DLINE JOURNALS

Journal of Digital Information Management

JDIM

ISSN 0972-7272
Published online: 27 July 2024

A Brief Review of the Automation of Dependency Satisfaction With in Microservices Architectures

Ammar Esrawi, Bassel AlKhatib
Syrian Virtual University
Syrian Arab Republic
esrawi.ammar@gmail.com
t_balkhatib@svuonline.org

ABSTRACT: *Accelerated advances in network speed, reliability, and security have increased the demand for software and services to move from being stored and processed locally on users' devices to being managed by third parties that can be accessed through the network. This has led to the need to develop new software architectures and software architectures that meet these new requirements. One example of software architecture is the recent emergence of a microservices architecture to meet online service providers' maintenance and scalability requirements. Microservices is an architecture style that focuses on breaking down a system into small, lightweight services that are intentionally created to perform a highly cohesive business function, and it is an evolution of the traditional service-oriented architecture style. The practical projection of microservices architecture in information engineering is several small applications that work together to provide an integrated service that appears to the external medium as a single application. Each of these applications has its own independent source code and deals with data sources that are different from the rest. As a result, it has an independent development team. Partial applications depend on each other to accomplish their functions, and therefore, any modification to one of the services will affect the work of the rest of the services; in large applications that require keeping the system always available, we must perform the process of publishing updates for parts of the system without stopping it completely, so we must monitor the impact of modifications and forecast system status before new updates are deployed. In achieving this prediction, we must know the dependencies between different services to arrive at a cognitive model representing the dependencies between services or applications.*

Subject Categories and Descriptors: [B.7 INTEGRATED CIRCUITS]: Microprocessors and microcomputers; [C.5.3 Microcomputers IMPLEMENTATION] [D.2.11 Software Architectures]; [C. Computer Systems Organization]: Hardware/software interfaces

General Terms: Software Analysis, Micorservices Architecture, SOA, Software Complexity

Received: 13 May 2024, Revised 3 July 2024, Accepted 15 July 2024

Keywords: Microservices Architecture, Monolithic Architecture, Software Complexity, Dependancy Satisfaction

Review Metrics: Review Scale: 0/6, Review Score: 5.12, Inter-reviewer consistency: 87.2%

DOI: https://10.6025/jdim/2024/22/3/99-107

1. Introduction

Faster, more reliable, and more secure networks have increased interest in moving applications and services from local storage and processing on user devices to third-party management that is only reachable through the network. Such a shift builds up a need to develop new software construction methods and architectural patterns to meet new requirements. One instance in the domain of software architectural design involves the rise of microservice architecture in response to maintenance and scaling requirements established by online service providers [1].

Microservices architecture has become a leading architectural trend in service-oriented software development. Microservices refer to an architectural style that decomposes a system into small, lightweight services explicitly designed to perform strongly related, businessrelevant functions. They represent an evolution of traditional service-oriented architecture. It is also defined as "the smallest autonomous processes that interact via messaging," and microservices architecture as "a distributed application where all its modular units are microservices."

The microservices approach has served as an attractive architecture for leading software consulting and product design companies in the past ten years. It enables teams and software organizations to be more productive overall and to develop more successful software products frequently. Companies like Netflix and SoundCloud have adopted the microservices style in the cloud and reaped a host of benefits from it.

Every microservice is a composition of several subservices that this microservice exposes to other services within the same system or externally as a service. Similarly, every microservice depends on subservices from other or external services. These very same subservience and their relationships can be expressed in a standardized feature model or a customized model to allow more detailed study later on, through automated feature model analysis, the relationships between these services to establish the state of this system: be it sound, unsound, or improvable.

2. Literature Review

2.1. Preliminaries

The use of Software Product Line techniques in modelling and managing dependencies in microservices-based applications has still not been investigated. Hence, no related work has been directly conducted on it.

The first section describes general research recently conducted on microservices. The second section reports previous works that have faced the challenges of modeling and managing microservices' dependencies. In the third section, we outline some research that has adopted SPL practices to overcome challenges in the current software engineering trends. The last section describes some significant research related to the automatic analysis of feature models.

2.2. Current Research in Microservices

The definitions, principles, and fundamental practices of microservices [2], [3], and [4] are likely to remain valid for an extended period. However, research discussing the various challenges of microservices is steadily adapting to the increasing findings in the field. Two surveys have been recently published [5] and [6] that attempt to reflect the current research in microservices.

Dragoni in [5] outlines the challenges of microservices-based applications in the past, present, and future. Regarding the past, the researchers assert that microservices architecture is the direct successor to Service-Oriented Architecture (SOA), arguing that microservices architecture is essentially a form of SOA. For the current state, the key characteristics and advantages of microservices and their impact on quality attributes such as availability, reliability, maintainability, performance, security, and testability are summarized [5].

Looking at future challenges, the focus is on issues arising as microservices become distributed programs, making their development more complex. Two main areas, reliability and security, are highlighted. Dragoni emphasizes research addressing these issues to answer questions such as managing changes to a service without adversely affecting the consuming services and preventing attacks exploiting network communications.

Francesco et al. [6] applied a systematic mapping study methodology to identify and evaluate the current state of research in microservices. They use concepts like publication trends, research focus, and industrial adoption potential in their classification process. For example, the research contribution category lists the recurring distribution of findings in the investigated papers. Another category examines research strategies, with "solution proposal" being the clear winner, interpreted as microservices being in their infancy and not yet integrated into standards, leading many researchers to propose their solutions to recurrent or specific problems. Another exciting category deals with the most common issues targeted by current research, identifying key terms like "complexity," "low flexibility," "resource management," and "service composition."

Francesco and colleagues argue that these findings confirm that while microservices can help achieve a good level of flexibility, they may also increase complexity due to the large number of distributed services involved. The "service analyzer" tackles the problem of homogeneous system breakdowns within prominent microservices-based

applications. Consequently, numerous software applications in recent years have developed large, unsustainable architectures, prompting the software industry to address this issue.

Much work [1] [7] [8] has been presented in recent years to facilitate the decomposition of existing monolithic systems to enjoy the benefits of microservices architectural style. Al-Shqeerat et al. [1] focused on identifying challenges, architectural schemes and views, and quality attributes related to microservices systems. Kecskemeti et al. [29] introduced a methodology to decompose monolithic services into multiple microservices, applying this methodology to various results from a real project called ENTICE. Similarly, Levkovitz et al. [30] described a technique to identify and define good candidates for becoming microservices within a monolithic enterprise system.

Despite the hype and popularity of microservices, many software systems remain in their monolithic form. Recent research persistently aims to ease the transformation of these large software systems into microservices-based applications [5][9][10].

Dragoni [5] used a real-world case study to illustrate how rearchitecting a monolithic unit into microservices using specific techniques positively affects scalability. Similarly, Kucuk and Tmzalit [9] provided technical lessons from migrating monolithic systems to microservices, addressing details of microservice deployment and organization.

Mazlami [10] presented another very recent but particularly noteworthy piece of research. Mazlami in [10] introduced a formal model for microservices extraction to enable the computational recommendation of microservices candidates. Consequently, instead of solely relying on existing informal migration techniques, a foundation for automated support tools is provided, along with a web prototype to demonstrate appropriate performance assessments. Given the formal model, they are the first to offer a semi-automated approach covering the recommendation of microservices from a monolithic codebase without requiring significant user input.

2.3. Modeling and Managing Service Dependencies

The problems regarding microservices' deployment and operational management have been given substantial research attention in the literature. However, limited work has been done on modeling service dependencies to present automated reliability analysis before deployment. Here, we outline three carefully selected research papers that treat this issue differently.

Given that microservices architecture is the direct successor to service-oriented architecture (SOA), managing services and their dependencies will be crucial to the architectural style of these services. One significant work is presented by Insel [11], which describes an approach for managing service dependencies using various XML technologies, such as XML, XPath, and RDF. By leveraging these general-purpose technologies, it is possible to represent dependency graphs in a way that can be analyzed using existing XML parsers.

There has never been a standard way to describe dependency information; therefore, this is the first time that management systems could generally benefit from it. This solution is limited to dependency management, with no dedicated formal model and no possibility of automatic analysis, but querying and visualization services together with their dependencies.

In his dissertation, Juhl [12] introduces a method for modeling the reliability of microservice architectures using dependency graphs, which are transformed into fault trees. In these dependency graphs, nodes represent applications, and directed edges indicate dependencies. As part of the dissertation, various methods for generating dependency graphs from deployed microservice architectures were presented. The dependency graphs were defined to be directed, acyclic, and rooted to facilitate the transformation into a fault tree. The main idea is that fault trees are then used to model the failure deployment of building microservices to get an overview of which parts of this architecture are more critical than others.

The dissertation restricts itself to already deployed systems and primarily calculates the deployment probability of failure for services. Here, automated consistency checks and validation of dependencies remain outside this scope because the approach was not designed from a perspective to guarantee valid service configurations before deployment. Another limitation of our work is that we only considered the granularity level of application and service dependency graphs, not specific versions of microservices. Nevertheless, different approaches for creating dependency graphs have already been applied to the prototype system of our thesis. Since there is an extraordinary amount of research in this direction, we will not focus on it further in our work.

Sherman et al. [13] introduced a formal model for multistage live experimentation. They presented a prototype application allowing release engineers to define, automate, and enact complex live experimentation strategies on microservices-based applications. The concept relies on traffic routing mechanisms using lightweight proxy components. The prototype is non-intrusive, requiring no feature toggling or other code-level changes. However, one key requirement is the correct routing of requests between different service instances and versions. Thus, the approach relies on the coexistence of service versions.

As explained in the previous section, the formal model models different versions of the available microservices. Using a domain specific language, the release strategies are specified by configuring metrics and thresholds that will assist in determining if the release is successful and should be accepted or if it must roll back. This approach

to the paper greatly facilitates the automation of experimentation within microservices-based applications for versioning, collaboration, and strategy sharing across changes or teams.

Some of the objectives mentioned therein are related to the goals of the current dissertation: firstly, Sherman solves the issue of automated validation of service deployment success. Moreover, one of the goals of the current dissertation is the automatic validation of service dependencies to ensure a valid configuration of the service before its experimental deployment.

2.4. Automated Analysis of Feature Models

Numerous works propose using formal specifications for the automated analysis of feature models. Benavides et al. [14] provided a summary of developments concerning linking formal specifications with feature models.

Mannion et al. [15], [16] were identified as the first to connect formal specifications with feature models. In [17], they were depicted as the first to use SAT (satisfiability testing) for analyzing feature models. Benavides et al. [14] also presented a comprehensive list of the most innovative contributions related to the automated analysis of feature models. For example, Yan et al.

[18] proposed an optimization method to reduce the size of the logical representation of feature models by removing unrelated constraints. Benavides et al. [14] also include an extensive list of analysis operations on feature models, referencing numerous studies addressing one or more of these analysis operations. Regarding health checking as part of automated analysis, some works in the literature analyze possible approaches and improvements to this aspect. Trinidad et al. [19] analyzed the detection of dead features based on previous work by Benavides et al. [14]. They proposed a method for discovering relationships that lead to the emergence of dead features. Trinidad et al. [20] also introduced further automated analysis methods that help detect inconsistencies in feature models and so-called fully mandatory features. These latter features are not explicitly defined as mandatory but are implicitly constrained across the trees. Fully mandatory features were also termed false optional features by Rincon [21], who proposed an existential rule-based approach for analyzing dead and false optional features in feature models. This approach formalizes first-order logic rules to identify dead features and false optional features, which also enables the identification of relationships causing related issues.

Himakumar [22] focuses on statistically finding inconsistencies using model checking and an incremental consistency algorithm. Wang [23] presents a methodology based on dynamic prioritization to resolve inconsistencies in feature models. Mendonça [24] provides spatio-temporal interpretations that follow reasoning techniques in feature models and enhances the computational performance of these techniques.

Support for feature model configuration is the subject of many works, with various proposals presented to address and improve this process. Here, we briefly present a few selected papers. They were illustrating different aspects of this. Mendonça [25] introduces SPLOT, a web-based logic and configuration system for SPLs. Their system provides inference and interactive configuration services for researchers and practitioners in SPL. The interactive configuration support validates each configuration decision to enforce consistency, resulting in a backtrack-free configuration process that benefits users by ensuring they are never forced to reconsider previous decisions.

Botterweck [26] illustrates interactive techniques for configuring complex feature models. These techniques include visual interaction with a formal inference engine, visual representation of multiple interconnected hierarchies, configuration progress indicators, and filtering visible nodes.

Barreras et al. [27] propose using soft constraints and classifying potential semantics for such constraints. Soft constraints can enhance configuration support by highlighting the most common configuration options.

Although all these concepts can be beneficial for microservice based feature models, they were not adopted in this work because they are primarily used to perform various aspects of automated feature model analysis.

2.5. Determination of Legacy SPL Elements for Microservices Applications

The principles of microservice-based applications fit the domain of Software Product Lines. Proper correspondences between a specific architectural style and a software development model set a route for re-using SPL techniques, like feature modeling, within the realm of microservices. Modeling microservice applications and their dependencies as a feature model allows the translation of this feature model into a formal specification. This allows the use of SAT solvers for both the automated verification of the feature model's validity and the validation of configurations of microservices.

We briefly introduced the fundamental concepts of microservices based applications and gave a concise overview of some SPL principles as the fundamental assets in SPLs. Comparing the principles of both domains reveals that microservices align well with SPL concepts. Regarding the core assets in SPLs, we emphasized that all production lines share a specific structure [28].

The microservices architectural pattern fits perfectly as an application for such a structure, where they would be shared amongst all microservices—real core assets in a microservices based application.

Microservices are a specific software component and unit type that can be independently replaced and upgraded [29]. Components are a typical example of core assets.

This comparison is further supported by the idea of reusing shared microservices and core assets. By attempting to derive more suitably defined legacy software libraries, microservices-based applications are the product in SPLs, where microservices are used to develop microservices dependent applications and core assets are equally used to develop products. This linkage leads to the final establishment of a set of microservices-based applications, which are the general counterpart of SPLs.

It allows us to model the diversified service domain in live experimentation environments. In this way, we can only model microservices-based applications that are likely to appear in several sets of available microservices and multiple versions of microservices. Besides, several microservices-based ap plications can re-use applications and share services among them, which is a characteristic property of SPLs, where core assets will be re-used in production line products.

Table 1 illustrates the assignment of legacy software libraries from SPLs and microservices.

SPLs	Microservices
Software production line	Microservices-based application set
Product	Microservices-dependent applications
Core assets	Microservice

Table 1. Illustrates the assignment of legacy software libraries from SPLs and microservices

3. Methodology

3.1. Microservices applications as feature models

We can differentiate practices used to model the structural relationships of microservices-based applications, such as applications, services, and service versions, from practices that model the constraints among services in these applications, for example, depending on another service.

3.1.1. Features and Structural Relationships

Regarding features and the structural relationship between features in feature models for microservices-based applications, we have identified the following core components:

Microservices-based Applications: The root node clearly illustrates the domain concept for microservices applications as configurable for the current production line in the feature model representing microservices-dependent applications. Feature configurations enabled by these models represent each specifically grouped set of microservices with specific constraints between services.

Ultimately, the feature model and the corresponding service configuration are used together for validation with the help of an SAT analyzer to obtain clear confirmation about the validity of the configuration. Fundamentally, valid feature configurations constitute products of our microservices application production line.

Microservices: Direct sub-features of the root feature in the feature model serve as a collection relationship and visualization of available microservices for the current model. Typically, services are defined as optional features, allowing them to select random subsets of available microservices for a given production line. Ultimately, this provides more flexibility when developing microservice-based products. However, in the absolute need to include a particular service in every product of a specific production line, it can instead be designed as a mandatory subfeature of the root feature.

Versions: Versions are designed as alternative sub-features to services since the goal is ultimately to obtain feature configurations that explicitly define the distinct service versions for a specific microservices-based application. From a practical standpoint, we do not need to enable feature configurations representing runtime environments where multiple versions of services may exist in parallel.

Function Versions: Direct sub-features of the microservice feature in the feature model serve as a collection relationship and visualization of function versions used within that service. Based on the specified characteristics, we have derived a formal representation of feature relationships between parents and children for feature models based on microservices. Initially, the feature model for microservices applications consists of a limited set S of n services.

$$S = \{s_1, s_2, s_3, \ldots, s_n\}$$

The service s_x itself is available in different versions. Thus, each service s_x leads to a limited set V of n versions vx_1, vx_2,... v_{xn} for each service sx belonging to S:

$$V(s_x)=\{v_1, v_2, v_3,..., v_n\}$$

Each service is a set of function versions, so each service version leads to a specific configuration of function versions f_1v_1, ..., f_1v_m, ..., f_nv_1,..., f_nv_m for each service version vx belonging to V:

$$F(V_x) = \{f_1v_1 \wedge \ldots \wedge f_1v_m \wedge \ldots \wedge f_nv_1 \wedge \ldots \wedge f_nv_m\}$$

Through the concept of microservices-based applications, microservices, function versions, versions, and the relationships between these features, we already have the basic components that allow us to model feature models for microservices applications.

The structure of these models yields a certain correspondence and some distinctive characteristics. These feature models for microservices-based applications exhibit four levels in the hierarchical sequence of features, and each path from the root to any leaf has the same depth. Each level represents an abstract idea.

The top level illustrates the domain concept for modelling a set of microservices applications, the second level lists available services and service functions in the level below, and the lowest level illustrates the available versions for each function individually.

Of the available relationship types between parents and children in the domain of feature models, the "group-or" relationship alone is not relevant and therefore not used for modeling microservices applications; practical microservices or function version applications do not have practical dependencies, "at least one" being acceptable for parent-child relationships.

We also use "Semantic Versioning" (SemVer) to manage software releases. Following this pattern, software version numbers and numbering schemes provide meaning that indicates the source code and the extent of changes occurring from one version to the next.

3.2. Constraints on relationships between features within services

Regarding constraints across trees between features in feature models for microservices-based applications, we have identified three constraints required for modeling sufficient dependencies between microservices.

Required: We have already introduced the required constraint, which clearly indicates that a specific feature requires another specific feature. Such constraints are typically found in microservices-dependent applications; thus, it is logical to reuse them in this current context to represent services that depend on the existence of another service.

Excluded: Like the required constraint, the excluded constraint has already been introduced. These constraints seem to be used for modeling features that cannot coexist in the same product. Although this constraint is less common than the required constraint, it can help highlight conflicting microservices, such as microservices obtaining the same network ports.

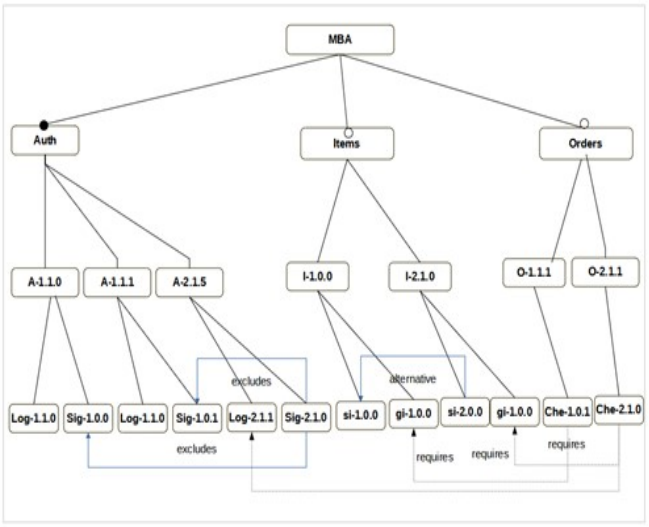
Alternative: Although tree constraints allow for highly complex relationships between any features in the model, we only need one additional potential constraint that has meaning in the domain of microservices applications. The alternative relationship between parents and children, as introduced, has been adopted as a constraints tree to clarify that the feature relies specifically on one feature from a particular set of possible different options. This is useful for representing microservices with specific dependencies that can be satisfied by multiple services, but only one is needed to accomplish the function. Exception constraints can appear with alternative constraints for explicit specification of conflicting alternative microservices. Such structures are particularly required for an optional alternative microservice constraint. If such a service is not specified for the configuration, the exception constraints ensure that conflicting alternatives cannot be determined simultaneously for the current configuration.

Since function dependencies can change due to version upgrades, we have placed constraints between features representing versions instead of modeling them between functions. Of course, constraints between functions can be specified, as constraints between functions may not change much. However, ultimately, it will only add unnecessary complexity to the feature model and can be individually resolved in the corresponding applications of these feature models in case this feature is desired.

Based on the specified characteristics, we have derived a formal representation of tree constraints for feature models based on microservices. Constraints cx always arise from specific versions. Thus, each function version vx leads to a limited set C of n constraints.

$$C\,(f_n v_m\, x) = \{c_1, c_2, c_3, \ldots, c_n\}$$

Example of a Structural and Relational Model Represented



3.2.1. Automated analysis of microservices applications

We adapt the logical formulae discussed concerning the notions of feature models for microservices-based applications. Table 2.8 shows the logical formulas that need to be specified for the three types of features and the three constraints across trees in these feature models.

The root feature can be adopted without modification; hence, it is represented by a simple formula, 'a'. Table 2.8 illustrates the other components and constraints.

Logical Formula	Feature Model Context	Relationship Name
a	Basic concepts in the microservices modeling domain. It represents the root feature a.	Application
$s \rightarrow a$	s is a sub-feature of a.	Optional Service
$s \rightarrow a$ $a \rightarrow s$ $s \leftrightarrow a$	s is a sub-feature of a, and a is a parent feature of s. a is a mandatory feature in s.	Mandatory Service
$v_1 \rightarrow s$ $v_n \rightarrow s$ $s \rightarrow 1\text{-}of\text{-}n\ (v_1, \ldots, v_n)$	v_1 is a sub-feature of s. s is a sub-feature of v_n. It is the parent of all its specific versions.	Service version
$f_1v_1 \rightarrow s_1$ $f_nv_n \rightarrow s_1$ $s_1 \rightarrow 1\text{-}of\text{-}n\text{-}m\ (f_1v_1, \ldots, f_nv_m)$	$f_1\text{-}v_1$ is a service version of s_1. $f_n\text{-}v_n$ is a service version of s_1. s_1 is the parent of all services within it.	Feature Version
$f_1v_1 \rightarrow f_nv_m$	f_1v_1 requires f_nv_m	Requirement
$f_1v_1 \rightarrow \neg f_nv_m$ $f_nv_m \rightarrow \neg f_1v_1$ $\neg\ ((s_iv_k \wedge (s_iv_k \rightarrow f_1v_1)) \wedge (s_jv_l \wedge (s_jv_l \rightarrow f_nv_m)))$	f_1v_1 excludes f_nv_m. f_nv_m excludes f_1v_1.	Exclusion
$f_1v_1 \rightarrow (f_1v_1 \vee f_nv_m)$ $f_nv_m \rightarrow (f_1v_1 \vee f_nv_m)$ $(s_iv_k \wedge (s_iv_k \rightarrow f_1v_1)) \vee (s_jv_l \wedge (s_jv_l \rightarrow f_nv_m))$	f_1v_1 is an alternative to f_nv_m	Alternative

Table 2. Illustrates the other components and constraints

3.3. Health checking and configuration validation

The proper reuse of the translation rules required, combined with feature modeling for production line modeling in service-oriented applications, opens up the possibility of applying automated analysis to feature models within the field of microservices. In the case of health checking and configuration validation, approaches for feature models based on microservices will not differ much from the general approaches to feature modeling.

4. Conclusion

Partial applications depend on each other to perform their services, so any change in one service will change the functioning of the other services. In large-scale applications where high system availability needs to be maintained constantly, we want to deploy updates to system components so they are not completely stopped. Now, it will become necessary to monitor the effect of changes and predict the system's state before deploying new updates. A prerequisite for such prediction is understanding the dependencies between different services, converging to a cognitive model representing the dependencies between services or applications.

Every microservice consists of some partial services provided to the remaining system or as external services. In the same way, each microservice depends either on partial services from other microservices or external services. These partial services and their relationships can be modeled in a standardized feature or self-defined model. This will enable us later to examine the connection between these services through an automated analysis of the feature model and determine the system's state as healthy, unhealthy, or improvable.

Much research has already been performed in the operational deployment and management of microservices, but intensive modeling of dependencies between services rarely exists. Similarly, there was no previous treatment for describing dependency information in a standardized way; therefore, this is the first time management systems could profit from it in general. Because no formal model exists, this approach is limited only to managing dependencies without the possibility of automating analyses, and its main tasks are querying and visualizing services together with their dependencies.

We knew how principles from service-oriented applications would fit the SPL domain. Proper correspondence between a concrete architectural pattern and a software development model for the domain of microservices allows for reusing SPL techniques, such as feature modeling. As soon as the dependencies between microservice applications are modeled as a feature model, such a feature model can be translated into a suggested formula. Thus, this enables using SAT-solving tools to automatically verify

a feature model's correctness and validate specified microservice configurations.

The research aims to improve the production line by adding a new step—health testing, executed in microservice deployment operations—to ensure the system remains consistent after deployment. This will, therefore, improve productivity in development and operation teams by automating satisfaction tests that are performed manually.

Consequently, mapping microservice dependencies into an SPL feature model in the SPL domain allows us to reuse tools from SPL in the microservices domain, such as SPL analyzers. This helps to easily verify the model's correctness and the validity of all possible configurations for such services.

References

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

[2] Lee, Kwanwoo., Kang, Kyo., Lee, Jaejoon. (2002). Concepts and guidelines for feature modeling for product line software engineering. *In: Software Reuse: Methods, Techniques, and Tools* (pp. 62–77).

[3] Newman, Sam. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc.

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

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

[6] Microservices: Trends, Focus, and Potential for Industrial Adoption. (2017). *In: Software Architecture (ICSA), 2017 IEEE International Conference on* (pp. 21–30). IEEE.

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

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

[9] Gouigoux, Jean-Philippe., Tamzalit, Dalila. (2017). 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 (pp. 62–65)*. IEEE.

[10] Mazlami, Gerald., Cito, Johannes., Leitner, Philipp. (2017). Extraction of microservices from monolithic software architectures. *In Proceedings of the 24th IEEE International Conference on Web Services (ICWS)—Applications Track*.

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

[12] Uhle, Johan., Tröger, Peter. (2014). On Dependability Modeling in a Deployed Microservice Architecture (Master's thesis, University of Potsdam, Germany).

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

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

[15] Mannion, Mike. (2002). Using first-order logic for product line model validation. *In Software Product Lines* (pp. 149–202).

[16] Mannion, Mike., Camara, Javier. (2003). Theorem proving for product line model verification. In International Workshop on Software Product-Family Engineering (pp. 211–224). Springer.

[17] Batory, Don. (2005). Feature models, grammars, and propositional formulas. *In Software Product Lines* (pp. 7–20).

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

[19] Trinidad, Pablo., Benavides, David., Ruiz-Cortés, Antonio. (2006). Isolated features detection in feature models. In CAiSE Forum.

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

[21] Rincón, L. F., Giraldo, Gloria Lucia., Mazo, Raúl., Salinesi, Camille. (2014). An ontological rule-based approach for analyzing dead and false optional features in

feature models. *Electronic Notes in Theoretical Computer Science*, 302, 111–132.

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

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

[24] Mendonça, Marcilio. (2009). *Efficient reasoning techniques for large-scale feature models (Doctoral dissertation)*.

[25] Mendonça, Marcilio., Wasowski, Andrzej W., Czarnecki, Krzysztof. (2009). SAT-based analysis of feature models is easy. In: Proceedings of the 13th International Software Product Line Conference (pp. 231–240). Carnegie Mellon University.

[26] Botterweck, Goetz., Schneeweiss, Denny., Pleuss, Andreas. (2009). *Interactive techniques to support the configuration of complex feature models*.

[27] Barreiros, Jaime., Moreira, Ana. (2011). Soft constraints in feature models. *In International Conference on Software Engineering Advances*.

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

[29] Fowler, Martin., Lewis, James. (2014). Microservices. Thought Works. Retrieved from http://martinfowler.com/articles/microservices.html