*ECAI 2023*
*K. Gal et al. (Eds.)*
*© 2023 The Authors.*
*This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).*
*doi:10.3233/FAIA230640*

# FMTESTING: A FEATUREIDE Plug-in for Automated Feature Model Analysis and Diagnosis

**Tamim Burgstaller, Viet-Man Le, Thi Ngoc Trang Tran and Alexander Felfernig**

Graz University of Technology, Graz, Austria
{tamim.burgstaller,vietman.le,ttrang,alexander.felfernig}@ist.tugraz.at
ORCiD ID: Tamim Burgstaller https://orcid.org/0009-0007-4522-8497,
Viet-Man Le https://orcid.org/0000-0001-5778-975X, Thi Ngoc
Trang Tran https://orcid.org/0000-0002-3550-8352, Alexander Felfernig https://orcid.org/0000-0003-0108-3146

**Abstract.** The increasing size and complexity of feature models (FMs) can trigger anomalies or faults, challenging stakeholders in keeping FMs consistent with the domain requirements. Existing quality assurance tools do not provide advanced techniques to point out possibilities to adapt an FM for consistency recovery. In this paper, we present FMTESTING, which is a plug-in for FEATUREIDE, an ECLIPSE-based IDE supporting different phases of feature-oriented software development. FMTESTING is capable of automatically generating property-based test cases based on six different types of FM analysis operations. Furthermore, for violated test cases, diagnoses are provided to precisely indicate faulty FM elements (constraints) that should be adapted to restore consistency. Our tool provides user interfaces inside FEATUREIDE to ensure convenient use, even for users who are not domain experts.

**Keywords:** Software Product Lines, Feature Models, Configuration, Constraint Satisfaction, Model-based Diagnosis, Direct Diagnosis

## 1 Introduction

Feature models (FMs) [11] are in wide-spread use for modeling Software Product Line (SPL) variabilities and commonalities. In FMs, product line features are organized in a hierarchical structure that reflects their relationships and dependencies. The root of the hierarchy represents the entire product line, and each node in the hierarchy represents a feature that can be selected or deselected to create a specific product variant. Extended feature models also support attributes and cardinalities, which can be used to specify further constraints among the features in the model [4, 25].

Feature models have been widely adopted to describe all the features and constraints for configuring valid software products in Software Product Lines [2, 11, 27]. FMs can become quite complex, making it highly challenging to keep them consistent with the domain requirements. Consequently, intelligent mechanisms for feature model quality assurance have to be provided [13, 16].

FM testing and debugging can be supported by *analysis operations* that help to check the conformance of an FM with regard to a set of well-formedness properties [3, 22]. An example of such a property (well-formedness rule) is the *non-existence of dead features*. Dead features are "inactive" in every possible feature model configuration, i.e., there does not exist a configuration which includes this feature.

Such well-formedness rules can be regarded as *test cases* that help specify a feature model's intended semantics. FM testing can be combined with corresponding diagnosis operations that help to identify faulty FM elements (constraints) in an efficient fashion. The underlying approach is to exploit test cases for the induction of *conflicts* [10] (some test cases are inconsistent with feature model constraints), which can then be resolved based on model-based diagnosis [7, 9, 22]. In this context, test cases are part of test suites that define intended FM semantics and thus help to assure FM quality [16].

Existing quality assurance techniques and tools focus on the "execution" of different types of analysis operations that analyze structural model properties [3, 9, 12, 20, 21, 24, 26]. Although these approaches can be used to indicate violations of predefined properties, they do not support in a focused fashion a pin-pointing of faulty feature model elements which is needed for FM adaptations [16].

In line with [5, 16, 19], we have developed FMTESTING[1] which is a plug-in for FEATUREIDE [26].[2] FMTESTING allows for an automated test case generation based on the properties of FM analysis operations. Our tool provides automated FM testing and debugging, on which corrective explanations are determined when property-based test case violations occur in a feature model. Differing from the existing tools, FMTESTING creates *why not* explanations in terms of minimal sets of constraints (diagnoses) responsible for faulty model semantics. This way, stakeholders can more easily detect relationships/constraints acting as a source of faulty feature model behavior.

The major contributions of this paper are the following. First, we show how the concepts of direct diagnosis [7] can be applied to the automated debugging of feature models. Second, we present the FMTESTING FEATUREIDE-plugin which integrates wide-spread feature model analysis operations [3] with knowledge base testing and debugging approaches.

The remainder of this paper is organized as follows. In Section 2, we introduce basic concepts of feature models (FMs), FM analysis operations, and FM testing and debugging. In Section 3, we present our implementation of the FMTESTING plugin including the corresponding technical (also diagnosis-related) backgrounds. Limitations of the current work and open issues for future work are discussed in Section 4. The paper is concluded with Section 5.

[1] github.com/AIG-ist-tugraz/FMTesting/releases/tag/v0.0.1
[2] featureide.de

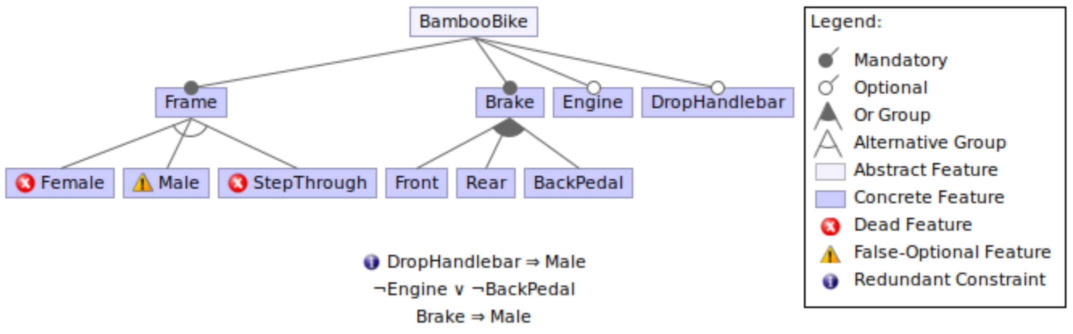


**Figure 1.** A working example of the *presumably faulty* feature model of a *Bamboo Bike* used throughout the paper. This model is captured from FEATUREIDE [26], where the highlighted anomalies such as *dead feature*, *false optional feature*, and *redundant constraint* are identified by the built-in engine of FEATUREIDE.

## 2 Preliminaries

In the following, we introduce basic concepts and techniques that have been integrated into the FMTESTING plugin.

### 2.1 Feature Models

Feature models are configuration models representing the variability properties of a software product line (SPL) in terms of features and their relationships [3, 6]. An example of the *presumably faulty* feature model of a *Bamboo Bike*[3] is depicted in Figure 1. Features are typically arranged in a hierarchical fashion using relationships such as *mandatory* (e.g., each *BambooBike* includes a *Frame*), *optional* (e.g., an *Engine* could be included), *alternative* (e.g., a *Frame* is either a *Female*, a *Male*, or a *StepThrough* frame), and *or* (e.g., a *Brake* can be defined as *Front*, *Rear* or also as *BackPedal*). Moreover, cross-tree constraints such as *excludes* and *requires* are also integrated into the model to set cross-hierarchical restrictions for features. For instance, *Engine* does not allow the inclusion of *BackPedal*, and the inclusion of the *DropHandlebar* feature requires the inclusion of the *Male* feature. For further details about feature model representation, we refer to Batory [1].

### 2.2 Feature Model Analysis

Feature model analysis operations can be regarded as the computer-aided analysis of feature model properties [3]. A set of 30 analysis operations have been identified [3], including operations for model consistency, anomaly detection, explanations, and feature model configuration capabilities. In the following, we highlight some analysis operations for anomaly detection used throughout the paper.

Anomalies are triggered by the increasing size and complexity of feature models. Some examples thereof are *void feature models* that do not represent any configuration, *dead features* that are not included in any possible configuration, and *conditionally dead features* that become dead under certain circumstances (e.g., when including specific feature(s) in a configuration). Anomalies also can be *full mandatory features* that are included in every possible solution, *false optional features* included in all configurations although they have not been modeled as mandatory, and *redundant constraints* that do not change the semantics of the feature model [17]. Figure 1 depicts a feature model with anomalies. In this example, due to the constraint $Brake \rightarrow Male$, *Male* is false optional, *Step-through* and *Female* become dead features, and thus, the constraint $DropHandlebar \rightarrow Male$ is redundant. For further details of feature model analysis operations, we refer to [3, 5, 12].

To support reasoning about feature model properties, feature models can be transformed into a *Constraint Satisfaction Problem* (CSP) [23], where each feature $f_i$ is related to the binary domain $\{(t)rue, (f)alse\}$. The mentioned relationships and cross-tree constraints are represented as constraints on the CSP level. A set of rules defining how to translate a feature model into a corresponding constraint-based representation is discussed in Benavides et al. [3]. Table 1 depicts the constraints $c_i \in CF$ derived from the feature model shown in Figure 1. In this context, $c_0 : BambooBike = t$ is a *root constraint* that avoids the derivation of empty configurations.

**Table 1.** $c_0$ and $CF = \{c_1..c_9\}$.

| constraint | CSP representation |
|---|---|
| $c_0$ | $BambooBike = t$ |
| $c_1$ | $BambooBike \leftrightarrow Frame$ |
| $c_2$ | $BambooBike \leftrightarrow Brake$ |
| $c_3$ | $Engine \rightarrow BambooBike$ |
| $c_4$ | $DropHandlebar \rightarrow BambooBike$ |
| $c_5$ | $(Female \leftrightarrow \neg Male \wedge \neg StepThrough \wedge Frame)$ $\wedge(Male \leftrightarrow \neg Female \wedge \neg StepThrough \wedge Frame)$ $\wedge(StepThrough \leftrightarrow \neg Male \wedge \neg Female \wedge Frame)$ |
| $c_6$ | $Brake \leftrightarrow Front \vee Rear \vee BackPedal$ |
| $c_7$ | $DropHandlebar \rightarrow Male$ |
| $c_8$ | $\neg(Engine \wedge BackPedal)$ |
| $c_9$ | $Brake \rightarrow Male$ |

### 2.3 Feature Model Testing

In order to support feature model quality assurance, test suites ($T$) can be applied to define the intended behavior of a feature model [16]. As part of a test suite, test cases are used to validate feature models against a predefined set of different properties required by feature model analysis operations [3, 20] (but are not limited to such types of feature model properties). *A test case is interpreted as a constraint representing the intended semantics of a feature model.* Examples of test cases $t_i$ (see also Table 2) in such test suites are $t_1 : Engine = t$ (assuring the existence of at least one configuration with $Engine$ activated) and $t_2 : Male = f$ (assuring the existence of at least one configuration with $Male$ deactivated). A further test case $t_3 : Female = t \wedge Engine = t$, specifying that there should exist at least one co-occurrence of the features $Female$ and $Engine$. For further details regarding feature model test cases and feature model test suites, we refer to [14, 16, 17].

[3] www.my-boo.com

Test cases can be derived from various sources, such as analysis operations, previously completed consistent feature model configurations, or can be manually defined by knowledge engineers in the feature model development and maintenance processes.

**Table 2.** An example set of test cases specifying the intended behavior of the feature model represented in Figure 1.

| ID | constraint |
|---|---|
| $t_1$ | $Engine = t$ |
| $t_2$ | $Male = f$ |
| $t_3$ | $Female = t \wedge Engine = t$ |

## 2.4 Feature Model Debugging

Feature model testing can be combined with corresponding diagnosis operations to identify *minimal* explanations (diagnoses - $\Delta$) for violated test cases. An example of such explanations would be a minimal set of constraints responsible for the faulty behavior of a feature model [16]. Such constraints have to be deleted/adapted to make the feature model consistent with $T$. The concept of *minimality* means that if $\Delta$ is minimal, there does not exist any other minimal diagnosis $\Delta'$ such that $\Delta'$ is a subset of $\Delta$. In our working example, $\Delta_1 = \{c_9\}$, $\Delta_2 = \{c_5\}$, and $\Delta_3 = \{c_2\}$ are the explanations showing different options for deleting/adapting the constraints in $CF$ such that the consistency between the feature model (Figure 1) and test cases $\{t_1..t_3\}$ (Table 2) is restored. In the following, we present the general idea and point out the advantages of the two basic algorithms implemented in FMTESTING to determine explanations for different feature model fault patterns.

**Determination of Diagnoses**. DIRECTDEBUG [16] extends the *direct diagnosis* approach [7] to support the automated testing and debugging of variability models. This algorithm follows the *divide-and-conquer* strategy and does not require any support of a conflict detection or a related derivation of hitting sets [22]. Moreover, it takes into account all the given test cases at the same time and identifies one diagnosis ($\Delta$) that makes all test cases consistent with the knowledge base.

DIRECTDEBUG requires four inputs: (1) $\delta$ that avoids redundant consistency checks, (2) the diagnosis candidates $C \subseteq CF$, (3) a background knowledge $B = CF \setminus C \cup \{c_0\}$, and (4) a set of test cases $T_\pi$. $\delta = \emptyset$ indicates that $C$ has already been checked for consistency with $B$. The algorithm returns an $MSS$ - $\Gamma$ (Maximum Satisfiable Subset - see *Definition 3* in [16]), a corresponding diagnosis is $C \setminus \Gamma$. For further details regarding the algorithm, we refer to [15, 16].

**Determination of Redundancies**. WIPEOUTR$_{FM}$ [17] is a *complete* algorithm assuring the identification of all redundant constraints in a feature model. This algorithm relies on the following assumption: "If a constraint $c_i \in CF$ is non-redundant, then its deletion from $CF$ will change the semantics of $CF$". In order to examine the non-redundancy of $c_i$, we first delete it from $CF$ and then add its negation to $CF$. By now, if the current $CF$ becomes consistent, then $c_i$ is non-redundant. This approach is more efficient than the approach that checks redundancy properties based on concrete configurations [5]. For further details on WIPEOUTR$_{FM}$, we refer to [17].

## 3 Feature Model Testing Tool: A plug-in for FeatureIDE

Our tool, FMTESTING, provides a mechanism to automatically generate property-based test cases and allows the automated determination of faulty constraints. The architecture and key components of the tool are presented in the following subsections.

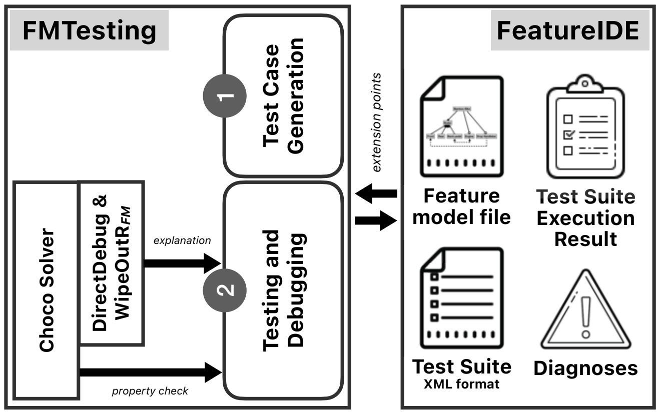


**Figure 2.** The architecture[5] of the FMTESTING with two key components.

### 3.1 Architecture

FMTESTING is built on the basis of *extension points* provided by ECLIPSE and FEATUREIDE. These extensions may contain simple editors and views to menu entries, specific settings, and much more. FMTESTING consists of two key components (see Figure 2): (1) *test case generation* that generates a test suite based on a selected set of feature model analysis operations, and (2) *feature model testing and debugging* that asynchronously executes a test suite validation and identifies corrective explanations (in terms of diagnoses) to resolve the violated tests. Such corrective explanations provide different ways to delete/adapt the relationships/constraints in a feature model and therefore, resolve its anomalies.

Due to the time-consuming nature of test case generation as well as feature model testing and debugging, FMTESTING exploits the parallelization features of the available implementation frameworks Java ForkJoin[6] that takes into account available resources of the machine and therefore reduces its execution runtime.

### 3.2 Test Case Generation

FMTESTING extends the checking methods proposed by Felfernig et al. [5] to generate six types of test cases: *void feature models*, *dead features*, *conditionally dead features*, *full mandatory features*, *false optional features*, and *redundant constraints*. Table 3 shows test case templates for the corresponding analysis operations as well as related property checks and explanations. FMTESTING gets a feature model in the FEATUREIDE XML format as the input and stores the generated test suite in a XML file. FMTESTING allows knowledge engineers to choose test case types (analysis operations) to include in a test suite (see Step 1 of Figure 3).

### 3.3 Automated Testing and Debugging

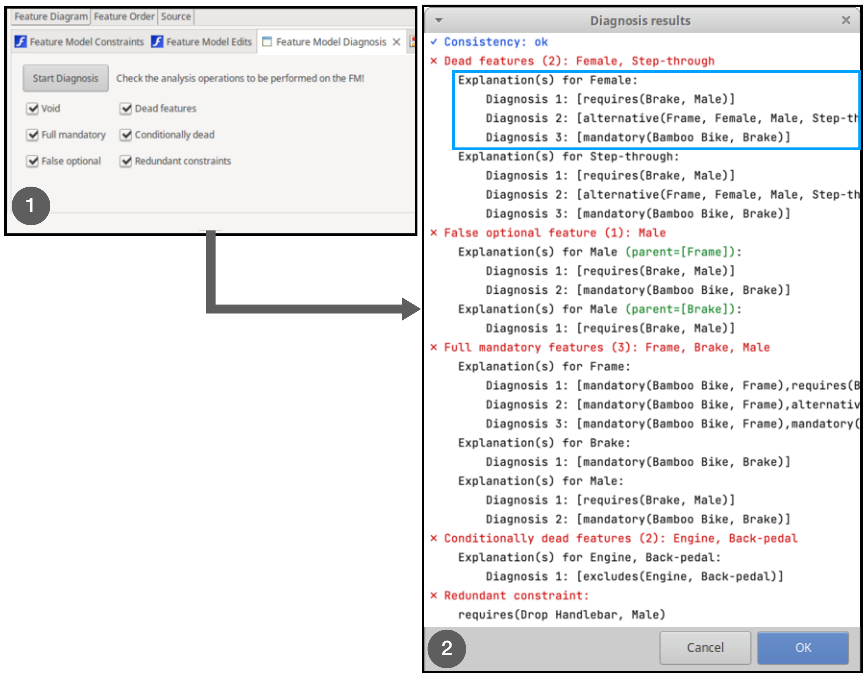
FMTESTING provides a testing and debugging engine that executes asynchronously the validation of every test case of a given test suite. In case of any violated test cases, the engine activates in parallel

[5] In Figure 2, we use the icons from https://icons8.com, including *General Warning Sign* icon, *File* icon, *List* icon, and *Survey* icon.

[6] https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

**Table 3.** Feature model analysis operations, test case templates, property checks, and related explanations. For example, a consistency check $inconsistent(CF \cup t_i)$ is activated to figure out whether a *void feature model* test case ($t_i = \{c_0\}$) is violated. A related explanation can be determined by solving the FM diagnosis operation with ($\emptyset$,$CF$,$\emptyset$,$t_i$). The algorithms for identifying diagnosis (DIRECTDEBUG) and detecting redundancy (WIPEOUTR$_{FM}$) are presented in Section 2.4.

| **Analysis Operation** | **Test Case Template** | **Property Check** | **Explanation (Diagnosis Task)** |
|---|---|---|---|
| Void feature model | $t_i = \{c_0\}$ | *inconsistent*$(CF \cup t_i)$ | DIRECTDEBUG($\emptyset$,$CF$,$\emptyset$,$t_i$) |
| Dead ($f_i$) | $t_i = \{c_0 \wedge f_i = true\}$ | *inconsistent*$(CF \cup t_i)$ | DIRECTDEBUG($\emptyset$,$CF$,$\emptyset$,$t_i$) |
| Conditionally dead ($f_i$) | $t_i = \{c_0 \wedge f_j = true \wedge f_i = true\}$ *where* $f_i$ *is optional and not dead,* $f_j$ *is not dead* | *inconsistent*$(CF \cup t_i)$ | DIRECTDEBUG($\emptyset$,$CF$,$\emptyset$,$t_i$) |
| Full mandatory ($f_i$) | $t_i = \{c_0 \wedge f_i = false\}$ | *inconsistent*$(CF \cup t_i)$ | DIRECTDEBUG($\emptyset$,$CF$,$\emptyset$,$t_i$) |
| False optional ($f_i$) | $t_i = \{c_0 \wedge f_j = true \wedge f_i = false\}$ *where* $f_i$ *is optional and not dead,* $f_j$ *is the mandatory parent of* $f_i$ | *inconsistent*$(CF \cup t_i)$ | DIRECTDEBUG($\emptyset$,$CF$,$\emptyset$,$t_i$) |
| Redundant ($c_i$) | $t_i = \{c_i\}$ | *inconsistent*$(CF \setminus t_i \cup \neg t_i)$ | $c_i \in$ WIPEOUTR$_{FM}(CF)$ |



Check the analysis operations to be performed on the FM!

[✓ Void] [✓ Dead features]
[✓ Full mandatory] [✓ Conditionally dead]
[✓ False optional] [✓ Redundant constraints]

```
✓ Consistency: ok
× Dead features (2): Female, Step-through
    Explanation(s) for Female:
        Diagnosis 1: [requires(Brake, Male)]
        Diagnosis 2: [alternative(Frame, Female, Male, Step-th
        Diagnosis 3: [mandatory(Bamboo Bike, Brake)]
    Explanation(s) for Step-through:
        Diagnosis 1: [requires(Brake, Male)]
        Diagnosis 2: [alternative(Frame, Female, Male, Step-th
        Diagnosis 3: [mandatory(Bamboo Bike, Brake)]
× False optional feature (1): Male
    Explanation(s) for Male (parent=[Frame]):
        Diagnosis 1: [requires(Brake, Male)]
        Diagnosis 2: [mandatory(Bamboo Bike, Brake)]
    Explanation(s) for Male (parent=[Brake]):
        Diagnosis 1: [requires(Brake, Male)]
× Full mandatory features (3): Frame, Brake, Male
    Explanation(s) for Frame:
        Diagnosis 1: [mandatory(Bamboo Bike, Frame),requires(B
        Diagnosis 2: [mandatory(Bamboo Bike, Frame),alternativ
        Diagnosis 3: [mandatory(Bamboo Bike, Frame),mandatory(
    Explanation(s) for Brake:
        Diagnosis 1: [mandatory(Bamboo Bike, Brake)]
    Explanation(s) for Male:
        Diagnosis 1: [requires(Brake, Male)]
        Diagnosis 2: [mandatory(Bamboo Bike, Brake)]
× Conditionally dead features (2): Engine, Back-pedal
    Explanation(s) for Engine, Back-pedal:
        Diagnosis 1: [excludes(Engine, Back-pedal)]
× Redundant constraint:
    requires(Drop Handlebar, Male)
```

**Figure 3.** The usage of FMTESTING follows two steps. In Step 1, after modeling a feature model using FEATUREIDE, FMTESTING first allows a user to select (from the *Feature Model Diagnosis* tab) the types of test case. Then, the user clicks on the *Start Diagnosis* to trigger the testing and debugging execution. In Step 2, a dialog will pop-up to show testing results and corresponding explanations identified by the testing and debugging engine.

a corresponding explanation generator on the basis of two algorithms DIRECTDEBUG and WIPEOUTR$_{FM}$. The explanation generator identifies corrective explanations to resolve anomalies of the feature model. Since DIRECTDEBUG determines exactly one diagnosis at a time, we combined this algorithm with a construction of the hitting set directed acyclic graph (HSDAG) [22] to determine the complete set of diagnoses.[7]

The input of the FMTESTING engine is a set of test cases selected from an already generated test suite or from a set of test case types when no test suites exist. The results of the test suite execution (i.e., anomalies and diagnoses) are returned and displayed in a dialog where they are represented in the form of a tree shape and grouped by test case type. The diagnoses that have been identified are displayed in ascending order in terms of impact on the structure of the feature model. This representation helps the user quickly identify problem areas concerning a certain anomaly type.

An example explanation is shown in Figure 3 (see the text in the blue rectangle). The tool detects that the feature *Female* is a *dead feature* and then generates three corrective explanations (diagnoses): *Diagnosis 1: [requires(Brake, Male)]*, *Diagnosis 2: [alternative(Frame, Female, Male, Step-through)]*, and *Diagnosis 3: [mandatory(BambooBike, Brake)]*. These explanations offer three ways of constraint/relationship deletion or adaptation to resolve the anomaly. Particularly, the dead feature *Female* can be resolved by adapting/deleting one of the following constraints/relationships: (1) the constraint *"requires"* between *Brake* and *Male*, (2) the relationship *"alternative"* between *Frame* and its sub-features (*Female, Male, Step-through*), and (3) the *"mandatory"* relationship between *BambooBike* and *Brake*.

[7] For further details of combining DIRECTDEBUG with a construction of HSDAG, we refer to [5, 7].

## 4 Limitations and Future Work

The tool has five limitations that need to be improved within the scope of future work. *First*, the current version of FMTESTING can exploit only six analysis operations to generate test cases. Supporting further analysis operations such as *core features* and *multiplicity bounds* is, therefore, necessary to make the tool more applicable to real-world feature models. The *second* limitation is the lack of *unintended behavior* checks. For instance, the check $BambooBike = f$ is needed to get rid of empty configurations. In the future version of our tool, we will integrate a mechanism to generate and specify *negative test cases* [16]. *Third*, a large number of generated test cases can cause inefficient testing and debugging operations. To address this, intelligent techniques such as test case redundancy detection [17], test case aggregation [14], and test case selection and prioritization might be essential to reduce the number of generated test cases. Additionally, the debugging performance can be improved by utilizing a parallelization approach proposed by Le et al. in [18]. *Fourth*, generated test cases need to be evaluated based on mutation testing [8] where mutation operations are generated based on insights into typical errors made when building feature models. *Finally*, the explanations of violated test cases are given as constraint sets, which are not directly related to the feature model's structural information and, consequently, could challenge stakeholders in comprehending the faults. A means to express explanations in a more user-friendly manner, such as a visualization directly inside the feature model, is therefore needed [12].

## 5 Conclusions

In this paper, we have introduced a FEATUREIDE plug-in named FMTESTING which provides a mechanism for the automated test case generation based on six basic analysis operations. Our tool can check whether a feature model shows one or more of these six anomalies and execute asynchronously test case validation as well as corresponding diagnosis. Although our tool already helps to improve feature model development and maintenance, further extensions, for example, in terms of analysis operations are still needed to provide enhanced services for real-world scenarios.

## Acknowledgements

The presented work has been developed in the FFG-funded project OPENSPACE ($FO999891127$).

## References

[1] D. Batory, 'Feature models, grammars, and propositional formulas', in *Software Product Lines*, eds., Henk Obbink and Klaus Pohl, pp. 7–20, Berlin, Heidelberg, (2005). Springer Berlin Heidelberg.

[2] D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank, 'Automated analysis in feature modelling and product configuration', in *Safe and Secure Software Reuse*, eds., J. Favaro and M. Morisio, pp. 160–175, Berlin, Heidelberg, (2013). Springer Berlin Heidelberg.

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

[4] K. Czarnecki, S. Helsen, and U. Eisenecker, 'Formalizing Cardinality-based Feature Models and Their Specialization', *Software Process: Improvement and Practice*, **10**(1), 7–29, (2005).

[5] A. Felfernig, D. F. Benavides Cuevas, J. Á. Galindo Duarte, and F. Reinfrank, 'Towards anomaly explanation in feature models', in *ConfWS-2013: 15th International Configuration Workshop (2013), p 117-124*. CEUR-WS, (2013).

[6] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, *Knowledge-Based Configuration: From Research to Business Cases*, Morgan Kaufmann, 2014.

[7] A. Felfernig, M. Schubert, and C. Zehentner, 'An efficient diagnosis algorithm for inconsistent constraint sets', *Artif. Intell. Eng. Des. Anal. Manuf.*, **26**(1), 53–62, (February 2012).

[8] J. M. Ferreira, S. R. Vergilio, and M. Quinaia, 'Software product line testing based on feature model mutation', *International Journal of Software Engineering and Knowledge Engineering*, **27**(05), 817–839, (2017).

[9] M. Hentze, T. Pett, T. Thüm, and I. Schaefer, 'Hyper explanations for feature-model defect analysis', in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, (2021). Association for Computing Machinery.

[10] U. Junker, 'QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems', in *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, p. 167–172. AAAI Press, (2004).

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, 'Feature-oriented Domain Analysis (FODA) Feasibility Study', Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, (1990).

[12] M. Kowal, S. Ananieva, and T. Thüm, 'Explaining anomalies in feature models', in *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, p. 132–143, New York, NY, USA, (2016). Association for Computing Machinery.

[13] D. Le, H. Lee, K. Kang, and L. Keun, 'Validating consistency between a feature model and its implementation', in *Safe and Secure Software Reuse*, eds., J. Favaro and M. Morisio, pp. 1–16, Berlin, Heidelberg, (2013). Springer Berlin Heidelberg.

[14] V. M. Le, A. Felfernig, and T. N. T. Tran, 'Test case aggregation for efficient feature model testing', in *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '22, p. 174–177, New York, NY, USA, (2022). Association for Computing Machinery.

[15] V. M. Le, A. Felfernig, T. N. T. Tran, M. Atas, M. Uta, D. Benavides, and J. Galindo, 'Directdebug: A software package for the automated testing and debugging of feature models', *Software Impacts*, **9**, 100085, (2021).

[16] V. M. Le, A. Felfernig, M. Uta, D. Benavides, J. Galindo, and T. N. T. Tran, 'Directdebug: Automated testing and debugging of feature models', in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 81–85, (2021).

[17] V. M. Le, A. Felfernig, M. Uta, T. N. T. Tran, and C. V. Silva, 'Wipeoutr: Automated redundancy detection for feature models', in *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, p. 164–169, New York, NY, USA, (2022). Association for Computing Machinery.

[18] V.-M. Le, C. V. Silva, A. Felfernig, D. Benavides, J. Galindo, and T. N. T. Tran, 'FASTDIAGP: An algorithm for parallelized direct diagnosis', *Proceedings of the AAAI Conference on Artificial Intelligence*, **37**(5), 6442–6449, (Jun. 2023).

[19] V. M. Le, T. N. T. Tran, and A. Felfernig, 'A conversion of feature models into an executable representation in microsoft excel', in *Intelligent Systems in Industrial Applications*, eds., M. Stettinger, G. Leitner, A. Felfernig, and Z. Ras, pp. 153–168, Cham, (2021). Springer International Publishing.

[20] M. Mendonca, A. Wąsowski, and K. Czarnecki, 'SAT-Based Analysis of Feature Models is Easy', in *SPLC'09*, pp. 231–240, USA, (2009).

[21] M. Nieke, J. Mauro, C. Seidl, T. Thüm, I. C. Yu, and F. Franzke, 'Anomaly Analyses for Feature-Model Evolution', in *17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, p. 188–201, New York, NY, USA, (2018). ACM.

[22] R. Reiter, 'A theory of diagnosis from first principles.', *Artif. Intell.*, **32**(1), 57–95, (1987).

[23] Francesca Rossi, Peter van Beek, and Toby Walsh, *Handbook of Constraint Programming*, Elsevier, 2006.

[24] G. R. A. Schmitt, C. Bettinger, and G. Rock, 'Glencoe–A Tool for Specification, Visualization and Formal Analysis of Product Lines', in *Proceedings of ISTE 25th International Conference on Transdisciplinary*

*Engineering*, volume 7 of *Advances in Transdisciplinary Engineering*, pp. 665–673, Amsterdam, (2018). IOS Press.

[25] N. Siegmund, S. Sobernig, and S. Apel, ‘Attributed Variability Models: Outside the Comfort Zone’, in *11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, p. 268–278. ACM, (2017).

[26] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, ‘Featureide: An extensible framework for feature-oriented software development’, *Sci. Comput. Program.*, **79**, 70–85, (jan 2014).

[27] K. Villela, A. Silva, T. Vale, and E. S. de Almeida, ‘A survey on software variability management approaches’, in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC ’14, p. 147–156, New York, NY, USA, (2014). Association for Computing Machinery.