Balancing Variability and Costs in Software Product Lines: An Experience Report in Safety-Critical Systems

Udo Knop
bluesolve GmbH
München Germany
udo.knop@bluesolve.tech

Peter Hofman
Siemens Healthineers
Forchheim Germany
peter.hofman@siemens-healthineers.com

Michael Mihatsch
bluesolve GmbH
München Germany
michael.mihatsch@bluesolve.tech

Martin Siegmund
bluesolve GmbH
München Germany
martin.siegmund@bluesolve.tech

ABSTRACT

This paper provides a detailed testing strategy for Software Product Lines (SPLs) that aims to balance the amount of variability offered with the cost of that variability. The strategy is a combination of feature modeling, combinatorial testing, and deployment-based testing, and it is designed to handle the unique challenges presented by testing SPLs, including the large number of possible feature interactions.

The paper includes an experience report in the syngo SPL by Siemens Healthineers that had approximately 900 optional features and was deployed at about 35.000 end-customer installations. Testing this variability using conventional approaches was an immense challenge due to the vast number of required test cases and test configurations. As a result, the actual variability offered to customers was restricted to four configurations, and the time-to-market of new features was limited to four releases per year.

The goal of the project was to develop a testing strategy that would allow for the delivery of significantly more different configurations in a shorter time without significantly increasing testing effort and without sacrificing quality. To achieve this, the team applied a combination of different strategies: feature modeling, combinatorial testing, and deployment-based testing. Additionally, they built upon two ideas: restricting testing of feature interactions to those with any kind of dependency and those that should be included in an offering to customers.

The results demonstrate that the testing strategy allows for ensuring the quality of significantly more deliverable configurations without a significant increase in testing effort. Moreover, the strategy enables the addition of new functionality while accurately identifying which test cases require adjustments or new development. This allows for significant reuse of existing test cases, contributing significantly to the time-to-market and the transition to monthly release cycles.

Overall, this paper provides valuable insights and guidance for practitioners and researchers working with SPLs and facing the challenges of testing them. The presented strategy offers a promising approach to reducing the number of necessary cross-module test cases in the context of software platform development, demonstrating the feasibility and potential benefits of this approach.

CCS CONCEPTS

• Software and its engineering → Software creation and management → Software verification and validation → Empirical software validation

• Software and its engineering → Software organization and properties → Software system structures → Software system models → Feature interaction

KEYWORDS

Software product Line, feature model, deployment-based testing, test case construction

ACM Reference format:

Udo Knop, Peter Hofman, Michael Mihatsch, and Martin Siegmund. 2023. Balancing Variability and Costs in Software Product Lines: An Experience Report in Safety-Critical Systems. In *Proceedings of SPLC conference (SPLC'23), August 28 – September 1, 2023, Tokyo, Japan. ACM, New York, NY, USA, 9 pages.* https://doi.org/10.1145/1234567890

1 Introduction

Software Product Lines (SPLs) are an effective approach for exploiting commonalities between products while providing variability to allow individuality of products. SPLs offer several advantages, such as customization, increased market reach, better resource utilization, improved time-to-market, increased flexibility, improved maintainability, and additional cost savings. However, variability also comes with a cost, including increased development and maintenance costs, testing costs, and complexity [15]. Finding a balance between the amount of variability offered and the cost of that variability is a crucial aspect of SPL development.

Pohl et al. [14] have investigated how SPLs can help to achieve the challenges in software testing in general and emphasize the importance of testing all possible feature interactions, while acknowledging the contribution to model-based testing. Nevertheless, Metzger et al. [12] conclude that more research is required on product line quality assurance techniques despite the impressive progress made so far. There is a significant amount of literature on how to determine meaningful test configurations, for example, using combinatorial testing techniques like t-wise testing (e.g. Baranov et al. [2], Ferreira et al. [4]). However, constructing concrete test cases for all possible feature interactions remains a challenge.

This paper presents an experience report illustrating the difficulties of testing SPLs and how a combination of various strategies led to convincing results. Specifically, we focus on a SPL for a safety-critical system in the healthcare industry with approximately 900 optional features, which could potentially offer significant variability to customers. Testing this variability using conventional approaches is an immense challenge due to the vast number of required test cases. Consequently, the actual variability offered to customers was restricted to four (!) configurations. The high testing effort also limited the time-to-market of new features to four releases per year. The goal of the project was to develop a testing strategy that would allow for the delivery of significantly more different configurations in a shorter time without significantly increasing testing effort and without sacrificing quality.

To achieve this, we applied a combination of different strategies: feature modeling, combinatorial testing, and deployment-based testing. Additionally, we introduced two new ideas: restricting testing of feature interactions to those with any kind of dependency and those that should be included in an offering to customers. We introduced the concept of "semantic dependencies," an extension of the feature model that includes additional dependencies beyond those required for determining buildability. We also introduced "customer-required sub-graphs," which represent the parts of the feature model potentially deployed to customers and subject to testing.

In this paper, we provide a detailed evaluation of our testing strategy. Our results demonstrate that our strategy allows for ensuring the quality of significantly more deliverable configurations without a significant increase in testing effort. Moreover, our strategy enables the addition of new functionality while accurately identifying which test cases require adjustments or new development. This allows for significant reuse of existing test cases, contributing significantly to the time-to-market and the transition to monthly release cycles.

The rest of the paper is organized as follows: in the next section, we present the initial situation of our project, the challenges faced by stakeholders, and the objectives we aimed to achieve. In section 3, we introduce some background and related concepts relevant to our case. In Sections 4, we highlight the key elements of our approach, including the extension of the feature model and the construction of test cases that consider all restrictions in this model. Finally, in Section 5, we describe the phased approach we selected to meet our objectives in the project and we present the results which we achieved as of today. We conclude the paper by discussing the limitations of our approach, identifying areas for future work, and offering our final conclusions.

2 Initial Situation

In our project, we focused on a large software platform used within several product lines of medical devices. This platform had a significant number of variation points, as many features were optional and could be configured to be included or excluded in a deployment. About 900 features on requirement level were explicitly modeled in a feature model. This level of variability made testing a significant challenge.

The software platform exhibited a significant number of intricate dependencies among the software's architectural building blocks, as well as between different features. Moreover, there were several dependencies on external software components, such as open-source and off-the-shelf products. Although these dependencies were captured in the feature model as far as buildability was concerned, there were instances where features had technical dependencies, such as shared access to database tables, that could be deployed independently. Consequently, the feature model did not reflect these dependencies, as they did not affect buildability. Nevertheless, these dependencies were well-known to the development team and were verified through manual test cases.

To control variability cost, modules were introduced. A module is a reusable, modular, cohesive and domain specific grouping of features. The development approach for the software platform is based on a staged testing approach, with intensive testing of features and feature dependencies on the module level, followed by intensive testing on the integration level, which we refer to as "Cross-Module Tests". Most module level tests could be automated, thus limiting variability cost on module level. Still, integration level "Cross-Module Tests" mostly were hardware-bound and executed manually.

To further control variability cost, deployment sets were introduced. A deployment set refers to a specific combination of modules (and thus features) that are included in a software deployment. While a deployment set always has the same modules as part of the physical deployment, some features were designed to be activated or deactivated through configuration. "Cross-Module Tests" were then performed for every deployment set.

The platform was deployed at approximately 35.000 end-customer installations. While the technical prerequisites were in place to adapt the respective deployment individually to the customer's needs, the number of actual deployment sets built for testing purposes was still limited due to high quality assurance efforts. Before the project's start, the tests (and thus variability available to the customer) were limited to four fixed deployment sets.

For each deployment set, manual test cases were created with the requirement of testing all "Cross Module" feature interactions between features which were included in the respective deployment set. A combination of two feature which was included in two different deployment sets was thus tested in two distinct test cases. These redundant tests of feature interactions resulted in an almost linear increase in test effort with the number of deployment sets, as shown in Figure 1. This made the creation of new deployment sets costly.

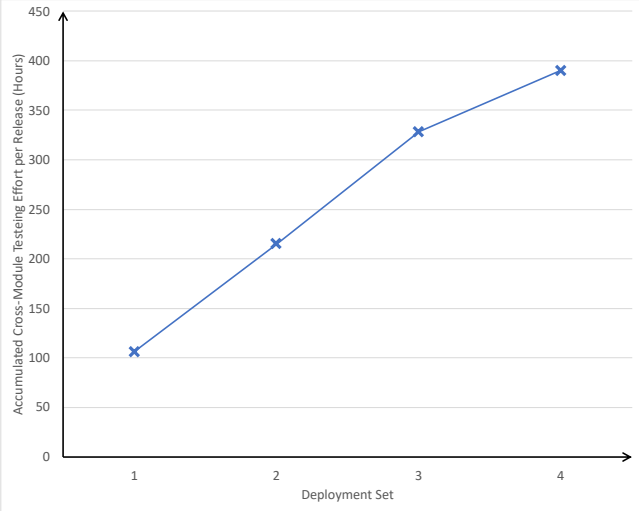


Figure 1: Accumulated Test Effort per Deployment Set

The test cases typically contained many features, as the team aimed to combine interacting features into test cases. Furthermore, the introduction of new features required all test cases that contained interacting features to be updated, even if only a few new features were introduced. As a result, many test cases for often all test configurations had to be adjusted.

The large number of feature interactions presented a significant challenge for testing the software product line, making it infeasible to test all possible interactions. However, the challenge was not to test all feature interactions but rather to avoid updating all test cases whenever a change was made, even if many feature interactions remained unchanged. Additionally, the fixed deployment sets were inflexible in accommodating new functionality and adding additional deployment sets resulted in redundant testing of many feature interactions.

This situation necessitated the development of a new testing strategy that could handle the large number of feature interactions and allow for more flexible testing of new functionality.

2.1 Business Challenges

The main business challenge was that increasing testing effort on integration level limits the variability that can be offered to the customer. Because of in this case high regulatory requirements for software quality, this severely limited the ability to introduce new features and products to the market.

Shortening time to market was the second major challenge. While monthly releases were desirable, releases were limited to a quarterly basis, mainly due to the effort required to execute tests and adapt test cases to the new release content.

A third challenge was connected to the size of installation packages for end customers. Physical deployment, such as via the internet, presented a challenge, particularly for software updates. On the one hand, update-related downtime of the product is expensive for the customer. On the other hand, support for many different software versions in the field added to maintenance cost of the SPL.

Additionally, by limiting deployment sets, modules had to be included in the deployment that were not needed by the customer. Besides influencing the size of the deployment package, this resulted in license costs for off-the-shelf (OTS) software that had to be included in the deployment but was not required. Furthermore, there was an unnecessary potential security risk for every module, feature, or OTS software that was deployed but not required.

2.2 Technical Challenges

The technical challenges in this project were centered around ensuring that all cross-module dependencies in the software were covered by test cases, including both direct and indirect (transitive) dependencies. Each test case had to be executed with a specific deployment set that included and excluded features. Due to quality and regulatory requirements, each feature combination that was deployed to an end customer had to be tested beforehand. However, testing all technically possible feature combinations, including indirect dependencies, would have resulted in an excessively high number of feature combinations, making the testing process infeasible.

2.3 Objectives

The main objectives of this research were to reduce the increase in testing effort on integration level, when introducing additional variability or functionality. This addresses the main business challenge, i.e. to allow the SPL to support additional products and to increase the variability in possible software deployments for end customers and to improve time to market. Importantly, this has to be done without sacrificing on quality (in our case: covering all

feature interactions) despite the increasing number of to-be-tested feature combinations.

Meeting this objective also acts as an enabler to addressing additional challenges, as briefly described in the outlook.

3 Background

Testing is an essential part of the SPL development process, but it presents a unique challenge due to the large number of possible feature interactions. The combinatorial explosion of feature interactions poses a major challenge for SPL testing, necessitating the use of efficient and effective testing strategies as pointed out e.g. by Cohen et al. [1] and Engström et al. [3]. One approach for reducing the number of required test configurations in large software systems is t-wise testing. Originally designed to reduce the number of test cases required to achieve a certain level of coverage, t-wise testing has been applied to SPLs as a means of reducing the number of required test configurations while still ensuring adequate coverage of feature interactions. T-wise testing aims to select a representative subset of test configurations that cover all possible combinations of t features (where t is a parameter specified by the tester). However, even with t-wise testing, the challenge of constructing concrete test cases for a given set of test configurations remains. This is particularly difficult for SPLs, where the large number of possible feature interactions can lead to a combinatorial explosion of test cases.

Recent advancements in SPL testing focus on challenges such as managing variability and modeling dependencies. Techniques such as feature modeling (e.g. Kang et al. [10]) and combinatorial testing (e.g. Oster et al. [13]) have been proposed to manage the complexity of SPL testing. Novel approaches for test case generation and selection, such as model-based testing, product sampling, and evolutionary algorithms, have also been proposed to improve the efficiency and effectiveness of SPL testing (e.g. Ferreira et al. [5], Galindo et al. [7], Henard et al. [8], Jung et al. [9], Varshosaz et al. [16], Xiang et al. [17]).

Our strategy adopted in the project combines three main techniques: feature modeling, combinatorial testing, and deployment-based testing. Feature modeling is a technique for representing the commonalities and variabilities of a SPL using a feature model. Combinatorial testing, like t-wise testing, is a technique that generates a set of test cases that cover all possible combinations of features. Deployment-based testing is a technique that focuses on testing only feature interactions that are delivered to customers. By combining these three techniques, we ensure that all feature interactions that are delivered to customers are thoroughly tested, while keeping the number of test cases to a minimum that is actually used. This approach is detailed in the following two sections.

In this paper, we also adopt a perspective on key terminology in the realm of software product line engineering which places greater emphasis on the requirements aspect rather than the implementation aspect. By doing so, we aim to provide a fresh viewpoint that aligns more closely with the needs and challenges faced by practitioners in the field. To establish a common understanding and lay the groundwork for our discussion, we offer brief definitions of some essential terms, adapted to reflect our requirements-centric focus:

- The **Products** in a Software Product Line (SPL) are described by the properties they have in common with each other and the variations that set them apart. The descriptions are in terms of the products' features.
- A **Feature** is a distinguishing characteristic of a product, usually visible to the customer or user of that product. An example is a capability that some products have but that others do not.
- A **Module** is a reusable, modular, cohesive and domain specific grouping of features, and
- An **Architectural Building Block** refers to technical components of the software, such as a Module and its lower-level components, like classes. While features may be implemented by an Architectural Building Block, their implementation is often distributed across multiple blocks.
- A **Deployment Set** refers to a specific combination of features that are included in a software deployment. It represents the features that are delivered to customers or internal users for use in their specific environment.
- A **Module Test** tests a single module and all its features in isolation, while a **Cross-Module Test** tests the interactions between features in multiple modules.

Our approach to these definitions diverges from the conventional usage found in the literature in some points, as we intentionally emphasize the role of requirements in shaping software product lines. This shift in perspective allows us to explore novel insights and opportunities for enhancing software development practices, ultimately contributing to the ongoing evolution of the field.

4 Solution

4.1 Testing strategy

The testing strategy is a combination of extensions to the feature model, combinatorial testing, and deployment-based testing. The following steps were taken to implement this strategy:

1. The feature model was extended to capture all pairwise feature interactions as dependencies. This ensured that all feature interactions were considered during testing.
2. The feature model was further extended to specify which feature combinations could be delivered to customers. This allowed for more targeted testing of feature interactions that were relevant to customers.
3. All feature combinations that could potentially interact in a customer deployment were identified, including transitive dependencies.
4. A test case was developed for all possible combinations of two interacting features F and G, including F & G, F & ¬G, ¬F &G, and ¬F & ¬G (given that individual combination is buildable and can be part of a customer deployment).

5. A minimal set of test configurations was identified to ensure that all identified feature combinations were tested in at least one configuration.
6. Each test case was finally assigned to a test configuration.

It is important to note, that step 6 happens after development of the test-cases (step 4). Thus, while writing the test cases, the possibility of configuration changes in the future has to be taken into account. In the next two sections, these steps will be explained in more detail.

4.2 Modelling approach

Conventional feature models are typically used to determine which concrete software configurations can be generated. We refer to this as "buildability" of a software configuration. Consequently, it is sufficient to include only those dependencies in the feature model which affect buildability. Our modelling approach extends the conventional feature model by also including feature interactions, which do not have an impact on buildability but nevertheless result in a feature interaction which must be tested, and we explicitly distinguish these dependencies from the former. We refer to these as "semantical" dependencies.

Another element of our modeling approach involves modeling which feature combinations should be offered to customers. To systematically present these two extensions, we begin by first examining the feature model that was in place before the start of the project, see Figure 2.

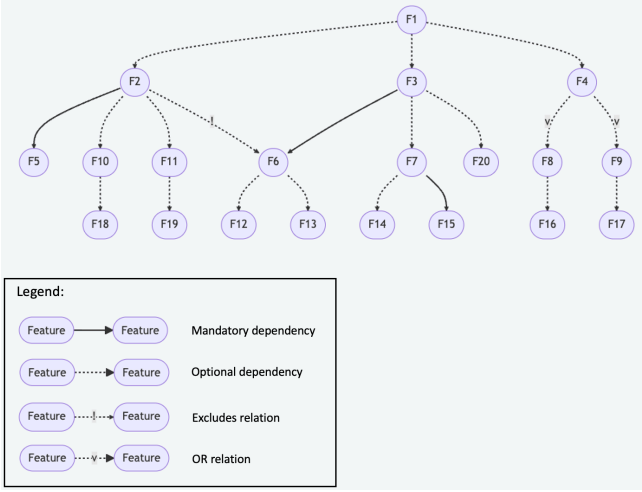


Figure 2: Original Feature Model

This is a conventional feature model where the features of the software platform are modelled as a graph with features as nodes and edges for dependencies. The model includes the following elements:

- **Mandatory dependencies**, which mark features that must be deployed together. These dependencies must be covered by test cases and limit test configurations to "buildable" configurations, meaning that mandatory dependencies must be obeyed.
- **Optional dependencies** follow the usual semantics of conventional feature models. The feature and all its optional dependencies must be covered by test cases. In the context of this project, this means that even configurations in which optional features are absent must be covered by a test case. Thus, they contribute significantly to the increase in the number of test configurations and test cases.
- **Exclude relations**, introduced to mark features that cannot be deployed together. This helps limit test cases as such feature combinations do not need to be tested.
- **OR dependencies**, introduced to mark features that require at least one alternative from a set of other features, such as storage options. These dependencies are like optional dependencies, but do not require a test case where none of the dependent features is combined with the parent feature.

In addition to these conventional dependency types, we have introduced a special kind of optional dependency called semantic dependencies. These dependencies are used to mark features that may not have a direct relation to each other, but whose functional behavior may be affected indirectly. A similar approach has been described by Lee et al. [2006] as "dynamic dependencies". Semantic dependencies are usually used in our project for infrastructure features like job scheduling. For instance, the job view (feature A) must reflect the progress of a transfer job (feature B). In this scenario, the behavior of feature A is influenced by B only if B is utilized. As a result, the combination of A and B must be tested, however it is not required to test A without B if A is covered by other test cases.

The modelling approach for the proposed solution further extends the feature model of the software platform to model customer deployments and to-be-supported feature combinations. This significantly restricts the possible product configurations, allowing only those combinations that are permissible according to the feature model during final product configuration.

A concrete customer deployment can be modeled as a subgraph of the feature model, containing only the nodes associated with features included in the deployment. Our approach extends the feature graph by adding additional attributes on edge and node levels to mark "to-be-supported" subgraphs. These subgraphs, defined based on customer requirements, are called "Customer Required to-be-supported Subgraphs" (CRS) in our model and each CRS must be covered by test cases, whereas feature combinations which are not inside any CRS do not have to be tested, see Figure 3.

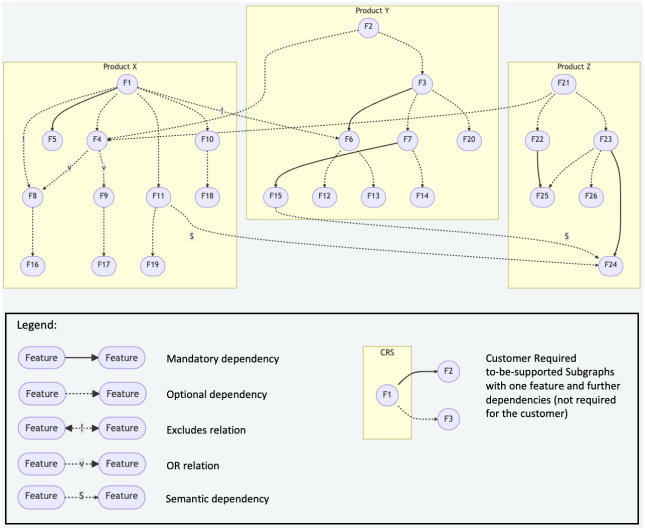


Figure 3: Extended Feature Modell with Semantic Dependencies and CRS

It is worth mentioning that although not depicted in this figure for the sake of simplicity, CRSs can overlap, which means that different CRSs may share the same features. For instance, infrastructure features like job management are usually included in multiple CRSs.

CRS play a crucial role in our solution strategy as they can significantly reduce the number of test cases if utilized effectively. A CRS still contains variability as the customer can choose any feature within the CRS, but it also restricts variability by limiting the choices to only those features that are part of the CRS. Finding the right balance between these two extremes is crucial:

- A single CRS that covers the entire feature model would offer the customer maximum variability, but would result in the maximum number of test cases.
- Creating a unique CRS for each customer that only includes their required features would result in a minimum number of test cases, but would also result in a huge number of CRS and long delivery cycles for each new customer.

By understanding customer needs, it is possible to create CRS that are relatively small in terms of variability but meet the needs of most customer groups. Furthermore, as we will show in the next section, adding new CRS results in only a limited increase in additional test cases.

4.3 Test Case Construction

Our test case construction process is based on automated analysis of the feature model and combinatorial sampling but has some unique features. Firstly, we take into account any level of interaction between features by including all feature combinations that are directly or indirectly linked through dependencies. Additionally, we only consider combinations of features that appear in the CRSs, ensuring that only configurations actually used are tested. To achieve this, our approach involves several steps:

1. **Computing all Customer Required to-be-supported Subgraphs (CRS)s:** First the CRSs and all contained features are determined from the feature model.
2. **Computing the Test Case Requirements (TCRs):** These need to adhere to the feature model ("buildability") and take into account all dependencies between features. More over, only those TCRs that are part of a CRS are considered. The exact process is explained below.
3. **Filtering the TCRs:** The input contains a list of TCRs to be ignored based on knowledge of domain experts. These are removed from the following steps.
4. **Computing minimal amount of Quality Assurance Sets (QAS):** These are software configurations which together contain all TCRs in the sense that all features in the TCR are also included in the configuration and all features which are explicitly marked as to be excluded in the TCR are not included in the configuration. In this case study, we aim to minimize the number of QASs required, in addition to covering all TCRs as setting up each QAS incurs significant costs.

The computation of the TCRs is based on the specific types of dependencies which exist in the feature model.

1. Firstly, we identify the "base features", which are those directly required by a CRS.
2. We then compute the connectivity component for each base feature: Starting from each base feature we follow all mandatory and optional dependencies to get the set of relevant features for this base feature.
3. We then compute all "buildable" combinations of these features using a SAT-solver. Buildable combinations must adhere to the dependencies in the feature model and appear in a CRS.
4. Next, we reduce these configurations by eliminating any features that are only connected to the base feature through other features that only appear negatively in the configuration.
5. Finally, we convert the resulting configurations into Boolean formulas to obtain the Test Case Requirements (TCRs), which form the basis for our manual test case construction.

We can illustrate the above steps using the following graph as an example:

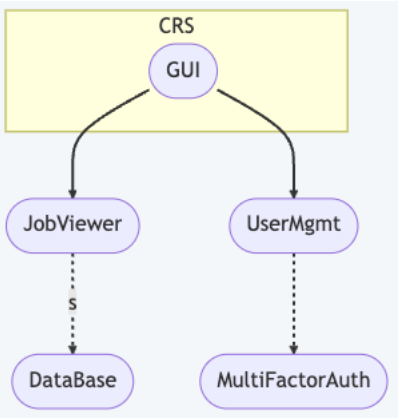


Figure 4: TCR Computation Example

1. Base Features: Here the only base feature is *GUI*, as it is the only feature directly required by the CRS.
2. Connectivity Components: *GUI* leads to *JobViewer* and *UserMgmt*, while *JobViewer* leads to *DataBase* and *UserMgmt* leads to *MFA*. Thus, the connectivity component contains those five features and the connecting dependencies. The set of relevant features is thus:
   { *GUI*, *JobViewer*, *UserMgmt*, *DataBase*, *MFA* }
3. Buildable combinations: The only relevant dependencies for buildability are the mandatory ones between *GUI* and *JobViewer*/*UserMgmt*. Therefore, there are no buildability restrictions for *DataBase* and *MFA*. The buildable combinations are thus:
   - *GUI* & *JobViewer* & *UserMgmt* & *DataBase* & *MFA*
   - *GUI* & *JobViewer* & *UserMgmt* & *DataBase* & *-MFA*
   - *GUI* & *JobViewer* & *UserMgmt* & *-DataBase* & *MFA*
   - *GUI* & *JobViewer* & *UserMgmt* & *-DataBase* & *-MFA*

   Since appearance in a CRS is also relevant, we need to compute the configuration, which is the smallest buildable configuration containing all needed features. Thus, the configuration contains *GUI*, *JobViewer* and *UserMgmt* since the dependencies to *DataBase* and *MFA* are not relevant for buildability, as they are optional. The only buildable combination that can appear in a CRS is then:
   - *GUI* & *JobViewer* & *UserMgmt* & *-DataBase* & *-MFA*
4. Configuration reduction: Since semantic dependencies are only interested in positive targets, *-DataBase* can be removed. *-MFA* is relevant though, as it appears due to an optional dependency.
5. Boolean formulas: We already expressed the configurations as formulas, thus the resulting TCR is:
   - *GUI* & *JobViewer* & *UserMgmt* & *-MFA*

5 Achievements and Future Work

5.1 Solution Roadmap

The implementation of our proposed solution strategy requires several steps to be taken.

- Implement tooling to define and maintain the feature model with the extensions outlined in Section 4.2.
- Implement tooling to specify CRSs and integrate CRSs into the current configuration process and tooling.
- Implement the calculation of TCRs and QASs.
- Migrate current test cases and deployment sets used for testing to reflect the computed TCRs and QAS.

This change affects a wide range of stakeholders in the organization: software and product engineers responsible for the feature model, testing managers and testing engineers responsible for defining the final test cases based on TCRs, devops engineers responsible for building deployment sets (testing and customer related), product managers responsible for defining CRSs, sales representative responsible for configuring customer deployments, and maintenance staff responsible for supporting deployed software configurations. Due to its impact, a staged approach is required. Our proposed solution roadmap includes the following phases:

- **Phase 1 - Concept Validation**: This phase involves validating the concept of CRS by using the current deployment sets used for testing. The feature model tooling is updated to accommodate the new extensions and the TCR computation is implemented. A pilot set of features is modeled, and respective test cases are migrated based on the resulting TCRs.
- **Phase 2 - Feature Support for a New Customer Group**: This phase involves extending the feature model with new features required to support a new customer group. TCRs, QASs, and customer-related deployment sets are constructed based on the new approach.
- **Phase 3 - Configuration and Build Process Adherence to Extended Feature Model**: This phase involves updating the tooling used for configuring customer-related deployment sets with additional validation checks to ensure that only configurations that comply with the extended feature model are deployed. This includes tooling to specify new CRSs and evaluate the impact on testing efforts by computing metrics such as the number of additional TCRs required. The tooling for building deployment sets based on CRSs is also updated.
- **Phase 4 - Full Rollout**: This phase involves fully implementing the new approach across all business lines, including sales.

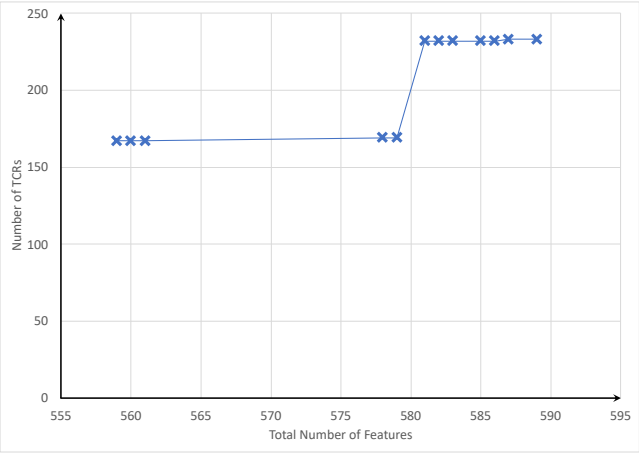
5.2 Implementation Status and Future Work

At the beginning of the project, the feature model was primarily used to model the variability of the software platform. As of the

writing of this paper, phases 1 and 2 of the solution roadmap have been successfully implemented. This includes an automatic analysis of the feature model to determine TCRs and QASs. The new testing strategy has been applied to the first deployment set and an additional 11 offerings without a significant increase in cross-module testing efforts. Phase 3 has been piloted already and requires future work to complete. As described, this also includes the use of the feature model for product configuration.

5.3 Results as of today

Figure 5 shows how the number of TCRs changes when features are added. Each data point corresponds to a CRS that is to be offered on the market, where the first data point belongs to the first of the four originally migrated deployment sets, and each subsequent data point corresponds to an additional offering that was either not available on the market before or had to be tested manually.



Customer-Required to-be-supported Subgraph	1	2	3	4	5	6	7	8	9	10	11	12
Number of Included Features	559	559	559	526	526	296	268	268	44	43	13	2
Thereof New Features		1	1	17	1	2	1	1	2	1	1	2
Total Number of Features (without duplicates)	559	560	561	578	579	581	582	583	585	586	587	589
Number of TCRs	167	167	167	169	169	232	232	232	232	232	233	233

Figure 5: Number of Computed TCRs when Adding Features

The following observations can be made:

- Most CRSs do not result in a significant increase in the number of required TCRs. For example, CRS 4 introduces 17 new features, but only results in 2 additional TCRs. This is because the new features either have very few dependencies on other CRSs or because dependencies only exist between features that are already part of the previous CRSs, and thus, are already covered by existing TCRs. It is a first success of our approach that this redundancy is automatically detected.
- Between CRS 5 and 6, only two features are added (581 instead of 579), but they have several dependencies on already existing features. This results in an increase in the required TCRs from 331 to 444, or by 34%. This increase is significant but makes sense in the context of expected feature interactions. It demonstrates how transparency about the degree of cross-module interactions can be gained through a gradual addition of new features.

In conclusion, the proposed solution strategy demonstrates a promising approach to reducing the number of necessary cross-module test cases in the context of software platform development. The successful implementation of the first two phases of our solution roadmap, as well as the promising results shown in the presented graph, demonstrate the feasibility and potential benefits of our approach.

6 Limitations

We have presented a strategy for balancing the amount of variability offered in software product lines with the cost of that variability, particularly in the area of testing. To effectively implement this strategy, a thorough understanding of customer needs and the value that certain variability offers to customers is necessary. This knowledge is crucial for defining Customer Required to-be-supported Subgraphs (CRS), which play a key role in limiting the number of test cases.

However, in many situations, this level of detail in customer needs is lacking. Further research is needed to find ways to construct CRS based on the feature model itself, for example by identifying strongly connected subgraphs, and with limited information about customer needs, such as features valued most by customers. In addition, further research is required to identify metrics to assess the value of CRS for customers, as well as the costs of the additional variability introduced by them.

The expressiveness of the feature model plays a critical role in reducing the number of test cases in software product line testing. Our proposed solution has incorporated simple extensions to limit variability in the feature model. However, further research is necessary to explore the potential of creating even more expressive feature models that are intuitive and manageable for software engineers.

Additionally, there is potential for further extension of the proposed solution, including the following:

- Deriving technical dependencies from code or IDE analysis tools.
- Supporting different optimization boundary conditions for the generation of Test Case Requirements (TCRs) and Quality Assurance Sets (QASs), such as a minimal number of QASs or minimal changes to TCRs and QASs compared to an earlier version of the Software Product Line.

These limitations should be considered when evaluating the proposed solution and its potential for use in practical applications.

7 Conclusion and Outlook

This paper has discussed the challenges of testing SPLs and proposed a new approach to address these challenges. Cost of quality which is incurred by variability significantly adds to the

total cost of variability which limits to use of SPLs. Thus, ways are needed to limit the number of test cases (for existing and newly introduced feature interactions) without sacrificing quality and important variability required by customers.

The main contributions of the paper to this research area are:

- We propose to utilize the feature model for not only capturing variability but also for limiting it. We have outlined two methods for this purpose: (1) using subgraphs of the feature model to represent customer needs, and (2) introducing additional dependency types to restrict unnecessary feature combinations. This enhances the feature model and reduces the number of necessary test cases by focusing only on relevant feature interactions.
- The proposed approach combines combinatorial testing and deployment-based testing and prioritizes test cases based on risk or impact to the system using feature-use analysis.

In conclusion, the proposed approach provides a way to reduce the number of test cases in SPLs without sacrificing quality and important variability. Further research and implementation of this approach could lead to significant benefits for the SPL community, including increased variability in software deployments and improved quality assurance.

Galindo et al. [2018] describe in which areas the automatic analysis of feature models can be applied. This experience report demonstrates how more and more benefits can be derived from a feature model during the course of our solution roadmap while extending the application of automated feature model analysis.

As an outlook, the proposed approach can act as an enabler for several business challenges. More variability in the SPL allows for more customer-specific deployments, thus reducing OTS-costs and potential security risks introduced by not deeded modules. Additionally, customer-specific installation packages can be smaller and faster to update. This can help to reduce the number of old software versions, that have to be supported in the field. Finally, introducing additional variability is a prerequisite for feature-subscription based pricing models.

REFERENCES

[1] David M. Cohen, Siddhartha. R. Dalal, Jesse Parelius and Gardner C. Patton, "The combinatorial design approach to automatic test generation," in *IEEE Software*, vol. 13, no. 5, pp. 83-88, Sept. 1996. https://doi.org/10.1109/52.536462.

[2] Eduard Baranov and Axel Legay. "Baital: An adaptive weighted sampling platform for configurable systems." *Proceedings of the 2021 ACM SIGSOFT 23rd International Symposium on Software Testing and Analysis.* ACM, 2021. https://doi.org/10.1145/3503229.3547030.

[3] Emelie Engström and Per Runeson. "Software product line testing - A systematic mapping study." *Information and Software Technology* 52, no. 12 (2010): 1240-1249. https://doi.org/10.1016/j.infsof.2010.05.011.

[4] Fischer Ferreira, Gustavo Vale, João P. Diniz, and Eduardo Figueiredo. "Evaluating T-wise testing strategies in a community-wide dataset of configurable software systems." *Journal of Systems and Software 166 (2021):* 110990. https://doi.org/10.1016/j.jss.2021.110990.

[5] Thiago do Nascimento Ferreira, Silvia Regina Vergilio, and Marouane Kessentini. "Applying Many-objective Algorithms to the Variability Test of Software Product Lines." *Proceedings of the 40th International Conference on Software Engineering. ACM, 2018.* https://doi.org/10.1145/3425174.3425211.

[6] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz. "Automated analysis of feature models: Quo vadis?" *Journal of Software and Systems Modeling* 17, no. 4 (2018): 1095-1113. https://doi.org/10.1007/s00607-018-0646-1.

[7] José A. Galindo, Hamilton Turner, David Benavides, and Jules White. "Testing variability-intensive systems using automated analysis: an application to Android." *Journal of Software and Systems Modeling 13*, no. 4 (2014): 669-684. https://doi.org/10.1007/s11219-014-9258-y.

[8] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines." *IEEE Transactions on Software Engineering 40*, no. 8 (2014): 779-794. https://doi.org/10.1109/TSE.2014.2327020.

[9] Pilsu Jung, Sungwon Kang, and Jihyun Lee (2020). Efficient Regression Testing of Software Product Lines by Reducing Redundant Test Executions, https://doi.org/10.3390/app10238686.

[10] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson (1990). Feature-oriented domain analysis (FODA) feasibility study (No. CMU/SEI-90-TR-21). Software Engineering Institute.

[11] Yuqin Lee, Chuanyao Yang, Chongxiang Zhu, and Wenyun Zhao (2006). An approach to managing feature dependencies for product releasing in software product lines. In Reuse of Off-the-Shelf Components: 9th International Conference on Software Reuse, ICSR 2006 Turin, Italy, June 12-15, 2006 Proceedings 9 (pp. 127-141). Springer Berlin Heidelberg. https://doi.org/10.1007/11763864_10.

[12] Andreas Metzger and Klaus Pohl. "Software Product Line Engineering and Variability Management: Achievements and Challenges." In *Proceedings of the 8th International Conference on Software Product Line*, pp. 1-8. ACM, 2014. https://doi.org/10.1145/2593882.2593888.

[13] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik (2011, August). Pairwise feature-interaction testing for SPLs: potentials and limitations. In Proceedings of the 15th International Software Product Line Conference, Volume 2 (pp. 1-8). https://doi.org/10.1145/2019136.2019143.

[14] Klaus Pohl and Andreas Metzger. Software product line testing. *Communications of the ACM 49*. 12 (2006): 78-81. https://doi.org/10.1145/1183236.1183271.

[15] Klaus Pohl, Günter Böckle, and Frank Linden. "Software product line engineering: Foundations, principles, and techniques." *Springer Science & Business Media*, 2005. https://doi.org/10.1007/3-540-28901-1.

[16] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. "A classification of product sampling for software product lines." *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 2018. https://doi.org/10.1145/3233027.3233035.

[17] Yi Xiang, Xiaowei Yang, Han Huang, Zhengxin Huang, and Miqing Li. 2022. "Sampling configurations from software product lines via probability-aware diversification and SAT solving." *Journal of Systems and Software*, vol. 171, pp. 348-358. https://doi.org/10.1007/s10515-022-00348-8.