the effort to improve the usability of the feature model editor and the readability of microservices-based feature models. The *alternative* cross-tree-constraint is the only type of cross-tree constraints that has no proper graphical notation. In order to spare users of the HEIMDALL application from the formally written-out Boolean formulas of these constraints, we decided to graphically visualize them as well. An *alternative* constraint comprises more than one target version, thus multiple arrows need to be drawn to appropriately represent the $[1..n]$ relationship between the originating service version and the targeted service versions. The arrows are similar to the ones of the *requires* constraints, but in order to visually group the arrows of an *alternative* constraint, each *alternative* constraint gets randomly color coded with a specific color. This should allow users to better distinguish multiple *alternative* constraints, even if they originate from the same microservice version.

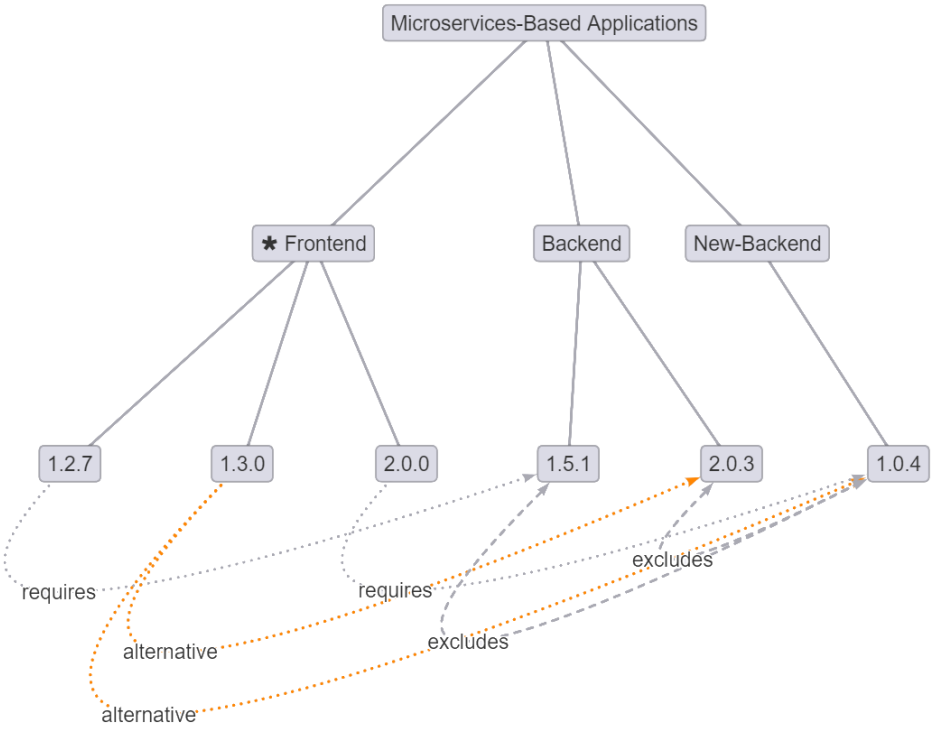


Figure 5.3: Visualization of Feature Models in HEIMDALL

5.6 Implementing Propositional Formulas

The prerequisites for performing automated analysis on feature models are to have an appropriate SAT solver at hand and to translate the feature model to propositional formulas. As described in Section 5.2, we have decided to use *Logic Solver* as SAT solver for the HEIMDALL prototype. Logic Solver is utilized by defining problems as logical constraints on Boolean variables. It then either provides all possible solutions, or determines that there is definitely no possible assignment of the variables that would satisfy the logical constraints. In Section 4.3.1 we have already defined the Boolean formulas that are required to specify a feature model, thus we now just need to accordingly express and compile these formulas with Logic Solver in order to have a feature model ready for automated analysis.

5.6.1 Logic Solver Expressions

Logic Solver uses an intuitive pre-order notation to adopt the expressions of propositional logic in a straightforward manner. The operators of propositional logic are represented by solver-functions with appropriately defined names. Table 5.1 shows the Boolean formulas of Table 4.2 that are required for the implementation and depicts the appropriately mapped Logic Solver expressions. Implications $\rightarrow$ can be defined with the help of the `implies` function and the $[1..1]$ grouped features are realized with the `exactlyOne` function. Furthermore, the `not` function can be used to implement the negation $\neg$ and a conjunction is mirrored with the `and` function.

Relation	Boolean formula	Logic Solver expression
Micros. Apps	a	`a`
Opt. service	$s \rightarrow a$	`Logic.implies(s, a)`
Mand. service	$s \rightarrow a$	`Logic.implies(s, a)`
	$a \rightarrow s$	`Logic.implies(a, s)`
Version	$v_1 \rightarrow s$	`Logic.implies(`v_1`, s)`
	$v_n \rightarrow s$	`Logic.implies(`v_n`, s)`
	$s \rightarrow 1\text{-}of\text{-}n(v_1, ..., v_n)$	`Logic.implies(s, Logic.exactlyOne(`v_1`,...,`v_n`))`
Requires	$s_{1_{v_1}} \rightarrow s_{2_{v_1}}$	`Logic.implies(`$s_{1_{v_1}}$`, `$s_{2_{v_1}}$`)`
Excludes	$\neg(s_{1_{v_1}} \wedge s_{2_{v_1}})$	`Logic.not(Logic.and(`$s_{1_{v_1}}$`, `$s_{2_{v_1}}$`))`
Alternative	$s_{1_{v_1}} \rightarrow 1\text{-}of\text{-}n(s_{2_{v_1}}, ..., s_{m_{v_n}})$	`Logic.implies(`$s_{1_{v_1}}$`, Logic.exactlyOne(`$s_{2_{v_1}}, ..., s_{m_{v_n}}$`))`

Table 5.1: Implementing Translation Rules for Microservices-Based Feature Models

5.6.2 Conjoining Boolean Formulas

Listing 5.3 shows the corresponding code for providing the SAT solver with the formulas defined in Table 5.1. Conjoining the Boolean formulas to the complete propositional formula is realized in Logic Solver by using the `requires()` function. A Boolean formula that is passed as argument to that method is conjoined to all of the previously added Boolean expressions of the solver. Line 1 indicates that the `fm` variable holds the complete data of a feature model. Line 2 shows the instantiation of the `satSolver` object and line 3 adds the `id` of the feature model to the solver.

The latter statement represents the usage of the Boolean formula for the root feature, i.e. the first row of Table 5.1. From line 5 to 30 we iterate through all the services of the Feature Model. Line 7 as the first line in the loop adds for each single service the Boolean formula that define the services to be subfeatures of the root feature. Lines 8 to 10 check whether the service is mandatory and correspondingly extend the implication of subfeatures with a further implication in the other direction in order to define the service to be mandatory. Line 12 defines the `versiondIds` array that is used to collect the `id` attributes of all version of a single service. From line 13 to 28 we iterate through all versions of the current service whereas line 14 adds the correct formula to define the versions to be subfeatures of the service feature. Additionally, on line 15, the `id` of the version is added to the `versiondIds` array. Further down on line 29, after looping through all the versions of a service, this array is then used to add the formula that defines that only exactly one version needs to be selected among all existing versions for a microservice. From Line 17 to 27 we iterate through all constraints that originate from the current service version. Lines 18 to 26 add the corresponding logical expressions for the current constraint depending on its `constraint_type` property. Finally, after looping through all services, versions, and constraints and conjoining the appropriate Boolean formulas, the `satSolver` object is now ready and available for automated feature analysis of the feature model.

```
1  (fm) => {
2    let satSolver = new Logic.Solver();
3    satSolver.require(fm.id);
4
5    for (let microservice of fm.microservices) {
6
7      satSolver.require(Logic.implies(microservice.id, fm.id));
8      if (microservice.mandatory === true) {
9        satSolver.require(Logic.implies(fm.id, microservice.id));
10     }
11
12     let versionIds = [];
13     for (let version of microservice.versions) {
14       satSolver.require(Logic.implies(version.id, microservice.id));
15       versionIds.push(version.id);
16
17       for (let constraint of version.constraints) {
18         if (constraint.constraint_type === 'requires') {
19           satSolver.require(Logic.implies(version.id, targets[0]));
20         }
21         else if (constraint.constraint_type === 'excludes') {
22           satSolver.require(Logic.not(Logic.and(version.id, targets[0])));
23         }
24         else if (constraint.constraint_type === 'alternative') {
25           satSolver.require(Logic.implies(version.id, Logic.exactlyOne(constraint.targets)));
26         }
27       }
28     }
29     satSolver.require(Logic.implies(microservice.id, Logic.exactlyOne(versionIds)));
30   }
31
32   return satSolver;
33 }
```

Listing 5.3: Translation of Feature Models to Propositional Logic

5.7 Correctness Checking of Feature Models

As introduced in Section 2.4.1, the correctness checking category of feature model analysis comprises consistency checking and finding dead features. In the following we will illustrate the main concepts that stand behind the implementations of these analysis methods in HEIMDALL.

5.7.1 Consistency Checking

Logic Solver provides a `solve` function for solver objects. This enables the computation of possible solutions for a given problem. Calling the `solve` function returns one single possible solution, or nothing. The latter is the case if there are no further solutions or the problem is not solvable at the first place. Therefore, in order to enable consistency analysis, a single call of the `solve` function is sufficient to determine whether the feature model is consistent or not. In HEIMDALL, consistency checking is implemented in such a way that the SAT service computes either all possible solutions, or in case a threshold has been defined, up to that specified threshold. In the end, the SAT service returns the solutions appropriately to the frontend service.

Implementation of Consistency Checking. Listing 5.4 shows the implementation of consistency checking. Line 1 indicates that a `satSolver` object is required and that a threshold can be defined with the help of the `solutionThreshold` parameter. Line 2 shows the `solutions` array in which the results of the SAT solvers are stored. The `currentSolution` variable declared in line 3 is used to hold the respective current solution that we compute on line 5 with the help of the `solve` function for each iteration of a while loop. If no solution is left to be computed or no solution exists at all, the while loop is not executed anymore. Inside the while loop, on Line 6, the identifiers of the feature model elements for the current solution are fetched and assigned to the `trueVars` variable. This means that we fetch all `id` properties of every microservice and version that are part of a possible SAT solver solution. On line 7, the `trueVars` array is stored together with a representing solver result code, in this case `consistent`, as an object in the solutions array. Later we will also introduce some other values for the `satCode` variable. These variable is required by the frontend service to distinguish what kind of result has been computed by the SAT service. Line 8 represent a Logic Solver idiom to remove the current solution from the set of possible solutions so that for the next iteration of the while loop, another solution is fetched via the `solve` method on line 6. Lines 10 to 12 ensure that the while loop is left prematurely in case `solutionThreshold` is defined and the threshold has been reached.

```
1  (satSolver, solutionThreshold) => {
2    let solutions = [];
3    let currentSolution;
4
5    while ((currentSolution = satSolver.solve())) {
6      let trueVars = currentSolution.getTrueVars();
7      solutions.push({ satCode: 'consistent', fmElementIds: trueVars });
8      satSolver.forbid(currentSolution.getFormula());
9
10     if (solutionThreshold !== undefined && solutions.length === solutionThreshold) {
11       break;
12     }
13   }
14
15   return solutions;
16 }
```

Listing 5.4: Consistency Checking of Feature Model

Visualizing Consistent Feature Models. On the frontend side, each solution can eventually be visualized separately in a feature diagram via a simple dropdown selection. The visualization of the the possible solutions has been realized very similar to the examples shown in Section 4.3.2. Figures 5.4(a)–(d) show the four possible solutions that result from performing consistency checking on the feature model of the running example.

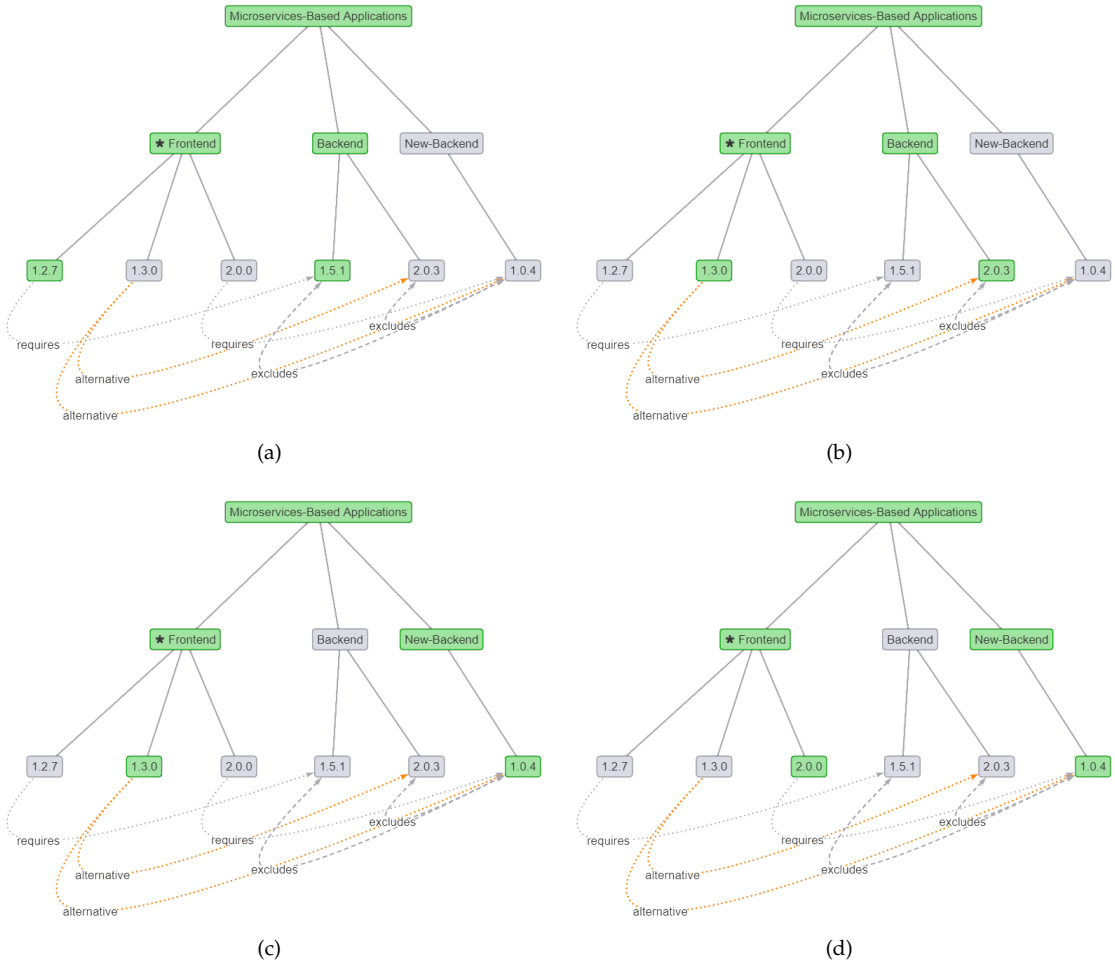


Figure 5.4: All Possible Solutions Resulting from the Consistency Check

Backtracking in Inconsistent Feature Models. Regarding inconsistent feature models, the Logic Solver on its own is only able to determine if a feature model is inconsistent. It does not give any suggestions on how to resolve inconsistencies. For that purpose, we have extended the correctness checking methods with backtracking capabilities in order to resolve inconsistent feature models. The idea is to check whether the feature model would become correct if one of the possible inconsistency causing Logic Solver expressions would be removed. In case removing this particular expression does not result in a consistent feature model, it is reused and another expression is removed in order to aim for a consistent model. In case the model is inconsistent

no matter which expression we remove, we repeat the whole procedure again, but this time we remove two formulas at any one time. If none of the removals of possible double combinations returns a consistent model, we continue with the removal of triple combinations of formulas and so forth. By this kind of backtracking, we ensure that we get the solutions to resolve the inconsistent feature models that have the least number of expressions that need to be removed in order for the model to become consistent. For the backtracking algorithm to work, we need to ensure to collect the Boolean formulas that are added to the SAT solver during the translation of a feature model. If consistency checking itself shows no possible solutions, the backtracking algorithm is started and the collected formulas are passed to the algorithm.

Backtracking Algorithm Implementation. Listing 5.5 shows the implementation of the backtracking algorithm. Line 1 depicts that two distinct arrays containing Boolean formulas are required: the `fmFormulas` array contains the formulas for the structural relations and the second array, i.e., `errorProneFMFormulas`, contains the formulas for the cross-tree constraints as well as the formulas that are required to define a microservice to be a mandatory feature. The former array directly stores the formulas as array elements but the latter one wraps them inside objects that have, on the one hand, a `formula` property that contains the actual boolean formula, and on the other hand, an `id` property that references the element of the feature model that is causing the respective formula definition. On line 2 the `backtrackSolutions` is defined in which solutions from the backtracking algorithm are stored. Line 4 represents a for loop which ensures that backtracking first tries to compute solutions that comprise one Boolean formula less of the `errorProneFMFormulas` array that exist in the original propositional formula and on every iteration that follows more formulas are incrementally removed from it. On line 5 all possible combinations of the formulas in `errorProneFMFormulas` with the current number of reduced formulas `curNumOfErrorProneFMFormulas` are computed with the help of a generatorics library `G`[16] and its `combination` method. The for loop on this line loops over each of these possible combinations whereas the current combination of formulas is held in the `errorProneFMFormulasCombination`. On line 7 a new solver is instantiated that will hold the shrunken propositional formula. Line 8 to 10 will conjoin the Boolean formulas of structural relations and line 11 to 13 will conjoin the formulas of the `errorProneFMFormulasCombination` array. Similarly to the consistency checking algorithm in Listing 5.4, the `solve` method of the solver is used to find possible solutions, just this time for a subset of the Boolean formulas. If a solution exists, line 17 to 24 compute which subset of Boolean formulas from the original propositional formula has not been used in the current combination of formulas. Several steps are required to achieve that. Line 17 uses the `filter` method from the popular underscore[17] JavaScript library to filter out `errorProneFMFormulas` that are part of the current combination of formulas. The corresponding filter criteria that is used on line 19 is based on the `_guid` attribute. This attribute is attached by Logic Solver to every Boolean formula that has been conjoined with the `require` method and it allows to uniquely identify a specific formula. After the computation of the harmful Boolean formulas, line 25 uses the `pluck` method from the underscore library to extract the ids of the responsible feature model elements. These are then stored to the `backtrackSolutions` array with `inconsistent` as a representing SAT solver result-code. After that, the current solution is rejected on line 27 to allow the detection of possible other solutions for the given propositional formula. Lines 31 to 33 ensure that backtracking is stopped in case the current number of removed inconsistency causing formulas lead to one or more solutions for resolving the inconsistent feature model. Finally, after inconsistency-causing feature model elements have been detected successfully, the corresponding solutions are returned on line 36.

[16] https://www.npmjs.com/package/generatorics
[17] http://underscorejs.org

```
(fmFormulas, errorProneFMFormulas) => {
  let backtrackSolutions = [];

  for (let curNumOfErrorProneFMFormulas = (errorProneFMFormulas.length-1);
     curNumOfErrorProneFMFormulas > 0; curNumOfErrorProneFMFormulas--) {
    for (let errorProneFMFormulasCombination of G.combination(errorProneFMFormulas,
     curNumOfErrorProneFMFormulas)) {

      let backtrackSolver = new Logic.Solver();
      for (let fmFormula of fmFormulas) {
        backtrackSolver.require(fmFormula);
      }
      for (let errorProneFMFormula of errorProneFMFormulasCombination) {
        backtrackSolver.require(errorProneFMFormula.formula);
      }

      let backtrackSolution;
      while ((backtrackSolution = backtrackSolver.solve())) {
        let culpritFormulas _.filter(errorProneFMFormulas, function(errorProneFMFormula) {
          for (let fmFormula of errorProneFMFormulasCombination) {
            if (errorProneFMFormula.formula._guid === fmFormula.formula._guid) {
              return false;
            }
          }
          return true;
        });
        let culpritElementIds = _.pluck(culpritFormulas, 'id');
        backtrackSolutions.push({ satCode: 'inconsistent', fmElementIds: culpritElementIds });
        backtrackSolver.forbid(backtrackSolution.getFormula());
      }
    }

    if (backtrackSolutions.length > 0) {
      break;
    }
  }

  return backtrackSolutions;
}
```

Listing 5.5: Backtracking for Inconsistent Feature Models

Visualizing Inconsistent Feature Models. The visualization for inconsistency causing features and constraints is quite similar to the visualizations introduced in Section 4.3.2. For example, Figure 5.5 shows possible outcomes of performing consistency checking on the inconsistent feature model introduced in Figure 4.4, which has been a slight alteration of the running example by defining all services to be mandatory. Figures 5.5(a)–(d) illustrate the four possible inconsistency causing elements that only require a single Boolean formula to be removed or adjusted in order to turn the Feature Model into a consistent state. Highlighted services indicate that the service should not be defined to be mandatory and highlighted constraints state that the particular constraint should not be defined between the corresponding microservice versions.

5.7.2 Dead Features

Detecting dead features, i.e. service versions or even microservices themselves, can be implemented as a small extension of the consistency checking code. Since the code in Listing 5.4 computes all possible solutions for a given feature model, the solutions just need to be reconciled with

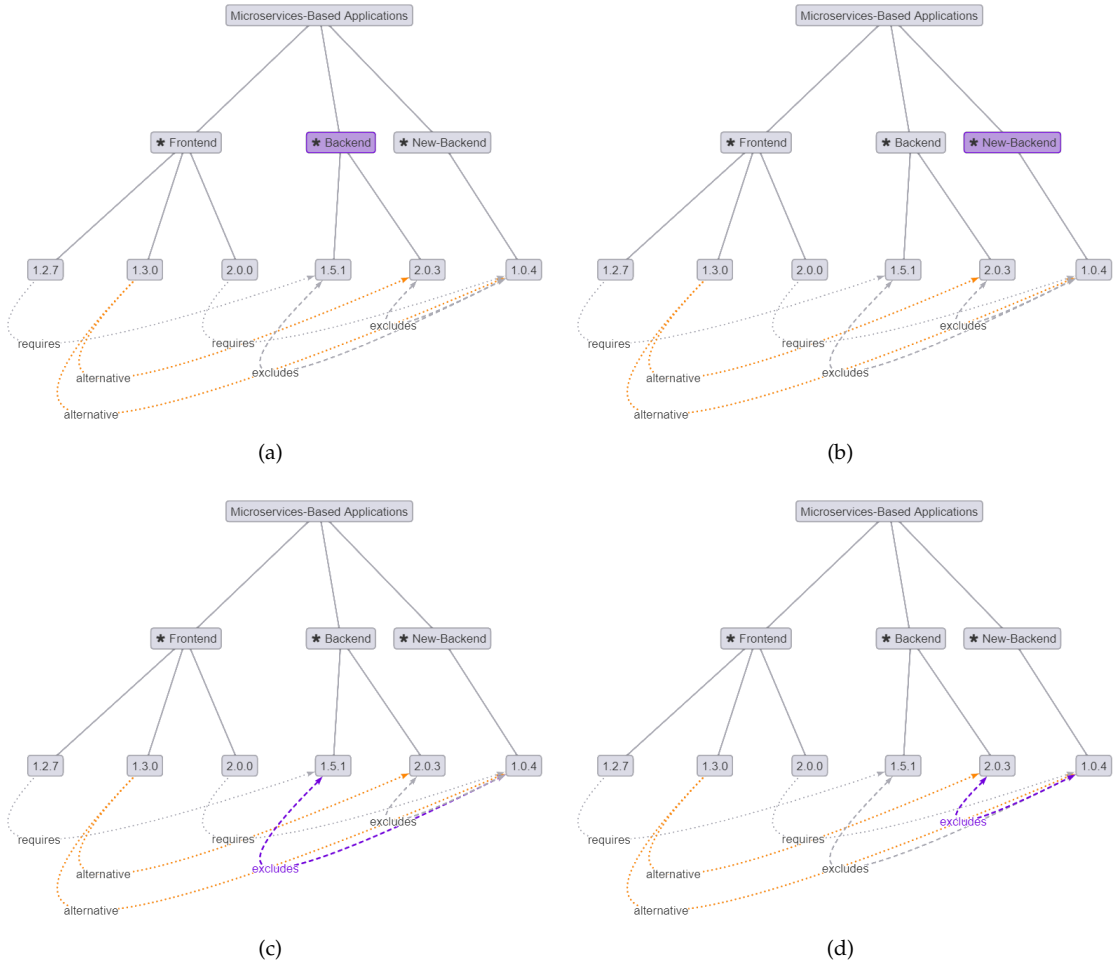


Figure 5.5: Backtracking Inconsistency Causing Feature Model Elements

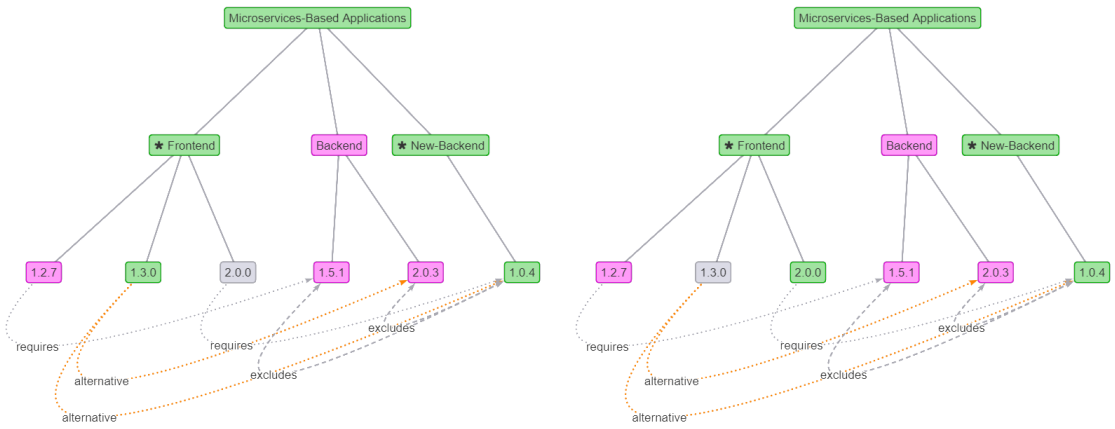
a list that holds all element identifiers of a given feature model. If one identifier from the list does not appear in any solution, it can be safely considered as a dead feature.

Implementation for Detecting Dead Features. Listing 5.6 shows the corresponding code extensions on the basis of Listing 5.4. Line 1 indicates that the element identifiers of a given feature model are available via the `fmElementIds` parameter. Line 2 introduces the `allTrueVars` array that is fed in every iteration of the while loop (line 7) with the ids of the current solution. The intention is to collect all feature model element identifiers that are comprised in the different solutions computed by the `solve` method. The if-condition on line 16 ensures that detection of dead features is only performed when actual solutions exist. Lines 17 to 19 use the `reject` method from the underscore library. It is applied to verify for each element of a feature model whether its identifier is part of the `allTrueVars` array and only the ones that do not appear in the `allTrueVars` are assigned to the `deadFeatures` array. Eventually on line 20, the list of dead features are added to the general solutions array, with `deadFeatures` as a representative label for the `satCode` property.

```
(satSolver, fmElementIds) => {
  let allTrueVars = [];
  let solutions = [], currentSolution;

  while ((currentSolution = satSolver.solve())) {
    let trueVars = currentSolution.getTrueVars();
    allTrueVars.push(...trueVars);
    solutions.push({ satCode: 'consistent', fmElementIds: trueVars });
    satSolver.forbid(currentSolution.getFormula());

    if (solutionThreshold !== undefined && solutions.length === solutionThreshold) {
      break;
    }
  }

  if (solutions.length > 0) {
    let deadFeatures = _.reject(fmElementIds, function(fmElementId) {
      return (allTrueVars.includes(fmElementId));
    });
    solutions.push({ satCode: 'deadFeatures', fmElementIds: deadFeatures });
  }

  return solutions;
}
```

Listing 5.6: Implementing the Detection of Dead Features

Visualizing Dead Features. The actual visualization of dead features is very similar to the corresponding illustrations in Section 4.3.2. Figures 5.6(a)–(b) show the same examples of dead features as the feature diagram in Figure 4.5 where the new-backend service of the running example has been defined to be mandatory. For this set-up, only two possible solutions exist. The dead features are highlighted in a different color compared to the highlighted features that form a specific feature model solution.



(a) (b)

Figure 5.6: Consistency Checking Combined with Detection of Dead Features

5.8 Interactive Feature Configuration

Configuration support is the second category of automated feature model analysis that we have introduced in Section 2.4.1. We have implemented an interactive service configuration support for the HEIMDALL application that assists the derivation process of microservices-based applications. The configuration system tries to guide the user with consistent configuration choices by validating the user's decision with respect to the dependencies between services. In order to emphasize the dependencies between services, the interactive feature configuration is heavily based on visualizing the selected features in the context of a service dependency graph. Uhle [UT14] has already highlighted that dependency graphs are often regarded as *program dependency graphs* in literature and that such graphs model how an application works internally, often on the granularity of software modules, control flow, or data. In HEIMDALL we use a very similar approach as Uhle [UT14], but instead of modelling service dependency graphs in the granularity of dependencies between microservices, we need to model the graphs in the granularity of dependencies between specific microservice versions in order to align with the concept of the underlying microservices-based feature models.

5.8.1 Dependency Graph for Feature Configuration

Figures 5.7(a)–(d) show the four possible valid feature configurations that exist for the feature model of the running example as dependency graphs. Each dependency graph mirrors one of the possible solutions that we have already illustrated in Figure 5.4. A dependency graph is implemented as a hierarchical *directed* network graph. We use a *directed* graph in order to be able to show which service depends on another service. The *directed* edges represent the dependencies and illustrate well which service is the consuming service and which is the producing one in the respective context. The hierarchical positioning is intended to accentuate the notion of upstream and downstream services. The services themselves are designed as hexagons; a shape that has been become quite common to indicate that the components at hand represent microservices. For example Sam Newman favours this particular representation of microservices for his slides [New15b]. Under each service a label illustrates its name and its selected version.

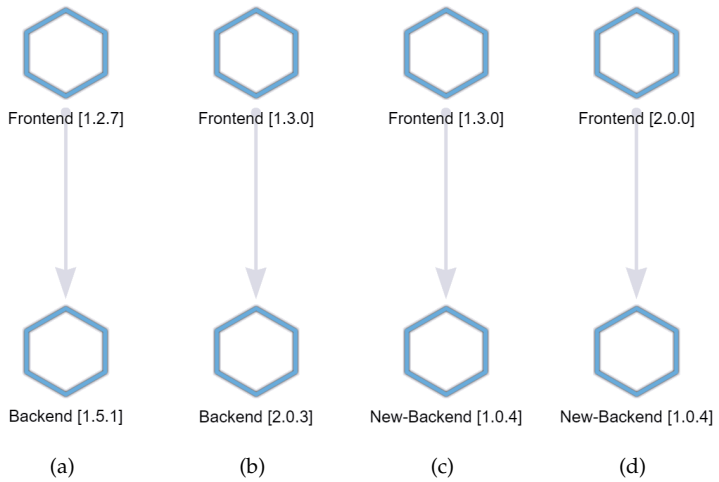


Figure 5.7: Feature Configurations Visualized as Dependency Graphs

5.8.2 Feature Configuration Process

Figures 5.8 illustrate the configuration process provided by HEIMDALL in the context of our running example. In Figure 5.8(a) it is shown that, in case the feature model is consistent, a *feature configuration* action can be performed on the basis of the current model. A list of the available microservices is placed on the right side of the screen, as shown in Figure 5.8(b). By clicking e.g. on the frontend service, it will be automatically placed on the graph space. As illustrated in Figure 5.8(c), it is now possible to choose a specific version for the current service which will be then appropriately appended to the label of the service. By clicking somewhere on an empty space on the graph area the service gets deselected and the list of the (remaining) services is available. Figure 5.8(d) shows that selecting the backend service creates it on the canvas next to the frontend service and again choosing a service version will label the backend service with the corresponding version tag. Having selected the current service with a defined version, allows us to model a dependency that originates from this current node. The corresponding dropdown lists the potential target nodes, thus as depicted in Figure 5.8(e), only the `Frontend [1.2.7]` service is currently listed. By adding it, the directed edge is accordingly created between the two services. Additionally, the dependency graph will recalculate the hierarchy of the nodes and correspondingly adjust the visualization of the hierarchical network, as captured in Figure 5.8(f).

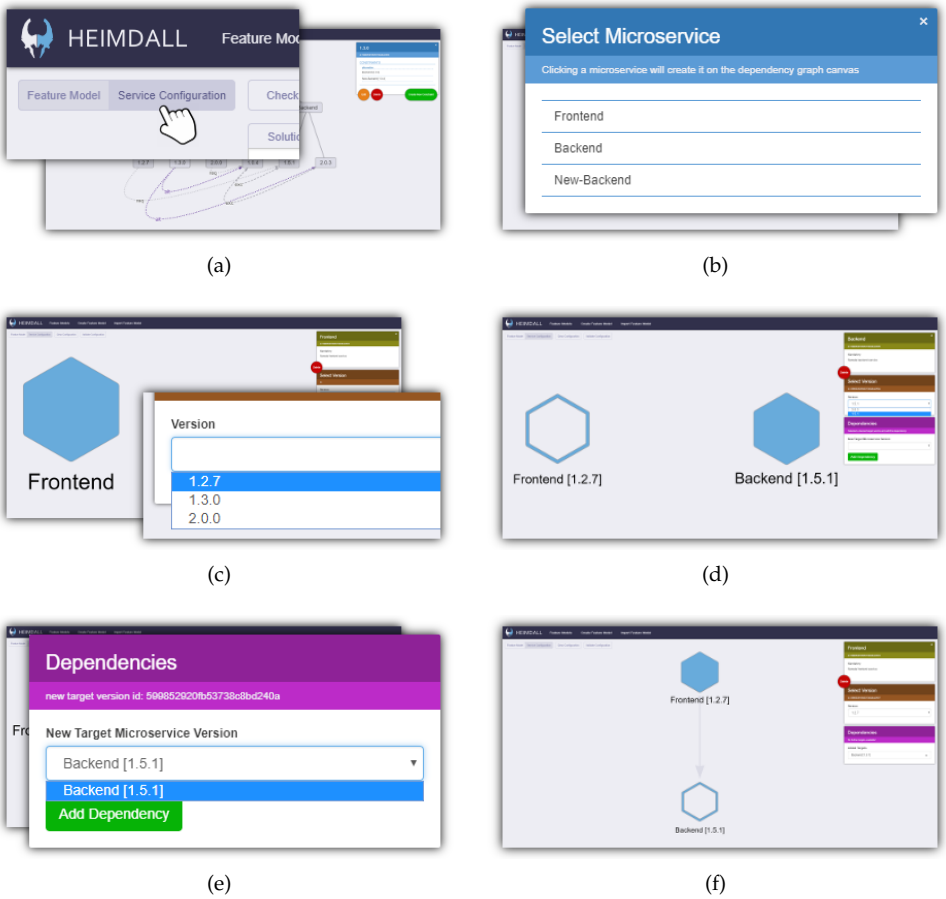


Figure 5.8: Service Configuration Process with HEIMDALL

5.8.3 Validating Feature Configurations

At any given time, the user is able to validate the current feature configuration based on the underlying feature model. The SAT service is responsible for validating the currently selected features and dependencies and to return meaningful responses. A feature configuration is basically an array that contains the ids of the currently used feature model elements in the dependency graph. The ids of microservices and versions always comply with the respective ids in the database. In contrast, certain dependencies might have arbitrary ids that have no counterpart in the database. This is the case if a dependency has been modelled between two services in the graph that has no corresponding constraint representation in the underlying feature model. We will discuss later how such cases can be treated. Generally, we have defined to detect three possible states that a given feature configuration can reflect, `legal`, `incomplete`, and `illegal` configurations. This desired analysis of feature configurations is only possible if conjoining the Boolean formulas of feature models is implemented a bit differently for feature configuration validation than it has been realized for correctness checking.

Conjoining Boolean Formulas for Feature Configurations. Listing 5.7 shows the adjusted code of Listing 5.3 and defines the propositional formula for feature models that are used for the validation of feature configurations. The `fm` object on Line 1 holds the complete data of a feature model whereas the `fc` array contains the ids of the elements that are selected for the feature configuration. Line 2 to 3 are the same as for correctness checking and also the start of the for loop over the microservices of the feature model on line 4 does not differ. On line 5 in contrast, an if-condition is wrapped around the whole code of the for loop since we only add Boolean formulas of structural relations that originate from services that are actually part of the current feature configuration. If true, lines 7 to 10 add the proper Boolean formulas for microservices, and additionally on line 11 also the id of the service feature is incorporated via the `require` method. The latter ensures that at the end the SAT solver will only find solutions that also comprise this specific service. The very same concept is also applied to the versions of services as indicated by line 15 and 18. The constraint logic is also adjusted a bit in order to cope with the increased emphasis on dependencies between microservices. Since the product derivation process is realized with the help of a dependency graph, dependencies are explicitly modelled and defined for a feature configuration. Therefore these dependencies need also explicitly be conjoined to the propositional formula of feature models. Line 23 and line 31 respectively add the proper Boolean formulas so that the version demands the explicit selection of `requires` and `alternative` constraints that originate from it. The lines 24 and 32 then check whether this constraints have been included into the feature configuration `fc` and correspondingly add the constraint ids to the propositional formula. Since `excludes` constraints are neither modelled in a dependency graph nor actually required for the configuration process, they just need to be added in the same manner to the propositional formula as it is realized for correctness checking.

```
1 (fm, fc) => {
2   let satSolver = new Logic.Solver();
3   satSolver.require(fm.id);
4   for (let microservice of fm.microservices) {
5     if (fc.includes(microservice.id) || microservice.mandatory === true) {  // fc
6
7       satSolver.require(Logic.implies(microservice.id, fm.id));
8       if (microservice.mandatory === true) {
9         satSolver.require(Logic.implies(fm.id, microservice.id));
10      }
11      solver.require(microservice.id);  // fc
12
13      let versionIds = [];
14      for (let version of microservice.versions) {
```

```
15          if (fc.includes(versionId)) { // fc
16            satSolver.require(Logic.implies(version.id, microservice.id));
17            versionIds.push(version.id);
18            solver.require(version.id); // fc
19
20            for (let constraint of version.constraints) {
21              if (constraint.constraint_type === 'requires') {
22                satSolver.require(Logic.implies(version.id, targets[0]));
23                satSolver.require(Logic.implies(version.id, constraint.id));    // fc
24                if (fc.includes(constraint.id)) { solver.require(constraint.id); }  // fc
25              }
26              else if (constraint.constraint_type === 'excludes') {
27                satSolver.require(Logic.not(Logic.and(version.id, targets[0])));
28              }
29              else if (constraint.constraint_type === 'alternative') {
30                satSolver.require(Logic.implies(version.id, Logic.exactlyOne(constraint.targets)));
31                satSolver.require(Logic.implies(version.id, constraint.id));    // fc
32                if (fc.includes(constraint.id)) { solver.require(constraint.id); }  // fc
33              }
34            }
35          }
36        }
37        satSolver.require(Logic.implies(microservice.id, Logic.exactlyOne(versionIds)));
38      }
39    }
40    return satSolver;
41  }
```

Listing 5.7: Translating Feature Models for Validating Feature Configurations

Implementation of Feature Configuration Validation. A feature configuration is called to be legal in case the configuration fits the constraints of the underlying feature model. The feature configuration is already appropriately conjoined to the propositional formula of the SAT solver, thus detecting legal feature configurations now has almost exactly the same underlying code as the one shown for the consistency checking in Listing 5.4. Listing 5.8 shows the corresponding implementation where the only difference is reflected by the `satCode` attribute containing the String `legal` as value.

```
1  (satSolver) => {
2    let solutions = [];
3    let currentSolution;
4
5    while ((currentSolution = satSolver.solve())) {
6      let trueVars = currentSolution.getTrueVars();
7      solutions.push({ satCode: 'legal', fmElementIds: trueVars });
8      satSolver.forbid(currentSolution.getFormula());
9    }
10
11   return solutions;
12 }
```

Listing 5.8: Validating Legal Feature Configurations

Visualizing Legal Feature Configurations. With respect to the four possible solutions of the running example, the corresponding legal feature configurations are appropriately visualized by highlighting the elements of the dependency graph in green colors, as shown in Figures 5.9(a)–(d).

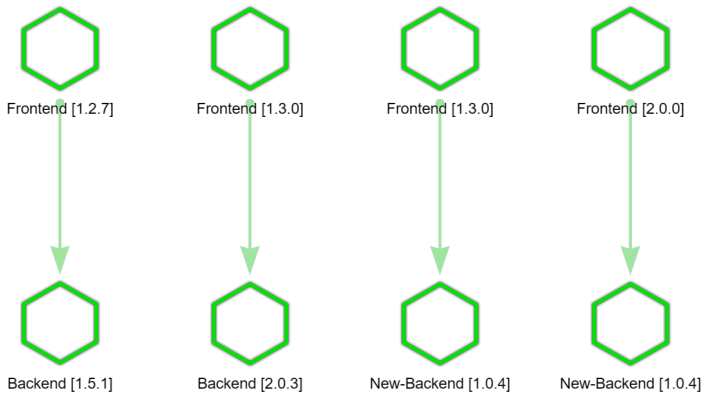


Figure 5.9: Legal Feature Configurations Visualized as Dependency Graphs

Computing Suggestions for Incomplete Configurations. In the strict sense, incomplete configurations fall into the category of illegal configurations. However, we have already indicated in Section 2.4.4 that such configurations can be treated in a more beneficial manner. With the help of the SAT solver, actual suggestions that may help to resolve incomplete configurations can be proposed. During the preparation of the propositional formula shown in Listing 5.7 we only add the Boolean formulas for structural relations and cross-tree constraints which are indeed demanded by the elements selected for the feature configuration. With that, the usage of the `solver` method does actually not only return solutions for complete feature configurations, but also returns the ids of elements in the feature model that would be valid suggestions that help to continue the completion of the configuration. Listing 5.9 shows an extended version of the validation code. Line 7 represents the main change that is necessary to check whether some element ids returned by the solver are not covered by the element ids of the feature configuration and are therefore indicating suggestions how to continue on an incomplete feature configuration. The `difference` method from the underscore JavaScript library is used to return the values from `trueVars` that are not present in `fc`. On lines 9 to 14 the code is correspondingly adjusted to either store solutions for legal configurations or for incomplete ones.

```
1  (satSolver) => {
2    let solutions = [];
3    let currentSolution;
4
5    while ((currentSolution = satSolver.solve())) {
6      let trueVars = currentSolution.getTrueVars();
7      let suggestedElementIds = _.difference(trueVars, fc);
8
9      if (suggestedElementIds.length === 0) {
10       solutions.push({ satCode: 'legal', fmElementIds: trueVars });
11     }
12     else {
13       solutions.push({ satCode: 'incomplete', fmElementIds: suggestedElementIds });
14     }
15     satSolver.forbid(currentSolution.getFormula());
16   }
17
18   return solutions;
```

```
19 }
```

Listing 5.9: Validating Incomplete Feature Configurations

Visualizing Incomplete Feature Configurations. With respect to the running example, an incomplete feature configuration might be at hand, when only the *Frontend [1.2.7]* service has been created so far on the dependency graph, as depicted in Figure 5.10(a). With the help of the SAT solver, actual suggestions that may help to resolve the incomplete configuration can be proposed. As illustrated in Figure 5.10(b), such elements are colored in orange. By clicking on a suggested service, the microservice as well as the dependency to it is automatically added to the dependency graph, as shown in Figure 5.10(c).

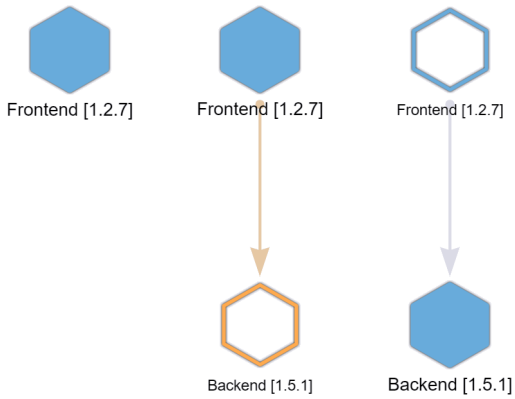


Figure 5.10: Incomplete Feature Configurations Visualized in Dependency Graphs

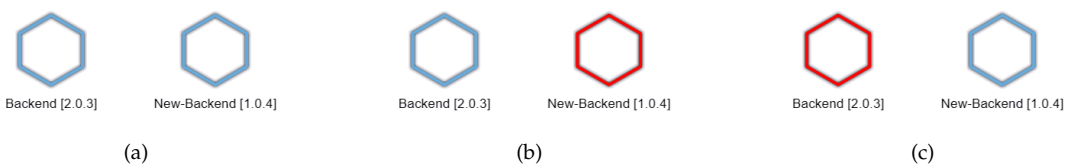
Resolving Illegal Feature Configurations with Backtracking. An illegal feature configuration is a configuration that violates the constraints of the feature model, thus the SAT solver can not solve the underlying propositional formula. Logic Solver on its own is only able to determine if a configuration is illegal, but not which elements are causing the illegal state. This is very similar to the situation with inconsistent feature models in correctness checking. The solver is not able to give suggestions on how to resolve an illegal configuration. Therefore we have extended the feature configuration validation methods with backtracking capabilities too. Again, the Boolean formulas that are added to the SAT solver during the translation of a feature model need to be collected for the backtracking algorithm. If configuration validation itself shows no possible solutions, the backtracking algorithm is started. Listing 5.10 shows the backtracking code for illegal feature configurations. The `fmFormulas` parameter on line 1 indicates that we need the Boolean formulas of the feature model and the `fc` variable holds the element ids of the feature model that form the feature configuration. The basic structure of this algorithm is exactly the same as the algorithm for the Backtracking in inconsistent feature models shown in Listing 5.5. The difference this time is that we do not reduce the amount of Boolean formulas of the feature model one by one, but we steadily reduce the amount of element ids contained in the feature configuration. Thus line 3 and 4 differ in that aspect and therefore loop over a decreasing amount of

combinations of the `fc` array. Also lines 10 to 12 do then include the reduced amount of element ids from the current feature configuration combination. If the `solve` method in line 15 is successful, the `backtrackTrueVars` array in line 16 fetches the subset of element ids from the feature configuration that would result in a `legal` configuration. On line 17 we then use again `difference` method from underscore to return the values from the `fc` array that are not present in the `backtrackTrueVars` so that finally the element ids that are causing the configuration to be illegal are stored in the `illegalElementIds` array. These ids are then appropriately pushed to the solutions array on line 18.

```
1  (fmFormulas, fc) => {
2    let backtrackSolutions = [];
3    for (let currentNumOfFmElementIds = (fc.length-1); currentNumOfFmElementIds > 0;
       currentNumOfFmElementIds--) {
4      for (let fcCombination of G.combination(fc, currentNumOfFmElementIds)) {
5
6        let backtrackSolver = new Logic.Solver();
7        for (let fmFormula of fmFormulas) {
8          backtrackSolver.require(fmFormula);
9        }
10       for (let fmElementId of fcCombination) {
11         backtrackSolver.require(fmElementId);
12       }
13
14       let backtrackSolution;
15       while ((backtrackSolution = backtrackSolver.solve())) {
16         let backtrackTrueVars = backtrackSolution.getTrueVars();
17         let illegalElementIds = _.difference(fc, backtrackTrueVars);
18         backtrackSolutions.push({ satCode: 'illegal', fmElementIds: illegalElementIds });
19         backtrackSolver.forbid(backtrackSolution.getFormula());
20       }
21     }
22
23     if (solutions.length > 0) {
24       break;
25     }
26   }
27   return backtrackSolutions;
28 }
```

Listing 5.10: Backtracking in Illegal Feature Configurations

Visualizing Illegal Feature Configurations. Figure 5.11(a) shows a feature configuration for which two specific services from the running example are selected, the *Backend [2.0.3]* and the *New-Backend [1.0.4]* service. Since both services exclude each other, the SAT solver can not compute a valid solution and thus the backtracking algorithm needs to be used to find errors in the feature configuration. In this case, either *New-Backend [1.0.4]* or *Backend [2.0.3]* needs to be removed, as respectively shown in Figure 5.11(b) and Figure 5.11(c).



(a) (b) (c)

Figure 5.11: Illegal Feature Configurations Visualized in Dependency Graphs

The Special Case of Illegal Dependencies. One aspect that hasn't been covered yet, is that feature configurations could potentially comprise dependencies that do not exist at all in the underlying feature model. Technically, this means that a feature configuration can contain element ids that do not reference to any valid element of the feature model. During the preparation of the propositional formula in Listing 5.7 unknown constraint ids are neglected at all due to the lines 24 and 32. Thus the SAT solver of the validation algorithm can not know about these unknown ids and therefore also wouldn't be able to detect that the configuration is illegal. Fortunately, such cases can be easily covered during the validation of feature configurations and we only need to extend the code in Listing 5.9 appropriately. Listing 5.11 shows the extended code, whereas actual changes are rather small. On line 8 we fetch the element ids from the feature configuration that are not part of the current possible solution. This is actually only possible for not existing dependency ids thus we store them in the *notExistingDependencyIds* array and push them on line 17 with an appropriate `satCode` to the solutions array.

```
1  (satSolver) => {
2    let solutions = [];
3    let currentSolution;
4
5    while ((currentSolution = satSolver.solve())) {
6      let trueVars = currentSolution.getTrueVars();
7      let suggestedElementIds = _.difference(trueVars, fc);
8      let notExistingDependencyIds = _.difference(fc, trueVars);
9
10     if (suggestedElementIds.length === 0 && wrongDependencies.length === 0) {
11       solutions.push({ satCode: 'legal', fmElementIds: trueVars });
12     }
13     else if (notExistingDependencyIds.length === 0) {
14       solutions.push({ satCode: 'incomplete', fmElementIds: suggestedElementIds });
15     }
16     else {
17       solutions.push({ satCode: 'illegalDependencies', fmElementIds: notExistingDependencyIds });
18     }
19     satSolver.forbid(currentSolution.getFormula());
20   }
21
22   return solutions;
23 }
```

Listing 5.11: Validating Feature Configurations Containing not Existing Dependencies

Visualizing Illegal Dependencies in Feature Configurations. Figure 5.12(a) shows a feature configuration containing the `Frontend [1.2.7]` and `Backend [2.0.3]` service versions with a dependency from the former to the latter. The SAT solver validation algorithm detects that such a dependency does not exist at all and therefore it will be highlighted accordingly as shown in Figure 5.12(b).

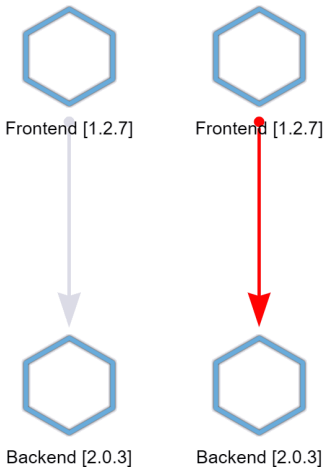


Figure 5.12: Illegal Dependencies Visualized in Dependency Graphs

6

Evaluation

In this chapter, we evaluate the automated analysis capabilities of HEIMDALL's SAT service. We focus on a quantitative evaluation to determine the performance and the results of the different automated analysis aspects when used in practice. The evaluation experiments are performed with a multitude of differently sized microservices-based feature models.

6.1 General Setup

All tests were performed on a laptop with Windows 10 Home as operating system and an Intel Core i7-4702MQ CPU @ 2.20 GHz with 32GB of RAM. The results of every experiment have been collected each in a comma-separated values (CSV) files. The CSV result files and the scripts that have been used to perform the experiments can be found in the repository that is defined in Appendix A.3.

6.2 Randomly Generated Feature Models

Generally, one challenge in analyzing feature models is the lack of large scale real models [MWC09]. The same holds true in the microservices domain, where insight in actual real world architectures of microservices-based systems is rarely available, especially for larger scale applications. In order to still have a solid foundation of microservices-based feature models, we implemented a random generator for such models.

6.2.1 Random Feature Model Generator

On the one hand, the generator allows us to specify some parameters that affect the properties of a feature model. On the other hand, the generator itself enforces certain characteristics that shape all microservices-based feature models used for the quantitative evaluations.

Parameters. The generator allows regulating the creation of feature models with the following four parameters:

- **Number of Services:**
  Probably the most important parameter is to enable specifying how many microservices the randomly generated model should comprise.

- **Expected Versions per Service:**
  Even though the quantity of available versions differs from service to service, we allow to emphasize the amount per service around a given number. This can also be a decimal number as it is used as input for the expected value for a normal distribution of versions, as further discussed in the next paragraph.
- **Ratio of Excludes Constraints:**
  Excludes constraints are the essential cause for inconsistent models. For experimentation purposes we allow regulating the number of excludes constraints as a ratio compared to the number of all possible connections between service versions. Thus a ratio of 100% implies that every single service version has excludes constraints to all versions of all other microservices.
- **Number of Excludes Constraints:**
  Alternatively to the ratio of excludes constraints, also a parameter for defining a fixed number of such constraints is allowed.
- **Ratio of Mandatory Services:**
  Mandatory services allow specifying which services should be absolutely contained in a feature configuration. The parameter allows controlling the ratio of services that will be defined by the generator to be mandatory. For n number of services, the ratio r defines $n/100 * r = m$ mandatory services.

Characteristics. The generator creates feature models that comply with the following characteristics:

- **Normal Distribution for Versions:**
  The number of versions for a certain microservice is depending on the parameter for expected versions per service n. The parameter n is actually the input for generating random numbers with a normal (Gaussian) distribution. The expected value as well as the standard deviation are defined by n. If the proposed number falls below 1 or exceeds the upper bound that is defined as $(2 * n) + 1$, the random number generation is triggered again. This rule safely defines an upper bound for unlikely runaway values and also ensures that at least one version is created for each service. In case the parameter n is set to 0, the underlying random value generator would end up in an endless loop. In order to prevent that, a parameter value of 0 is interpreted in such a way, that each service gets supplied with exactly just one single version.
- **Allow for Root Services:**
  After the creation of services and versions, dependencies are created in a specifically controlled manner. We iterate over the randomly generated services one by one and for each service we also iterate over the versions of a service. On each version we ensure that no dependencies are created towards the versions of services over which we have already iterated before. The first iteration of the dependency creation process handles therefore the first service which can not be targeted by any constraint originating from versions of subsequent services. This produces microservices-based applications that have at least one top level service, e.g. a frontend service or an API gateway.
- **No Circular Dependencies:**
  By avoiding creating dependencies to versions over which we have already iterated before, we also avoid the creation of circular dependencies between services. This characteristic is highly desired since circular dependencies should not appear in a microservices architecture at all.

- **Normal Distribution of Dependencies:**
  The number of other services that a given service depends on is similar to the distribution mechanism that is applied when creating versions. We assumed for the evaluation experiments that the expected value for the normal distribution is based on a fraction of the specified number of services n. Additionally, it is also multiplied by a factor that is constantly decreasing with the number of services n. This rule ensure that services in larger microservices-based architectures may potentially depend on more microservices than they would do in smaller architectures, but it also prevents the number of dependencies to become unlikely high for large scale applications. One specific version v_x of a service s_a can then have one to n dependencies to a subset of the versions $v_1, ..., v_m$ of the other service s_b, where m is the total number of versions of s_b and $0 < n <= m$. The number of dependencies n_1 that the first version v_1 of service s_a has, is a random number between 1 and m. The next version v_2 of the service s_a has then a number of dependencies that is a random number between n_1 and m. With that we ensure that version v_2 of service s_a does not depend on lower versions of s_b than version v_1 of service s_a does. A single dependency is modelled either with a *requires* constraint when a certain version v_x of service s_a depends on exactly one specific version of the other service s_b, or with an *alternative* constraint, when the version v_x of service s_a depends on one of the multiple possible other versions $v_1, ..., v_m$ of s_b.

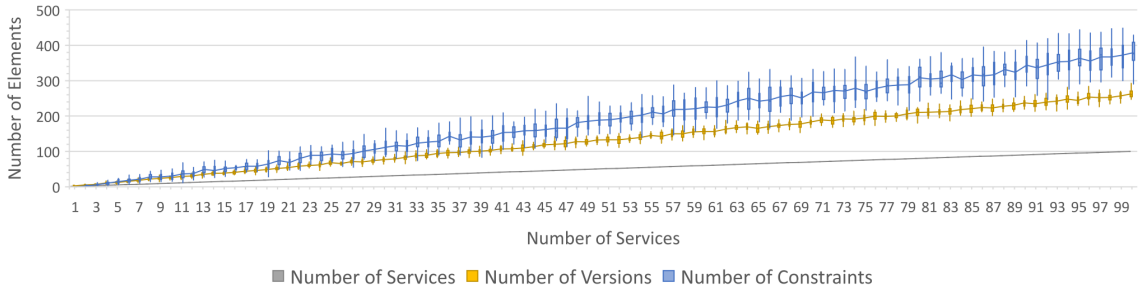
6.2.2 Examples of Randomly Generated Feature Models

In the following we perform a test run that should reveal the numbers of services, versions, and constraints of feature models that get created by the random feature model generator.

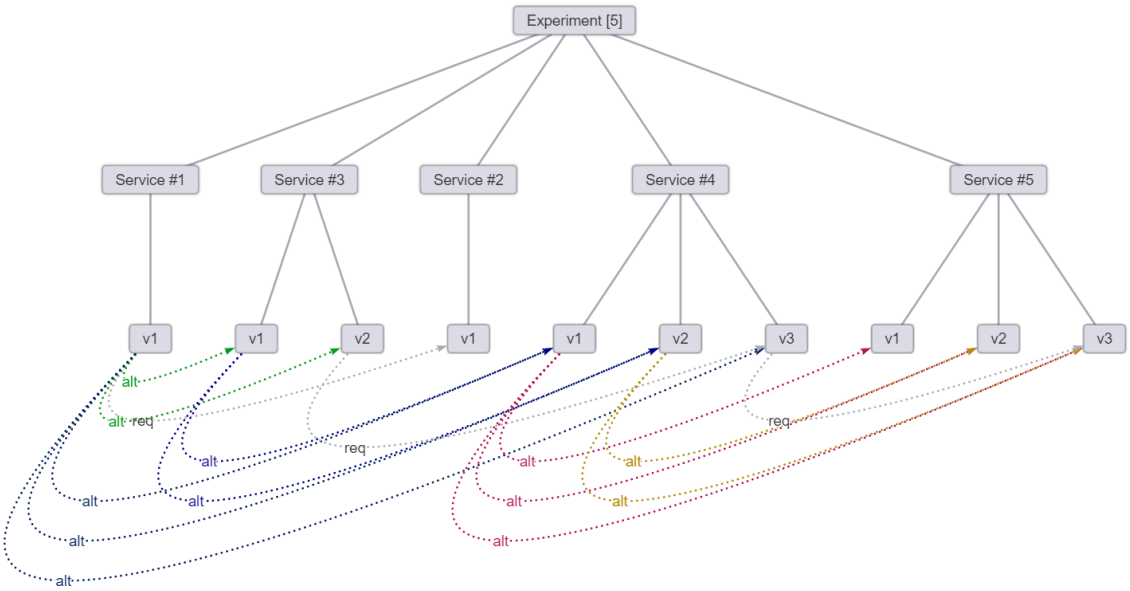
Parameters. We have specified the following parameters for these feature models; mandatory services and excludes constraints have been neglected the test run since they do not alter the tendency of the numbers for versions and constraints with respect to an increasing service number. Furthermore, the expected versions per service has been set to 2 which results in services having one to five versions. We assume this to be a realistic number since we guess that there is rarely the need for software and DevOps engineers to manage service configurations where more than five different versions of a service exist at the same time. We have created feature models ranging from one service to 100 services and for each number of services 25 sample models have been generated.

- Numbers of services: 1 - 100
- Expected versions per service: 2
- Ratio of Excludes Constraints: 0
- Ratio of Mandatory Services: 0

Results. Figure 6.1(a) shows the numbers of versions and constraints that have been created for the different amounts of services. The numbers are linearly growing, even for constraints, which we assume form a solid basis of feature models for the experiments that follow. Figure 6.1(b) illustrates one of these microservices-based feature models.



(a) Statistics of Randomly Generated Feature Models



(b) Example of a Randomly Generated Feature Models

Figure 6.1: Randomly Generated Feature Models

6.3 Evaluation of Correctness Checking

We now perform a set of different experiments that analyze the performance and the results of the different methods available for correctness checking.

6.3.1 General Setup for Correctness Checking

Generally each experiment for correctness checking runs 25 times for each single service size. In each run a random feature model gets generated, correctness checking is performed, time measurements and complementary statistics are stored into the CSV file of the current experiment, and finally the feature model gets deleted again. For the different experiments, a *threshold for the solution finding process* can be defined to determine how many solutions should be computed at most. In case the threshold is reached, dead feature detection can not be performed since all solutions of a feature model are required to enable the identification of dead features. Since multiple runs are performed for each service size, the results are visualized as box plots in order to

give a notion about the variance of the results for each particular service size. The following time measurements are performed for the different correctness checking aspects:

- **Translation:**
  The time it takes to perform the translation of feature models to propositional formula.
- **Consistency Checking:**
  The duration of consistency checking, thus finding a first solution for the feature model.
- **Solution Finding:**
  In case the feature model is consistent, the time that is required to either find all possible solutions of a feature model or the number of solutions that has been predefined as upper bound with the threshold for the solution finding.
- **Dead Feature Detection**:
  The duration of analyzing whether certain features appear not once in any of all the possible solutions for a feature model.
- **Backtracking:**
  In case the feature model is inconsistent, this holds the time that is consumed by the backtracking algorithm in order to find the contradictions in the feature model.
- **Overall:**
  The time that is consumed by the whole correctness checking method. With respect to the implementation explained in Section 5.7, the overall time can be composed of different parts, depending on the consistency of the feature model, the threshold for solution finding, and the number of found solutions:
  - **Consistent Feature Model** *[Threshold == 1[* **:**
    The sum of *translation* and *consistency checking*.
  - **Consistent Feature Model** *[Threshold > 1, Solutions < Threshold]* **:**
    The sum of *translation*, *solution finding*, and *consistency checking*.
  - **Consistent Feature Model** *[Threshold > 1, Solutions <= Threshold]* **:**
    The sum of *translation*, *consistency checking*, *solution finding*, and *dead feature detection*.
  - **Inconsistent Feature Model** *[Solutions == 0]* **:**
    The sum of *translation*, *consistency checking*, and *backtracking*.

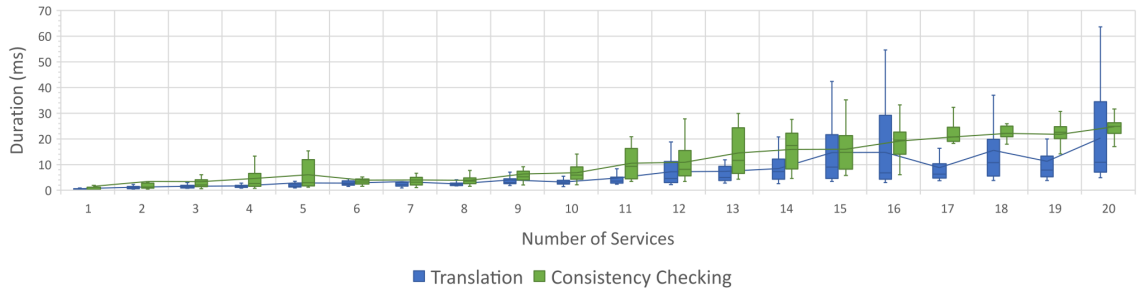
6.3.2 Consistency Checking and Dead Feature Detection

The first experiment intends to show the time it takes to translate a feature model to propositional formula, perform consistency checking, and fetching all solutions in order to detect dead features.

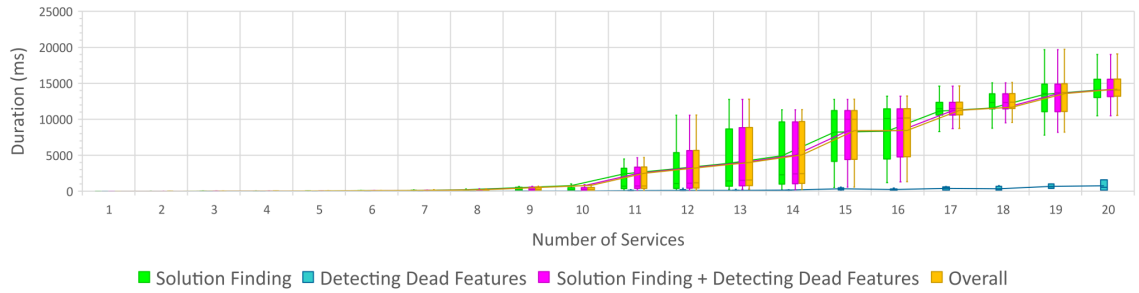
Experiment Parameters. Basically the same parameters for the feature models as for the test run in Section 6.2.2 are used. Excludes constraints and mandatory services are again neglected in order to ensure as far as possible that the feature models are consistent. The difference is that this time the feature models range from one service to 20 services. Correctness checking has been parametrized with a threshold of $30'000$ solutions since computing more solutions sometimes proved to bring Logic Solver to its knees, which trows a memory error in such cases.

- Numbers of services: 1 - 20
- Expected versions per service: 2
- Ratio of excludes constraints: 0
- Ratio of mandatory services: 0
- Threshold for solution finding: $30,000$

Results. Figure 6.2 shows the different time measurements for correctness checking. Since only consistent feature models have been validated time measurements for translation, consistency checking, solution finding, and dead feature detection have been captured. Figure 6.2(a) opposes the translation time to the consistency checking time. The duration for the former increases linearly the more services exist for a feature model, but even though the latter has similar tendencies, the performance is varying increasingly the more services exist. Both aspects need way less time than solution finding and feature detection therefore the other measurements are listed in a separate graph. Figure 6.2(b) depicts the times that it took to find all solutions for every feature model. Compared to translation and consistency checking, finding all possible solutions of a given feature model takes much longer. The detection of dead features is then again less time consuming, but since its prerequisite is to have all solutions at hand, a complete dead feature detection needs to comprise both methods. Still, the performance impact of the latter is negligible to the former. Finally, the time measurements for the whole correctness checking clearly shows that finding all solutions is primarily responsible for the correctness checking time.



(a)



(b)

Figure 6.2: Time Measurements of Correctness Checking

Figure 6.3 shows the amount of solutions computed with respect to the number of services. The amount of results starts varying much more with an increasing number of services. Since dead feature detection requires the computation of all possible solutions, it can't be performed for feature models that have a higher number of solutions than the solutions threshold. Thus, the average number of solutions flattens the more services a feature model possesses. The threshold is reached most of the times for the feature models that have a service size of 17 and higher. This indicates that the dead feature detection capability by HEIMDALL's SAT service is solely feasible for feature models comprising low numbers of microservices.

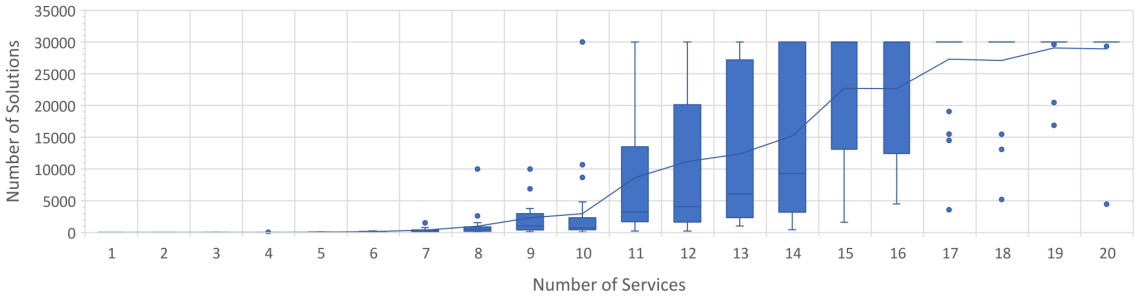


Figure 6.3: Solutions of Correctness Checking

6.3.3 Large Scale Consistency Checking

Since dead feature detection is quite costly, we perform a new experiment where we disable dead feature detection and focus on consistency checking of large scale microservices-based feature models.

Experiment Parameters. This experiment is performed with a threshold of 1 for the solution finding. This allows us to check whether the feature models are consistent, but dead feature detection is completely neglected. The feature models range from 10 to 200 services in steps of 10 services.

- Numbers of services: 10 - 200
- Expected versions per service: 2
- Ratio of excludes constraints: 0
- Ratio of mandatory services: 0
- Threshold for solution finding: 1

Results. Figure 6.4 shows that translation as well as consistency checking is increasing linearly at a very low grade. Therefore also the overall correctness checking performance increases at a very low rate for increasing numbers of services in the feature models. With that, consistency checking scales quite well and can be performed in a feasible amount of time for microservices-based feature models comprising hundreds of different services.

Figure 6.4: Time Measurements for Translation, Consistency Checking and Overall Duration

Results. Figure 6.5 shows results to a minimally adjusted alteration of the experiment. The difference is that this time the same feature model has been used for all 25 experiment runs per each number of services. The intention here is to depict the performance variation of the translation and consistency checking methods when executed on the same feature model. As the figure shows, the methods perform decently stable up to feature models with 120 services. Even though the average duration increase stays the same for higher number of services, the performance varies a bit more from then on, probably due the increased load.

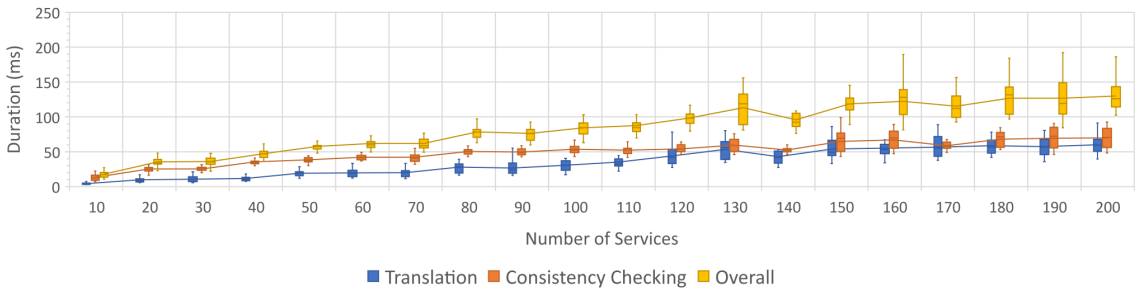


Figure 6.5: Time Measurements for Translation, Consistency Checking and Overall Duration

6.3.4 Enable Dead Feature Detection for Larger Scales

Dead feature detection has been proven to be quite costly since feature models with increasing service numbers result quickly in unmanageable amounts of possible solutions. One aspect that has not been considered yet is that lowering the number of versions per service as well as an increasing of mandatory services and excludes constraints lower the amount of possible solutions for a feature model. The next experiment focuses on how much these factors impact the possible result set of correctness checking and its performance.

Experiment Parameters. This experiment is executed in four different ways. First we solely increase the ratio of mandatory services and then we do the same for the ratio of excludes constraints. After that, only the expected number of services is decreased and in the final experiment, we use all three result lowering parameters together to test their combined impact on a larger scale of feature models. A ratio of 10% has been chosen for the mandatory services, a ratio of 0.1% for the ratio of excludes constraints, and the expected number of versions per service has been set to 1.75. For all experiments we reestablish the thresholds for solutions to 30'000 since low adjustments might still result in too many solutions.

Parameters Separated:
- Numbers of services: 20
- Expected versions per service: $0 - 2$
- Ratio of excludes constraints: $0 - 100\%$
- Ratio of mandatory services: 0 - 100%
- Threshold for solution finding: $30,000$

Parameters Combined:
- Numbers of services: 5 - 50
- Expected versions per service: 1.75
- Ratio of excludes constraints: 0.1%
- Ratio of mandatory services: 10%
- Threshold for solution finding: $30,000$

Results. As shown in Figure 6.6, increasing the ratio of mandatory services quickly reduces the number of solutions and the duration of correctness checking. However, since the feature models are randomly generated, the number of solutions may still vary a lot and the ratio of mandatory

services only allows a vague prediction about its influence on the size of the result set. In contrast, the number of solutions directly correlates to the correctness checking duration. This is mainly because finding all solutions for a feature model occupies most of the correctness checking length.

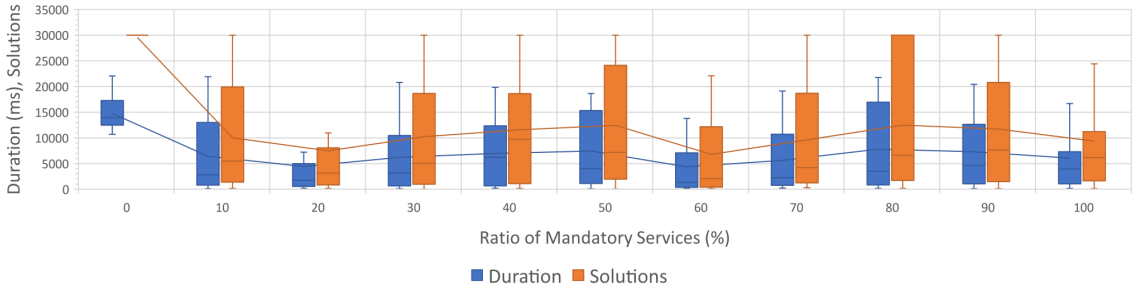


Figure 6.6: Increasing Ratio of Mandatory Services

Figure 6.7 shows that increasing the ratio of excludes constraints also quickly reduces the amount of possible solutions in the result set. In contrast to mandatory services, excludes constraints steadily decrease the possible amount of solutions. Same as before, the time spent for correctness checking directly correlates to the possible result set.

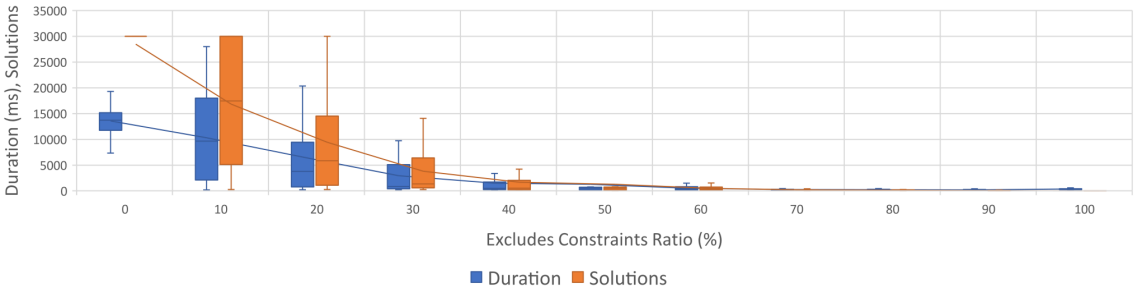


Figure 6.7: Increasing Ratio of Excludes Constraints

Reducing the expected number of versions per service is also effectively decreasing the number of possible solutions as shown in Figure 6.8. Furthermore, the correlation between the duration and the size of the result is depicted in these measurements too.

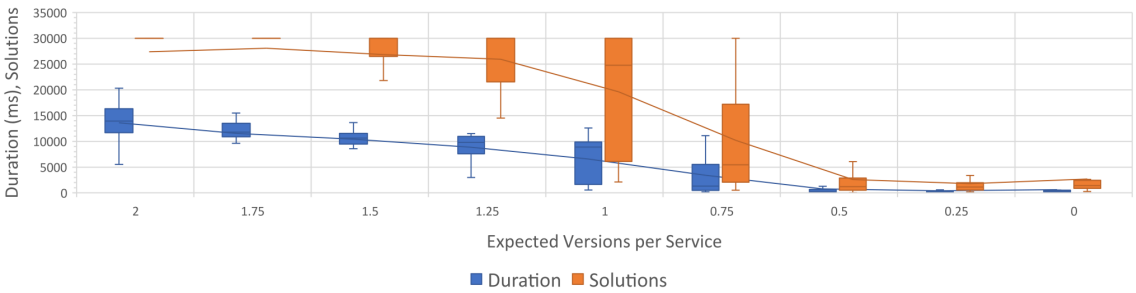


Figure 6.8: Decreasing the Number of Expected Versions per Service

Figure 6.9 shows the impact of combining a set of fixed parameters on several feature models of different sizes. Compared to the experiment in Section 6.3.2, feature models with up to 30 services won't reach the threshold of $30,000$ solutions this time. The more services, mandatory services, versions, constraints, and especially excludes constraints exist, the more does the result set size vary. Especially feature models with more than 30 services tend to show either result sets that exceed the threshold, or result sets of size 0, indicating the feature model to be inconsistent.

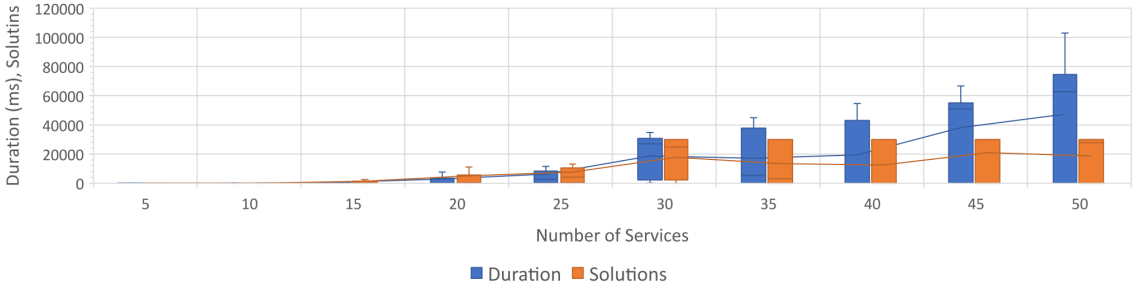


Figure 6.9: Combining Result Lowering Parameters

6.3.5 Analyzing Inconsistent Feature Models

As previously shown, already small percentages of mandatory services and excludes constraints can introduce inconsistency in feature models. The SAT service of the HEIMDALL application has an implemented backtracking algorithm which helps to detect the features and constraints that cause feature models to be inconsistent. The performance of the backtracking algorithm is the subject that the next experiment focuses on.

Experiment Parameters. This experiment is performed with a threshold of 1 for the solution finding, thus disabling dead feature detection and only focusing on the backtracking algorithm for inconsistent models. In order to enforce the feature models to be inconsistent, the number of expected versions per service has been set to 1 and a percentage of 100% is chosen for the ratio of mandatory services. Instead of setting a ratio for the excludes constraints, a fixed number is set for them. The experiment is run in two variants, one time with one excludes constraint and the other time with two. Since all services are set to be mandatory, the chance to get an inconsistent model is quite high with setting one or two excludes constraints. If in case the feature model is consistent, no measurements are stored and the run is repeated. The feature models range from 2 to 20 services in steps of 2 services.

1 Excludes Constraint Variant:
- Numbers of services: 2 - 20
- Expected versions per service: 1
- Number of excludes constraints: 1
- Ratio of mandatory services: 100%
- Threshold for solution finding: 1

2 Excludes Constraint Variant:
- Numbers of services: 2 - 20
- Expected versions per service: 1
- Number of excludes constraints: 2
- Ratio of mandatory services: 100%
- Threshold for solution finding: 1

Results. As shown in Figure 6.10, backtracking requires a considerable amount of the correctness checking duration for inconsistent feature models, thus translation and consistency checking are not visualized here in the box plot. Still, the corresponding times are implicitly readable by comparing the backtracking time with the overall time. Even though the duration tendencies are similar to the ones of the solution finding process, these particular backtracking computations

perform still better by over a factor of ten. However, the duration line tends to have a slight exponential tendency.

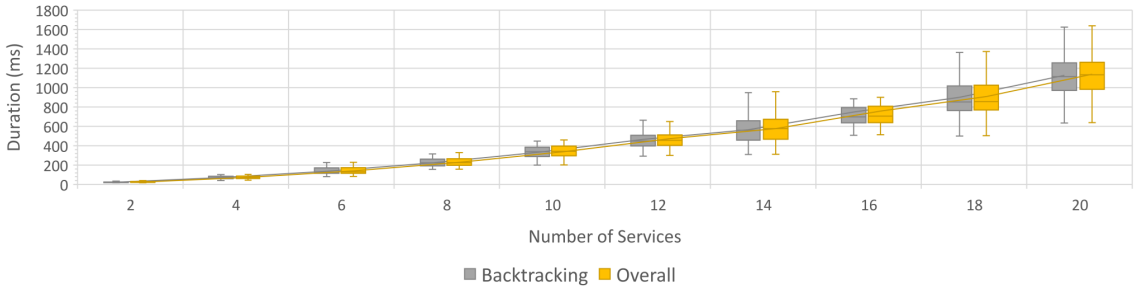


Figure 6.10: Time Measurements of Backtracking Inconsistent Models

The exponential tendency actually increases with every inconsistency causing element. This is shown in Figure 6.11 where the overall correctness checking times of feature models with one excludes constraints is compared to the ones with two excludes constraints. In Chapter 5 we have shown the backtracking algorithm which requires multiple consistency checking iterations, depending on how many potentially inconsistency causing element exist in the feature model. Thus analyzing inconsistent feature models scales really bad for defective large scale models.

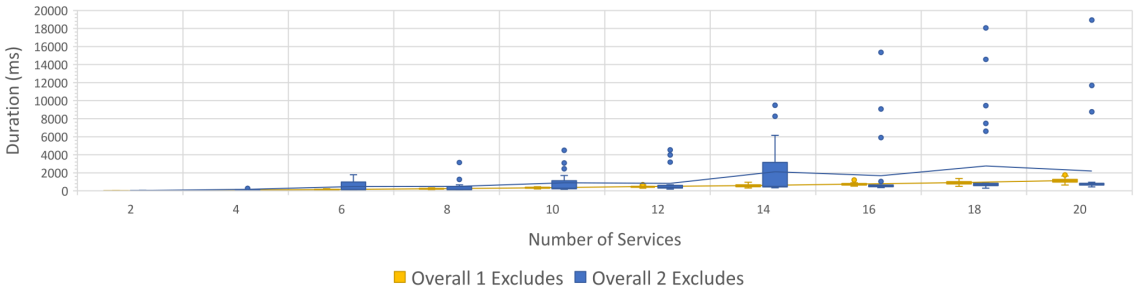


Figure 6.11: Comparing Time Measurements for an Increasing Number of Excludes Constraints

6.3.6 Findings of Evaluating Correctness Checking

From the correctness checking process, the translation of feature models to a propositional formula is the least time consuming computation. Checking the consistency of a feature model also requires relatively little time and scales well even for microservices-based feature models comprising hundreds of services. In contrast, the procedure of finding all solutions of a consistent feature model in order to enable dead feature detection needs most of the time of the correctness checking method. Computing all possible solutions for a given feature model proved to be costly for Logic Solver, the SAT solver used in HEIMDALL's SAT service. Similarly, in case the feature model is inconsistent, the backtracking algorithm is mainly responsible for the duration of the correctness checking process. The difference is that the solution finding process is mostly influenced by the number of elements in the feature model whereas the backtracking algorithm is mainly impacted by possibly inconsistency causing elements.

6.4 Evaluation of Feature Configuration Validation

In this section, we perform a set of different experiments that measure the performance for validating the three different categories of feature configurations, namely legal, incomplete, and illegal configurations.

6.4.1 General Setup for Configuration Validation

The general setup for configuration validation is very similar to the one for correctness checking. Again, each experiment usually runs 25 times for each single service size. For each single run a random feature model gets generated. This time it is absolutely necessary that all feature models are consistent since trying to validate a feature configuration against an inconsistent feature model would always fail. In each run, correctness checking is performed before the actual validation of the feature configuration, since we use the returned results to generate a feature configuration that fits the nature of the corresponding experiment. Regarding the correctness checking, the threshold for finding solutions for the different experiments, is set to only 1000 in order to avoid spending unnecessary time with the solution finding process (which is not of interested in the current experiments anyhow). As soon as a feature configuration has been generated, feature configuration validation is performed, time measurements and complementary statistics of the validation process are stored into a CSV file of the current experiment, and again in the end the feature model gets deleted. The results are again merely visualized as box plots in order to give a notion about the variance of the results for each particular service size. The following time measurements are performed during the validation of a feature configuration:

- **Translation:**
  The time it takes to perform the translation of feature models to propositional formula and to set the corresponding variables of the formula to true (according to the given feature configuration).
- **Legal:**
  The duration it takes to validate whether a feature configuration is legal. Since a legal configuration per definition has to completely match with one of the possible solutions of a feature model, Logic Solver just returns the one solution as result which is identical to the feature configuration.
- **Incomplete:**
  In case the feature configuration is incomplete, this is the time it takes that is required to compute the specific solutions of a feature model that conform with the truth values dictated by the feature configuration.
- **Illegal:**
  In case the feature configuration is neither legal nor incomplete, no solution is returned by Logic Solver. Therefore backtracking is required again, but this time contradictions do not need to be found in the feature model, but in the feature configuration. Therefore this is the time that is consumed by the backtracking algorithm in order to find contradictions in the given feature configuration.
- **Overall:**
  The time that is consumed by the whole feature configuration validation method. Due to the implementation of feature configuration validation as shown in Section 5.8.3, this means that the overall time measures one of the following combinations for the different types of feature configurations:

- **Legal Configurations:** The sum of *translation* and *legal*.
- **Incomplete Configurations:** The sum of *translation* and *incomplete*.
- **Illegal Configurations:** The sum of *translation*, *legal*, and *illegal*.

6.4.2 Evaluation of Legal Configurations

This experiment discloses the performance of our SAT service when it comes to validating legal configurations against a feature model.

Experiment Parameters. The feature models that form the basis of the experiment range from 10 to 100 services, increasing in steps of 10 services. Expected versions per service is set to 2 and excludes constraints and mandatory services are neglected. As already disclosed in the general setup section, for each experiment run, a feature configuration is validated against the underlying feature model. We generate the feature configuration by using the last solution returned by correctness checking and use the ids of the corresponding elements as feature configuration to be validated.

- Numbers of services: 10 - 100
- Expected versions per service: 2
- Ratio of excludes constraints: 0
- Ratio of mandatory services: 0
- Threshold for solution finding: 1000

Results. Figure 6.12 shows that the whole feature configuration validation process performs quite well even for feature models with 100 services. The translation of the feature model and the feature configuration to propositional formula is a bit faster than the actual validation of the legal configuration. The overall duration does not steadily increase with an increasing number of services per feature model.

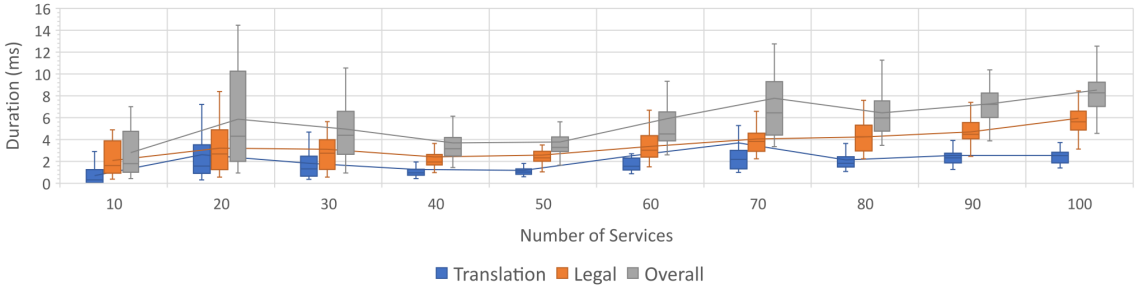


Figure 6.12: Performance of Validating Legal Configurations

Figure 6.13 shows that the performance is rather correlated to the number of elements that are contained in a feature configuration. This makes sense since only the parts of the feature models are translated and part of the feature configuration that are actually selected by the feature configuration.

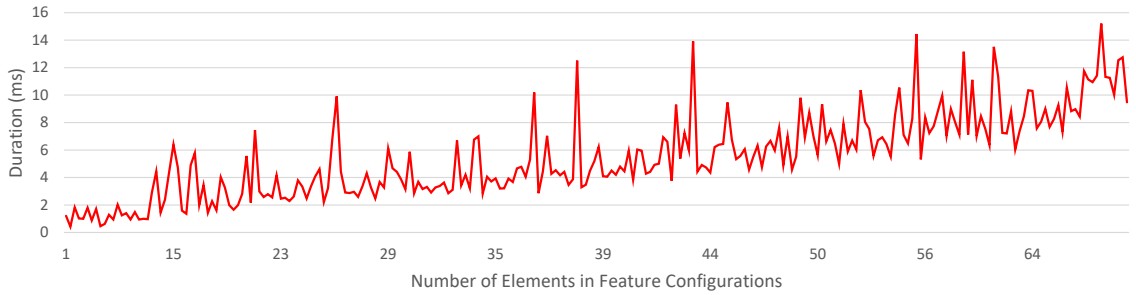


Figure 6.13: Performance of Validating Legal Configurations

6.4.3 Evaluation of Incomplete Configurations

The next experiment analyzes the performance of the SAT service's validation process for incomplete configurations.

Experiment Parameters. The parameters for the current experiment are similar to the previous experiment in Section 6.4.2. For a change, however, this time only $2/3$ of the element ids of a valid solution returned by the consistency checking is used as basis for the feature configuration. In order to increase the chance that the configuration is incomplete, 10% of the services are changed to be mandatory.

- Numbers of services: 10 - 100
- Expected versions per service: 2
- Ratio of excludes constraints: 0
- Ratio of mandatory services: 10%
- Threshold for solution finding: 1000

Results. Figure 6.14 shows that the validation of incomplete configurations performs not as good as the validation for legal configurations, but still models with 100 services can be computed in an acceptable time in not more than 400ms in average. The translation of the feature model and the feature configuration to propositional formula is as always extremely quick, thus the actual validation has the major influence on the performance. A further observation is that the overall duration does not entirely increase in a steady manner with the increasing number of services per feature model.

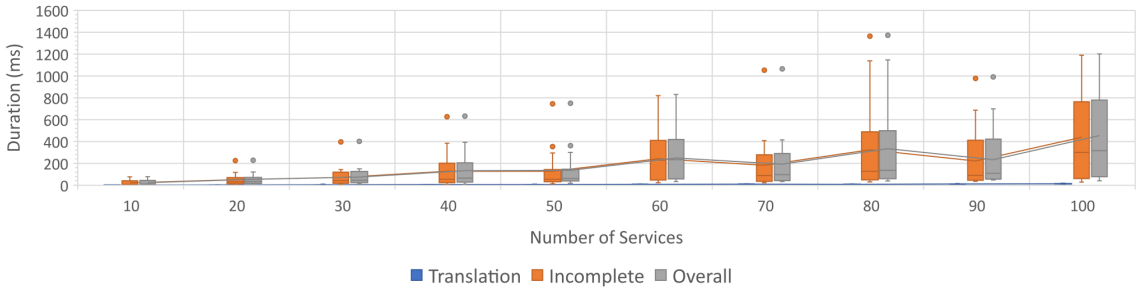


Figure 6.14: Performance of Validating Incomplete Configurations

In Figure 6.15 we see that the performance is actually correlated to the number of result sets that the SAT solver returns. This makes perfect sense, since the more possibilities exist to extend the currently validated feature configuration, the more solutions are computed by Logic Solver to give suggestions on how to continue the configuration. With respect to the computation of solutions, we have already observed in previous sections that the Logic Solver's solving function is costly.

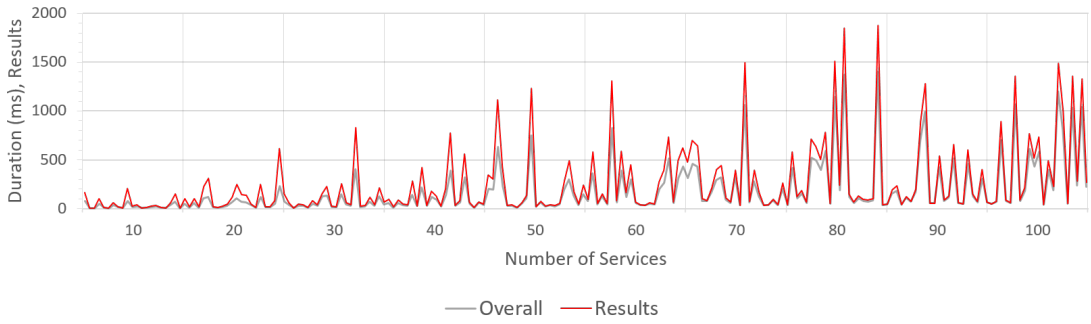


Figure 6.15: Correlation of Overall Duration and Number of Results Returned by the Validation

6.4.4 Evaluation of Illegal Configurations

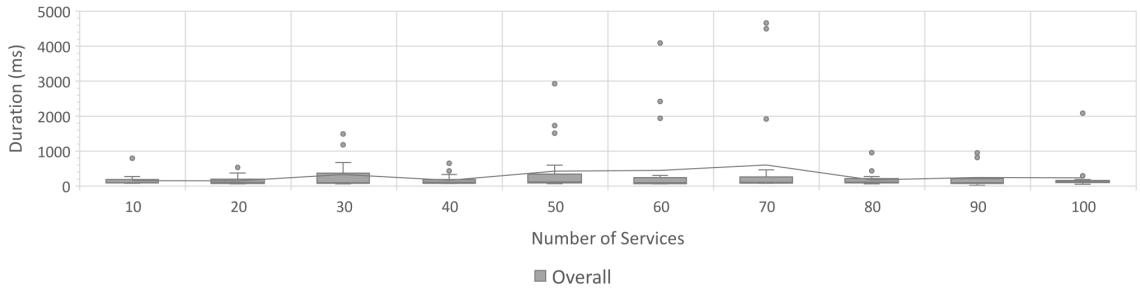
Finally we analyze the performance of validating illegal feature configurations, whereas backtracking is used to detect the configuration elements that are causing the configuration to be illegal.

Experiment Parameters. Again, the parameters resemble to the ones in the previous experiments of configuration validation, but this time the number of excludes constraints per feature model is set to 1. The two microservice versions that exclude each other are then both selected for the feature configuration in order to ensure that an illegal configuration is sent to the SAT service.

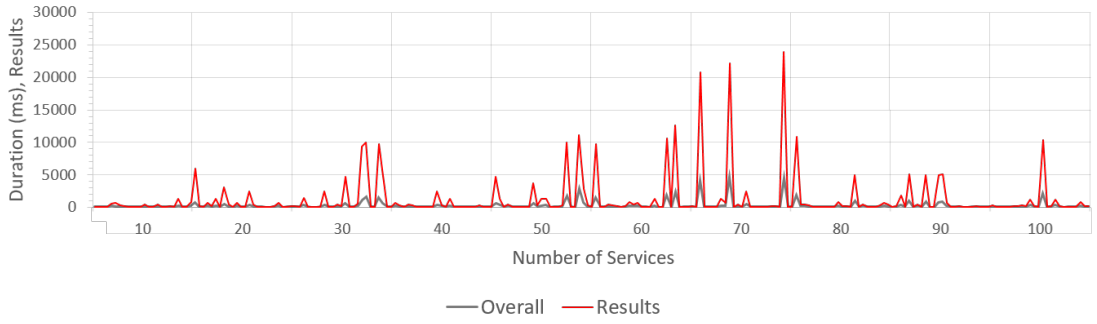
- Numbers of services: 10 - 100
- Expected versions per service: 2
- Number of excludes constraints: 1
- Ratio of mandatory services: 0
- Threshold for solution finding: 1000

Results. Figure 6.16(a) represents the overall duration of the validation procedure for illegal configurations. Translation time is completely omitted since it is barely contributing to the overall duration. Even though higher numbers of services in feature model can worsen the performance of the validation process, it is again the number of computed result sets that correlate with the time measurements, as shown in Figure 6.16(b). Similarly to the backtracking algorithm for inconsistent feature models, resolving illegal configurations can lead to a really high number of possibilities on how a feature configuration can be made legal. Thus it is rather a question of how complex the dependencies between the versions of services are, than rather the pure number of services and versions in a feature model. Figure 6.16(c) shows that the time measurements values look way more stable if runaway values are filtered out. Thus the performance of validating

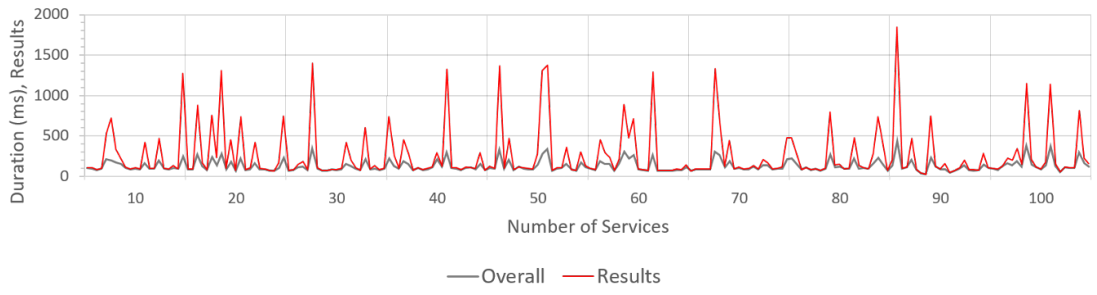
illegal configurations shows promising results for highlighting contradictions in feature configurations and usually does not need more than 400ms. Only really complex microservices-based feature models, which are potentially caused by the random nature of the generation procedure for the feature models, tend to challenge the validation of illegal feature configurations.



(a)



(b)



(c)

Figure 6.16: Performance of Validating Illegal Configurations

6.4.5 Findings of Configuration Validation

With respect to validating feature configurations, the validation of legal configurations performs really well, also for large microservices-based feature models. This is because a legal configuration is per definition one specific possible solution of the feature model and therefore no further

solutions need to be computed with the underlying SAT solver. The validation of incomplete configurations also performs in a decent manner, but it is highly depending on the complexity of the feature model, which features are selected and thus, how many possible set of features can be suggested to continue the current feature configuration. On the other hand, the validation of illegal configurations can be again costly, depending on the complexity of the constraints between the service versions. However, the last experiment has shown that filtering out a few runaway values reveals a quite stable picture of the performance of validating illegal configurations and we assume that the randomly generated feature models tend to have way more complex dependencies than real world microservices-based architectures have.

7

Closing Remarks

7.1 Conclusion

This thesis focused on how software product line concepts can be mapped to the microservices domain in order to enable automated analysis of microservices-based feature models. Chapter 2 introduces important concepts of microservices and software product line practices, such as fundamentals of feature models, the translation of feature models to propositional formulas, and how SAT solvers are utilized to perform automated analysis on feature models. In Chapter 3 some recent research in the microservices domain has been introduced and previous work which have tackled challenges related to the task of modelling and managing services dependencies have been presented. Furthermore, some research that has also adopted SPL practices to overcome challenges in current trends of the software engineering landscape have been highlighted and important research in automated analysis of feature models has been introduced. The collected background information serves as a basis to establish a mapping between SPL concepts and the domain of microservice applications, as stated in the first research question:

> *RQ1: How can software product line concepts be mapped to the domain of microservice applications to enable automated analysis of microservices-based feature models?*

In Chapter 4 the mapping of the SPL concepts to the microservices domain has been performed carefully and with respect to common properties of the respective artifacts in both domains. The mapping has been used to derive microservices-based feature models and the existing practice of translating feature models to propositional formula has been adapted to define a formal model of microservices-based feature models. The concepts for automated analysis of feature models have then been adopted to enable correctness checking of microservices-based feature models and to automatically validate the dependencies in service configurations. Eventually, the formal model and the automated analysis concepts have been used as a basis to implement the HEIMDALL application (described in Chapter 5) in order to answer the second research question:

> *RQ2: How can we build tooling that is capable of both validating given service configurations and recommending fixes for invalid service configurations based on satisfiability-solution techniques?*

The prototype system allows software and DevOps engineers to create feature models for microservices-based applications either with the help of a graphical editor or to import feature models that are defined in a YAML-based domain specific language that fully implements their characteristics. HEIMDALL also allows performing automated correctness checking on these models and enables configuration support to interactively describe valid service configurations. Both

of these automated analysis concepts have been realized with the help of an existing satisfiability solver called Logic Solver, which allowed to adopt the translation rules of feature models to propositional formula in a very straightforward manner.

Chapter 6 has then be used to validate the implementation in a quantitative performance evaluation. A random feature model generator has been implemented as part of the HEIMDALL application to facilitate the generation of the feature models that formed the basis for the experiments of the evaluation. The experiments have shown that the HEIMDALL application, albeit being a prototype, performs quite well for most of the automated analysis approaches, even for microservice applications with hundreds of different services.

7.2 Threats to Validity

The following section discusses threats to validity concerning both, the results of the performance evaluation as well as the mapping of SPL techniques to the domain of microservices-based applications.

7.2.1 External Validity

Logic Solver has been used as SAT solver for the HEIMDALL application. Even though it is based on MiniSAT, an industrial-strength SAT solver, it has its limitations. Evaluations have shown that an extensive use of its solving method is decreasing the performance of correctness checking and configuration validation and can even reach a hard-coded memory limit of the Logic Solver library. Furthermore it has not built-in dead feature detection or backtracking capabilities, therefore such mechanics needed to be implemented manually for the HEIMDALL application. These implementations might have its drawbacks and computationally faster algorithms may exist.

7.2.2 Internal Validity

The performance evaluation has been conducted on a laptop. Even though the machine is relatively powerful, it is possible that its performance influenced the results of certain measurements. However, a multitude of measurement runs have been performed for each experiment in order to compensate possible performance fluctuations. Another possible threat are the randomly generated feature models which might not reflect well enough the structure of real world microservices-based application with respect to services, versions and their dependencies. However, we believe that the randomly generated models have more complex dependencies than real world applications do. Therefore we hope that the latter would actually perform better than most of the already promising evaluations of artificial models have shown.

7.2.3 Construct Validity

The mapping of the SPL practices to the microservices domain, especially the mapping of feature models to microservices-based applications, have been performed carefully and with respect to common properties of the artifacts in both domains. However, it is not said that this is the only possible mapping and their might exist better approaches or improved alterations of the current approach.

7.3 Future Work

In the following section possible additions and improvements to the model, the developed prototype and the evaluations are discussed.

Extending the HEIMDALL Application. Currently, the validation of feature configurations need to be actively triggered in HEIMDALL. However, interactive configuration support generally includes that a feature configuration is automatically validated after every decision the user performs. An implementation of such a mechanism yields advantages that are twofold. On the one hand, the user would immediately notice bad decisions during the configuration process. On the other hand, the implementation could be done in such a manner that such choices would get rejected instantly and a configuration couldn't become inconsistent in the first place. Expensive backtracking operations could be avoided and large scale microservices-based feature models could definitely benefit from that.

Combining HEIMDALL with BIFROST. The actual idea of the topic for this thesis has been triggered by previous work done by Schermann et al. [SSLG16] who have introduced a formal model for supporting continuous deployment with automated enactment of multi-phase live testing strategies. A prototype called BIFROST has been implemented to provide tooling that allows developers to define and automatically enact such complex live testing strategies. HEIMDALL and BIFROST would complement each other well, since HEIMDALL could be employed to ensure that only valid service configurations would be used for the live testing strategies in BIFROST. Therefore combining both systems by ensuring that they can seamlessly interact with each other would definitely have beneficial synergy effects.

Extended Feature Models for the Formal Model. The formal model for microservices-based feature models is based on feature models, but this approach is not proven to be the best possible one. One alternative could be to use the possibilities of extended feature models which have been briefly introduced in Section 2.3.3. For example versions of services could be modelled as attributes of services and extra-functional features could used to control version specifications and the constraints between the microservices.

Quantitative Evaluations. The validity of the conducted performance evaluation is questionable since randomly generated microservices-based feature models have been used. An evaluation that comprises feature models of real world large scale applications would be desirable to clearly verify whether the performance of the automated analysis methods match the needs of software and DevOps engineers.

Qualitative Evaluations. Furthermore a qualitative evaluation of the HEIMDALL application would help to target which features could be improved and extended and what capabilities are probably missing in order to use it productively in the recurring tasks of software and DevOps engineers when deploying microservices-based applications.

Appendix A

Attachments

A.1 Heimdall Installation Guide

Heimdall

Heimdall allows to model microservices-based applications as feature models and is capable of both validating given service configurations and recommending fixes for invalid service configurations based on satisfiability-solution techniques.

Requirements

- Docker: 17.06.0-ce (or higher)
- Docker-Compose: 1.14.0 (or higher)

Setup

1. After cloning or copying the heimdall project to your desired destination, adjust the .env file in te project folder so that the host parameters of the services equal to the ip where your docker-engine runs, e.g. on windows machines with installed docker toolbox this is usually 192.168.99.100:
   SPL_HOST=192.168.99.100
   SPL_DB_HOST=192.168.99.100
   SAT_HOST=192.168.99.100

2. Then run docker-compose up inside the project folder
   docker-compose up

A.2 Running Example as Feature Model DSL

```yaml
name: Microservices-Based Applications
description: Feature model describing a sample set of microservices based applications
microservices:
- name: Frontend
  description: Sample frontend service
  mandatory: true
  versions:
  - semver: 1.2.7
    constraints:
    - constraint_type: requires
      targets:
      - microservice: Backend
        version: 1.5.1
  - semver: 1.3.0
    constraints:
    - constraint_type: alternative
      targets:
      - microservice: Backend
        version: 2.0.3
      - microservice: New-Backend
        version: 1.0.4
  - semver: 2.0.0
    constraints:
    - constraint_type: requires
      targets:
      - microservice: New-Backend
        version: 1.0.4
- name: Backend
  description: Sample backend service
  mandatory: false
  versions:
  - semver: 1.5.1
    constraints:
    - constraint_type: excludes
      targets:
      - microservice: New-Backend
        version: 1.0.4
  - semver: 2.0.3
    constraints:
    - constraint_type: excludes
      targets:
      - microservice: New-Backend
        version: 1.0.4
- name: New-Backend
  description: Sample service of a completely rewritten backend service
  mandatory: false
  versions:
  - semver: 1.0.4
    constraints:
    - constraint_type: excludes
      targets:
      - microservice: Backend
        version: 1.5.1
    - constraint_type: excludes
      targets:
      - microservice: Backend
        version: 2.0.3
```

Listing A.1: Sample Feature Model Defined in the YAML Based DSL

A.3 CD Contents

- **master_thesis.pdf**
  Master Thesis as PDF.
- **abstract.txt**
  Abstract in English.
- **zusfsg.txt**
  Abstract in German.
- **heimdall/**
  The source code of the entire HEIMDALL application
- **heimdall/msc-thesis-spl/src/demo/evaluation/**
  The scripts that were required to run the experiments for the Evaluation.
- **heimdall/msc-thesis-spl/src/demo/evaluation/results**
  The results of the evaluation as csv files.

Bibliography

[AAE16] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 44–51. IEEE, 2016.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[BBRC06] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006.

[BM11] Jaime Barreiros and Ana Moreira. Soft constraints in feature models. In *International Conference in Software Engineering Advances*, 2011.

[BMAC05] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated reasoning on feature models. In *CAiSE*, volume 5, pages 491–503. Springer, 2005.

[BSP09] Goetz Botterweck, Denny Schneeweiss, and Andreas Pleuss. Interactive techniques to support the configuration of complex feature models. 2009.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BWZ15] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W Eisenecker. Generative programming for embedded software: An industrial experience report. In *GPCE*, volume 2, pages 156–172. Springer, 2002.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34. IEEE, 2007.

[DDLM17] Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, and Manuel Mazzara. Microservices: Migration of a mission critical system. *arXiv preprint arXiv:1704.04173*, 2017.

[DFML17] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 21–30. IEEE, 2017.

[DGL+16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[EK02] Christian Ensel and Alexander Keller. An approach for managing service dependencies with xml and the resource description framework. *Journal of Network and Systems Management*, 10(2):147–170, 2002.

[Eva17] Clark C. Evans. The official yaml web site. http://yaml.org, 2017. Accessed on August 7, 2017.

[FL14] Martin Fowler and James Lewis. Microservices. *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on June 28, 2017]*, 2014.

[GKSS08] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[GT17] Jean-Philippe Gouigoux and Dalila Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 62–65. IEEE, 2017.

[Hem08] Adithya Hemakumar. Finding contradictions in feature models. In *SPLC (2)*, pages 183–190, 2008.

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[Inc17] Docker Inc. What is docker? https://www.docker.com/what-docker, 2017. Accessed on August 7, 2017.

[KMK16] Gabor Kecskemeti, Attila Csaba Marosi, and Attila Kertesz. The entice approach to decompose monolithic services into microservices. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 591–596. IEEE, 2016.

[Kra14] Lucas Krause. *Microservices: Patterns and Applications*. Luana Krause, 2014.

[LKL02] Kwanwoo Lee, Kyo Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. *Software Reuse: Methods, Techniques, and Tools*, pages 62–77, 2002.

[LNS+15] Vinh D Le, Melanie M Neff, Royal V Stewart, Richard Kelley, Eric Fritzinger, Sergiu M Dascalu, and Frederick C Harris. Microservice-based architecture for the nrdc. In *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*, pages 1659–1664. IEEE, 2015.

[LTV16] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*, 2016.

[Man02] Mike Mannion. Using first-order logic for product line model validation. *Software Product Lines*, pages 149–202, 2002.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009.

[MC03] Mike Mannion and Javier Camara. Theorem proving for product line model verification. In *International Workshop on Software Product-Family Engineering*, pages 211–224. Springer, 2003.

[McG04] James McGovern. *A Practical Guide to Enterprise Architecture*. Prentice Hall Professional, 2004.

[MCL17] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *In Proceedings of the 24th IEEE International Conference on Web Services (ICWS) - Applications Track*, 2017.

[Men09] Marcílio Mendonça. Efficient reasoning techniques for large scale feature models. 2009.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009.

[New15a] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. " O'Reilly Media, Inc.", 2015.

[New15b] Sam Newman. Principles of microservices, 2015. 20 Years of Java Just the Beginning.

[NMMA16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[Nor02] Linda M Northrop. Sei's software product line tenets. *IEEE software*, 19(4):32–40, 2002.

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

[PW17] Tom Preston-Werner. Semantic versioning 2.0.0 | semantic versioning. http://semver.org/, 2017. Accessed on July 24, 2017.

[RGMS14] LF Rincón, Gloria Lucia Giraldo, Raúl Mazo, and Camille Salinesi. An ontological rule-based approach for analyzing dead and false optional features in feature models. *Electronic notes in theoretical computer science*, 302:111–132, 2014.

[SRD16a] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Automated setup of multi-cloud environments for microservices applications. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pages 327–334. IEEE, 2016.

[SRD16b] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending feature models with relative cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 79–88. ACM, 2016.

[SRD16c] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Software product lines for multi-cloud microservices-based applications. In *6th International Workshop on Cloud Data and Platforms (CloudDP)*, 2016.

[SRD17] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending dynamic software product lines with temporal constraints. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 129–139. IEEE Press, 2017.

[SSLG16] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C Gall. Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Middleware*, page 12, 2016.

[TBC06] Pablo Trinidad, David Benavides, and Antonio Ruiz Cortés. Isolated features detection in feature models. In *CAiSE Forum*, 2006.

[TBD+08] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[TM17] Rebecca Turner and Kat Marchán. npm. https://www.npmjs.com/, 2017. Accessed on July 24, 2017.

[UT14] Johan Uhle and Peter Tröger. *On Dependability Modeling in a Deployed Microservice Architecture*. PhD thesis, Master's thesis, University of Potsdam, Germany, 2014.

[Wol17] Eberhard Wolff. *A Practical Guide to Continuous Delivery*. Addison-Wesley Professional, 2017.

[WXH+10] Bo Wang, Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Wei Zhang, and Hong Mei. A dynamic-priority based approach to fixing inconsistent feature models. *Model Driven Engineering Languages and Systems*, pages 181–195, 2010.

[YZZM09] Hua Yan, Wei Zhang, Haiyan Zhao, and Hong Mei. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. *Formal Foundations of Reuse and Domain Engineering*, pages 65–75, 2009.

[ZBCS16] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.