Supporting the Development of Interdisciplinary Product Lines in the Manufacturing Domain

Matthias Kowal * **Sofia Ananieva** ** **Thomas Thüm** *
Ina Schaefer *

* *TU Braunschweig, Germany*
{m.kowal, t.thuem, i.schaefer}@tu-braunschweig.de
** *FZI Research Center for Information Technology, Germany*
ananieva@fzi.de

Abstract: The increasing demand for highly customizable manufacturing systems leads to an extreme number of possible machine variants. Feature models are often used to manage this system diversity. The development and maintenance of feature models are error-prone and time-consuming tasks, especially considering industrial-size models with thousands of features. In many cases, engineers might want to focus only on a few features relevant for their own domain. Additionally, each change may lead to anomalies in the feature model. In this paper, we present an approach to provide engineering support by giving user-friendly explanations for hidden dependencies and anomalies in feature models.

Keywords: Variability Modeling, Engineering, Machine Manufacturing, Variant and Version Management

1. INTRODUCTION

Customers have a rising demand for fully customizable products that can be tailored to their specific requirements (Pohl et al. (2005)). In return, manufacturers have to pay more attention to variability and its management to deal with the rising complexity introduced by the variant diversity, e.g., as in the automotive domain. Engineers tried to reduce this problem with the introduction of product lines several decades ago (Kang et al. (1990)). A product line is comprised of a set of related systems that share several commonalities and variabilities. For example, each car must have a radio making it a common feature, but a navigation system is only optional. The goal of product lines is to foster reuse potential, reduce maintenance effort and provide a better cost-efficiency (Czarnecki and Eisenecker (2000); Pohl et al. (2005)). Feature models are often used to express the variability as well as dependencies in a product line (Benavides et al. (2010)). Thousands of features and dependencies between these features are common in industrial-size feature models (Tartler et al. (2011)). Engineers often encounter two major problems while dealing with feature models.

First, product line development is an interdisciplinary process involving multiple developers from different domains, e.g., mechanical-, software-, and electrical engineering. Hence, the feature model contains information that may not be relevant for a certain domain or developer and can be hidden (Lettner et al. (2015); Feldmann et al. (2015); Ananieva et al. (2016)). It is crucial that no information is lost during such a process. Dependencies between features from different domains must still be respected and visible to the developer. In addition, hidden dependencies may occur in the partial feature model due to constraints across the complete product line. We refer to these hidden dependencies as implicit constraints and provide engineering support by giving explanations why they are present leading to more precise communication between different disciplines and help identify unintended interferences.

Second, the maintainability of a feature model decreases with its size (Mens and Demeyer (2008)). Evolution of a feature model due to changing requirements, the addition of new features or dependencies has an increasing possibility to introduce anomalies (Mens and Demeyer (2008)). Anomalies can be rather harmless such as redundancy meaning that semantic information is modeled in multiple ways which is usually not preferable (von der Maßen and Lichter (2004)). However, anomalies can also be severe such as dead features. It is not possible to select a dead feature for any variant of the system making it useless. In order to support developers in the removal of anomalies, they must be detected and explained to comprehend the cause why an anomaly has occurred in the feature model (Benavides et al. (2010); Kowal et al. (2016).

In this paper, we present an approach supporting engineers in both aspects: (1) depicting partial feature models with all implicit constraints as well as their explanation and (2) giving user-friendly explanations for anomalies, without introducing new concepts and notations for feature models or increasing the modeling workload.

2. CASE EXAMPLE: PICK AND PLACE UNIT

The running example is a product line form the automation engineering domain. The Pick and Place Unit (PPU) is a universal production demonstrator for studying evolution and variability (Legat et al. (2013)). It consists

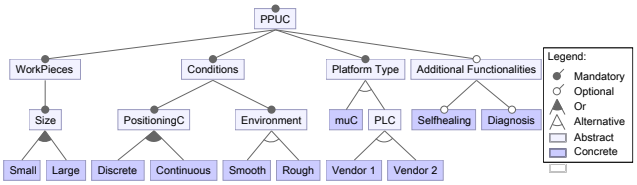


Fig. 1. Customer feature model (Legat et al. (2013))

of multiple variants and provides us with source code, UML diagrams and four feature models depicting different domains involved in the development of the PPU.

2.1 Feature Models

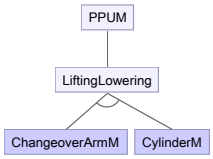
A feature model consists of a hierarchically arranged set of features and has typically a tree-like graphical representation. Fig. 1 shows a feature model of the PPU from the customer's point of view. Parent-child relationships are expressed using the following elements and semantics (see legend in Fig. 1 for the graphical representation (Kang et al. (1990); Czarnecki and Eisenecker (2000)):

- *Mandatory* – feature must be selected, if parent is,
- *Optional* – feature is optional,
- *Or* – one or more subfeatures can be selected,
- *Alternative* – only one subfeature can be selected.

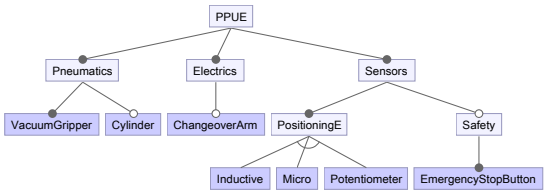
For example, the PPU can handle two types of workpieces simultaneously with *Small* and *Large*. The operating environment can either be *Rough* or *Smooth*, but not both at the same time. *Selfhealing* and *Diagnosis* are optional features. *Abstract* features are only used for structural aspects and do not contain realization artifacts, e.g., source code. Dependencies between features that are not part of a parent-child relationship are expressed with cross-tree constraints using propositional logic, $X \Rightarrow Y$. In case of the PPU feature model depicted in Fig. 1, cross-tree constraints are not present.

The development and maintenance of the PPU involves multiple disciplines with mechanical, software and electrical engineering. The customer feature model in Fig. 1 does not represent all disciplines in a sufficient manner which is why three additional engineering feature models are available (Legat et al. (2013); Feldmann et al. (2015)). Fig. 2 depicts the individual models describing the PPU in more detail for each domain. It is obvious that some similar features can be identified in multiple feature models, while other features are restricted to one model, since they are not relevant for other domains. The goal of separate feature models is to reduce the complexity for the engineers and let them focus on important parts for their domain. Several feature models in isolation are not sufficient to completely describe a product line. It is mandatory to express dependencies between the individual feature models as well. The developers of the PPU created a mapping matrix to express these global constraints connecting the customer feature model to the engineering models (Legat et al. (2013); Feldmann et al. (2015)).

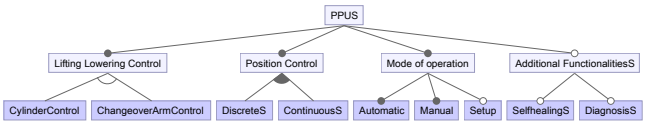
For example, the customer can select the small workpieces resulting in the selection of the features *ChangeoverArmM* in the mechanical, *ChangeoverArm* and *VacuumGripper* in the electrical and *ChangeoverArmControl* in the software model. Fig. 3 shows only an extract of the original matrix defined by Feldmann et al. (2015).



(a) Mechanical feature model



(b) Electrical feature model



(c) Software feature model

Fig. 2. Engineering feature models of the PPU (Legat et al. (2013); Feldmann et al. (2015))

			Developer's point of view					
			Mech.	Electrics/ Electronics			Software	
			Lifting/ Lowering	Pneum.	Electr./ Electron.	Sensors	Lifting/ Lowering Control	Position Control
view	Work pieces	Size: Small	Change-over arm	Vacuum Gripper	Change-over arm		Changeover arm Control	
		Size: Large	Change-over arm / Cylinder	(Cylinder) Vacuum Gripper			(Cylinder Control)	

Fig. 3. Extract from the mapping matrix (Feldmann et al. (2015))

2.2 Problem Statement

Engineers most likely maintain and develop only the feature model for their own domain. However, crucial information may be lost by considering just a portion of the product line, e.g. the dependencies expressed by the mapping matrix. The number of dependencies can easily add up to several thousands in industrial-size feature models making it unreasonable to present all of them, since only a small part is relevant to individual engineers. Additionally, the product line dependencies can produce implicit constraints in the considered partial feature model that are not visible at first. Regardless of the model part, each change may lead to an inconsistency. While the detection of such anomalies is well-researched, the actual explanation is often neglected or completely missing. We derive and present all relevant dependencies for an arbitrary partial feature model as well as explain the cause of an implicit constraint and all appearing anomalies. To maximize usability, we refrained from introducing new modeling concepts or notations while providing a fully functional open-source implementation in the FeatureIDE framework.

3. IMPLICIT CONSTRAINTS IN FEATURE MODELS

The definition of a mapping matrix is a successful first step to express dependencies between separate feature models (Feldmann et al. (2015)). Nevertheless, it has some drawbacks in terms of scalability and it is difficult to analyze. The connection of the individual engineering models

Fig. 4. Cross-tree constraints based on the mapping matrix (Legat et al. (2013); Feldmann et al. (2015)). Highlighted constraints are redundant.

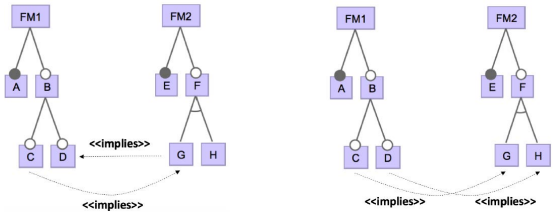


Fig. 5. Hidden implication and hidden exclusion

to each other is also missing, since the matrix expresses only dependencies from the customer model to the engineering models. In order to identify and explain implicit constraints of the PPU, it is necessary to transform the mapping matrix into a representation that is easier to analyze with common product line techniques. Feature models can be translated to propositional logic and most analysis techniques use this format (Batory (2005); Benavides et al. (2010)). Hence, we translated the matrix into propositional cross-tree constraints (cf. Section 2). The full list of constraints is depicted in Fig. 4. They represent our global dependencies that must be fulfilled for the complete product line. Highlighted constraints contain redundant information and, hence, it would be possible to remove them without introducing anomalies (Kowal et al. (2016)). The correctness of all cross-tree constraints was validated with the mechanical engineers developing the PPU.

3.1 Deriving Implicit Constraints

Implicit constraints can only occur in partial feature models, which are arbitrary submodels of the complete product line such as the engineering models. Implicit constraints always provide redundancy considering the complete product line, since the information is already available in the global cross-tree constraints as described by the mapping matrix or the propositional formulas in Fig. 4. The benefit of a partial feature model is a reduction in the visible complexity for the developer obtained by showing only necessary features and constraints.

Fig. 5 shows four feature models and presents two examples of implicit constraints. The first one describes a cycle of implications between three features in two separate feature models. Feature C from FM1 implies feature G from the FM2. However, feature G implies feature D from FM1. Both global cross-tree constraints result in a hidden dependency between feature C and D in FM1 with $C \Rightarrow D$. The second example presents another typical case of an implicit constraint. Feature C from FM1 implies feature G from the FM2. In addition, feature D from FM1 implies feature H from FM2. Through the alternative relationship between feature G and H in FM2, a hidden mutual exclusion occurs between feature C and D in FM1, i.e., $\neg(C \wedge D)$.

The PPU contains similar cases. For instance, consider the two workpiece types with *Small* and *Large*. Both features imply a different lifting/lowering mechanic in the mechanical model. Again, these features are part of an alternative relationship resulting in an implicit constraint of the form $(\neg Small \vee \neg Large)$. The PPU contains several more of such implicit constraints which we present in Section 3.2. Before we can explain the reason for an implicit constraint, it is a prerequisite to detect them.

The first challenge is the creation of a partial model based on a the complete product line, while preserving all dependencies. The removal of features must not change dependencies between features in the submodel. A state-of-the-art approach to eliminate features while maintaining dependencies between other features is feature model slicing (Acher et al. (2011)). Krieter et al. developed an efficient algorithm for feature model slicing in FeatureIDE (Krieter et al. (2016)). It has already been successfully applied in practice (Schröter et al. (2016)). Inputs to the algorithm are comprised of a feature model in conjunctive normal form (CNF) and a subset of features which are not part of the partial model. A CNF is a conjunction of clauses, a clause consists of a set of literals and a literal is a variable or its negation. The input feature model represents our complete product line and, hence, all four PPU models. After performing the feature model slicing, the algorithm returns a sliced feature model in CNF without the specified set of features while maintaining dependencies between features in the sliced model. We extend this method to derive the implicit constraints.

Given the PPU, the approach works as follows: First, the four engineering models are merged into one large model. Therefore, we create a new root feature, e.g., *PPU*. The old root features, namely *PPUC*, *PPUM*, *PPUE* and *PPUS*, become children of *PPU*. Second, we can select any feature in the large model, e.g., one of the old root features. The output of the slicing algorithm is compared to the large model in order to detect new constraints, which are then marked as implicit. For the customer model, we derive two implicit constraints with *(Diagnosis* $\vee \neg$ *Selfhealing)* and *(*$\neg$ *Small* $\vee \neg$ *Large)* (cf. Fig. 6). The sliced feature model can contain cross-tree constraints from the large feature model. This is the case if a global cross-tree constraint is defined solely with features of the partial model, e.g., the constraint only includes features from the customer model such as *Large* $\Rightarrow$ *Smooth*. This case does not occur in the PPU models, since the mapping matrix only defines dependencies between multiple models. It is necessary to distinguish between old global constraints and new implicit constraints. We decided to explicitly highlight implicit constraints with a red-colored frame in FeatureIDE (cf. Fig. 6). The hidden dependencies are now visible to the engineer.

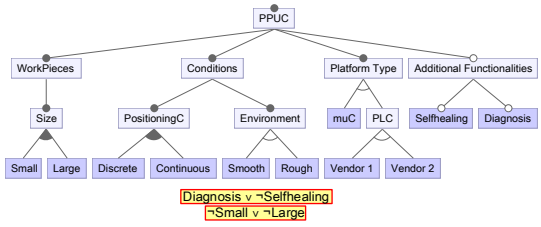


Fig. 6. Customer model with implicit constraints.

3.2 Explaining Implicit Constraints

Presenting the implicit constraints to the developer is hardly a sufficient solution, since it is tedious to manually identify the cause of such dependencies in large feature models. We provide an additional step to explain why an implicit constraint holds. The algorithm is based on boolean constraint propagation (BCP). Recalling the implicit constraint for the hidden implication $C \Rightarrow D$ in Fig. 5, we generate the following explanation: *Constraint $C \Rightarrow D$ is implicit, because: $G \Rightarrow D$ is a constraint and $C \Rightarrow G$ is a constraint.* The engineer knows exactly why an implicit constraint exists in the partial feature model. The causes for implicit constraints in the PPU are already much more complex and difficult to identify. Before providing explanations for the PPU, we describe the concept of our explanation algorithm in more detail.

BCP functions as our core algorithm and uses boolean formulas. They use operators such as OR, AND, and NOT to connect variables. Reasoning about boolean formulas is achieved by propagating assumed truth values for boolean variables. A brief example is shown below:

(1) $A \wedge B = C$: If C = *true*, then A and B must be true.
(2) $A \vee B = C$: If A = *false* $\wedge$ C = *true*, then B is true.

The input to BCP is defined by a set of variables and a formula in conjunctive normal form (CNF). The truth value of the variables is specified by a three-value logic with (true, false, unknown). Every clause in the CNF is assigned to one of the following types:

- **Satisfied**: At least one literal is true.
- **Violated**: All literals are false.
- **Unit-Open**: One literal is unknown while the remaining literals are false.
- **Non Unit-Open**: More than one literal is unknown, the rest is false.

A unit-open clause can be satisfied by setting its unknown literal to true. Using the clause $\neg X \vee Y \vee Z$, we demonstrate the different types:

- If X is false, the clause is **satisfied**.
- If X is true, Y is false and Z is false, the clause is **violated**.
- If X is true, Y is false and Z is unknown, the clause is **unit-open**. Z is derived as true.
- If X is true and Y and Z are unknown, the clause is **non unit-open**.

BCP performs the following steps: It is invoked with initial truth value assignments, which are called *premises*. Based on the premises, BCP deduces consequences for other literals and propagates them. The selection of truth values for the premises is a core step in our explanation algorithm. In a first iteration, BCP pushes all unit-open clauses in the CNF on a stack. Next, the last unit-open clause is removed from the stack and BCP deduces the truth value of the unknown literal. This process may result in new unit-open clauses that are again pushed to the stack. BCP continues these steps until a contradiction occurs during constraint propagation. BCP reports the violation and terminates. BCP saves information in a set of 3-tuples with

$$\{conclusion, reason, \{antecedents\}\}$$

for every deduced truth value assignment. A *conclusion* represents an inferred value, while the *reason* contains the unit-open clause responsible for the derived assignment. *Antecedents* are the predecessors of the considered clause.

Table 1. Explaining the implicit constraint $C \Rightarrow D$.

ID	Con.	Reason	AC	Stack
#1	D=0	premise		
#2	C=1	premise		$(\neg G \vee D)$, $(\neg C \vee B)$, $(\neg C \vee G)$
#3	G=1	$(\neg C \vee G)$	#2	$(\neg G \vee D)$!, $(\neg C \vee B)$, $(\neg G \vee F)$, $(\neg G \vee \neg H)$
Violated Clause: $(\neg G \vee D)$				
Explanation: *Constraint $C \Rightarrow D$ is implicit, because: $G \Rightarrow D$ is a constraint(violated clause) and $C \Rightarrow G$ is a constraint (#3).*				

Hence, an application of BCP to explain implicit constraints uses the CNF created from the complete product line, e.g., the combination of all engineering models. Using only the sliced model is not sufficient, since an explanation always includes other partial feature models as well. A truth value assignment for the features from the implicit constraint making it non-satisfiable. In that case, we ensure that BCP generates a contradiction and use the stored reasons for this violation to build our explanation. However, multiple assignments can lead to a non-satisfiable constraint. Every combination may result in a different explanation giving us only a part of the solution. A union of all partial explanations produces the final result which is able to explain an implicit constraint. Duplicate parts are ignored to reduce the length. We additionally store information about the tracing of each literal to the feature model during the CNF creation. Every literal belongs to a clause which either originates from the feature hierarchy or a global cross-tree constraint. This tracing enables us to provide user-friendly explanations to the developer, since pure CNF clauses are not directly visible in the model.

Consider the two (left) feature models in Fig. 5 resulting in the implicit constraint $C \Rightarrow D$. Table 1 presents the BCP process in order to explain $C \Rightarrow D$. We pass the combined feature model in CNF and premises to BCP. In this case, we have only one truth value assignment leading to a non-satisfiable clause with $D = false$ and $C = true$. BCP collects unit-open clauses from the CNF and pushes them on the stack. We refrain from presenting the complete CNF due to its length. BCP derives G to be true and updates all respective literals in the CNF with the truth value resulting in a violated clause $\neg G \vee D$. An explanation is generated by reporting the reasons: first, we take the violated clause and, second, we traverse the reasons for conclusions backwards to the premises. Initial value assumptions do not need to be reported shortening the explanation.

3.3 Explaining Anomalies

Although, we achieved a significant reduction of the complexity by considering only partial feature models and explaining hidden dependencies, engineers can still introduce problematic inconsistencies in the feature model during development and maintenance. Again, as a first step the detection of such anomalies is mandatory before we can actually explain the cause. The first anomaly that we tackle is already the most severe. If an engineer encounters a void feature model, it is not possible to derive any variant of the product line. For example in Fig. 1, adding the constraint

$WorkPieces \wedge \neg Conditions$ would result in a void feature model, since two core feature exclude each other. Features are defined as dead, if they can never be selected in any variant of the product line von der Maßen and Lichter (2004). Hence, they have no purpose at all. For example, in Fig. 1, adding the constraint $WorkPieces \Rightarrow Smooth$ makes the feature *Rough* dead. An alternative group allows only the selection of one feature at a time and *Smooth* is in all variants due to the implication by a core feature. This anomaly is problematic as artifacts could be developed but never used. A feature is defined as false-optional, if selecting its parent makes the feature itself selected as well, although it is defined as optional and not mandatory. The constraint $AdditionalFunctionalities \Rightarrow Diagnosis$ makes the feature *Diagnosis* false-optional. As last anomaly type, we consider redundant constraints. Fig. 4 already shows six redundant constraints for the PPU. Regardless of the anomaly type, the standard detection method is performed by using a satisfiability solver. The necessary calls are omitted at this point and presented elsewhere (Kowal et al. (2016)).

We were only able to detect redundant constraints in the PPU feature models. However, even in this small example, it can be hard to determine the cause for a redundancy and it gets increasingly more difficult with larger feature models. Our approach is able to generate explanations for all anomaly types by the same means. The basic principle of BCP, as presented in section 3.2, remains and we only need to adapt the premises. For example, a dead feature cannot be present in any variant and by setting the truth value of the respective variable to *true* as a premise, BCP will always produce a contradiction at some point giving us the explanation, since the CNF is not satisfiable.

3.4 Implementation

We implemented our approach in the open-source framework FeatureIDE and it is part since release, FeatureIDE 3.1.0 [1]. After modeling a feature model in FeatureIDE, our approach can be executed by doing a right-click on a feature and selecting *Show Hidden Dependencies of Submodel* in the context menu. A new page opens containing the partial feature model with the selected feature as root. Below the root, all features appear in the same way as in the complete feature model. Global cross-tree constraints mandatory for the partial model are depicted below the feature hierarchy. Implicit constraints can be distinguished by a surrounding red border and are marked as redundant as well. Explanations for implicit constraints and all other anomaly types are instantly visible if the engineer moves the cursor above the respective feature or constraint.

4. APPLICATION TO THE PPU

Returning to our running example, our approach derives four implicit constraints with:

(1) $Diagnosis \vee \neg Selfhealing$ (Customer Model)
(2) $\neg Small \vee \neg Large$ (Customer Model)
(3) $Cylinder \vee ChangeoverArm$ (Electrical Model)
(4) $DiagnosisS \vee \neg SelfhealingS$ (Software Model)

Fig. 7 presents explanations for these constraints computed by our extension of FeatureIDE. The first one is a hidden implication in the customer feature model because

[1] https://github.com/FeatureIDE/FeatureIDE

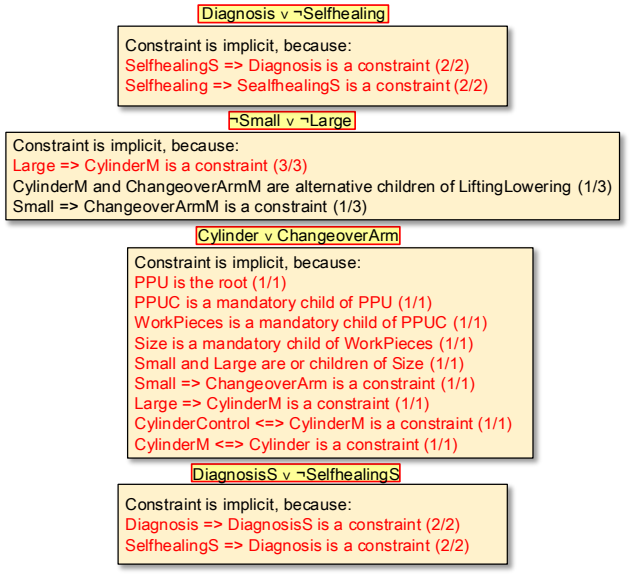


Fig. 7. Explaining implicit constraints of the PPU

of two global cross-tree constraints. A similar dependency is detected for the software model shown in the fourth constraint. The explanations reveal that almost identical features are involved in both cases. Recalling Fig. 3, the explanations can be mapped to the last three cross-tree constraints resulting in two implicit dependencies. A hidden exclusion is expressed in the second constraint. Two features in the customer model imply different features in an alternative group of the mechanical model. The third constraint gives the most complex explanation in our running example. Considering only the electrical feature model without implicit constraints, it is possible to avoid the selection of the *Cylinder* and *ChangeoverArm* at all. However, the derived implicit constraint forbids this configuration, since we have to select at least one of both features. We observe a transitive chain in the customer feature tree from the root *PPU* of the complete model to the two workpieces *Small* and *Large*. Again, both imply different features even from different feature models. And, due to bijections, an implicit dependency occurs in the electrical feature model.

Readers may wonder about two additional points visible in Fig. 7. First, we explain the numbers behind the individual explanation parts. Considering the second constraint $\neg Small \vee \neg Large$, BCP generates three separate explanations during the search for a short explanation Kowal et al. (2016). The first part, namely $Large \Rightarrow CylinderM$, is present in all three explanations, hence it has a (3/3) at the end. The other two parts are only present in one explanation resulting in (1/3). This aspect is further emphasized by the colors ranging from red to black. Red indicates that the part is present in all explanations, while the color gradually changes to black for a part being present in just one explanation. We enhanced BCP with further improvements considering the explanation length. The details are available in our previous work Kowal et al. (2016). The explanations of the redundant constraints in the PPU are not presented in this paper due to space reasons. The concept and tooltip are analogue to the explanation of implicit constraints in Fig. 7. A far more detailed and large-scale evaluation is available elsewhere (Kowal et al. (2016); Ananieva et al. (2016))

5. RELATED WORK

A considerate amount of research has been conducted on feature modeling. Some approaches already provide support for analysis techniques such as anomaly detection (Benavides et al. (2010, 2013)), but only a few respect dependencies between different partial models (Schröter et al. (2013); Lettner et al. (2015)) and even less actually explain the dependencies or anomalies (Benavides et al. (2010)). We are the first to make implicit constraints explicitly visible for the engineer during the modeling process. An orthogonal concept to reduce complexity for engineers are feature model views hiding the undesired features. Views are often connected to the configuration process and not the actual development or maintenance of a feature model (Schroeter et al. (2012)). Lettner et al. (2015) added functionality in FeatureIDE to support modeling at different abstraction levels and dependencies between them. Considering the concrete explanation process, we are most closely related to work done by Batory (2005). Batory adapted a BCP algorithm to support developers in the configuration process of a variant using a feature model. The open-source implementation is available in GUIDSL and gives feedback in terms of why a specific feature cannot be selected. No support for implicit or redundant constraints is available and the explanations are presented in terms of propositional formulas making it difficult to understand for developers. Explaining dead or false-optional features is also presented in other work (Trinidad (2012); Benavides et al. (2010)). However, no approach considers the explanation of implicit constraints, shows the scalability for industrial-size feature models and also provides an open-source implementation. A detailed comparison of our proposed approach and existing ones can be found in our previous work (Kowal et al. (2016); Ananieva et al. (2016)).

6. CONCLUSION

We have presented an approach for deriving and explaining implicit constraints as well as anomalies in feature models. The applicability was shown with a product line from the automation engineering domain and an open-source implementation is available in FeatureIDE. Some aspects are left for future work. An important task is to enable edit operations in the partial feature model and reflect them back to the complete one in order to fully support maintenance. A small user study with the mechanical engineers responsible for the PPU is planned to get qualitative feedback of how to improve explanations even further.

ACKNOWLEDGEMENTS

This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution.

REFERENCES

Acher, M., Collet, P., Lahire, P., and France, R.B. (2011). Slicing feature models. ASE '11, 424–427. IEEE Computer Society, Washington, DC, USA.

Ananieva, S., Kowal, M., Thüm, T., and Schaefer, I. (2016). Implicit Constraints in Partial Feature Models. Proc. Int'l Workshop Feature-Oriented Software Development (FOSD). ACM, NY.

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, 7–20.

Benavides, D., Felfernig, A., Galindo, J.A., and Reinfrank, F. (2013). *Automated Analysis in Feature Modelling and Product Configuration*. Proc. Int'l Conf. Software Reuse (ICSR). Springer, Berlin, Heidelberg.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6).

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*.

Feldmann, S., Legat, C., and Vogel-Heuser, B. (2015). Engineering Support in the Machine Manufacturing Domain through Interdisciplinary Product Lines: An Applicability Analysis. *IFAC-PapersOnLine*, 48(3).

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SE Institute.

Kowal, M., Ananieva, S., and Thüm, T. (2016). Explaining Anomalies in Feature Models. Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM, NY.

Krieter, S., Schröter, R., Thüm, T., Fenske, W., and Saake, G. (2016). Comparing algorithms for efficient feature-model slicing. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA.

Legat, C., Folmer, J., and Vogel-Heuser, B. (2013). Evolution in industrial plant automation: A case study. In *39th Annual Conference of the IEEE Industrial Electronics Society (IECON)*.

Lettner, D., Eder, K., Grünbacher, P., and Prähofer, H. (2015). Feature modeling of two large-scale industrial software systems: Experiences and lessons learned. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*.

Mens, T. and Demeyer, S. (eds.) (2008). *Software Evolution*. Springer-Verlag, Berlin Heidelberg.

Pohl, K., Böckle, G., and van der Linden, F.J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*.

Schroeter, J., Lochau, M., and Winkelmann, T. (2012). Multi-Perspectives on Feature Models. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, 252–268.

Schröter, R., Krieter, S., Tüm, T., Benduhn, F., and Saake, G. (2016). Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 667–678. ACM, New York, NY, USA.

Schröter, R., Thüm, T., Siegmund, N., and Saake, G. (2013). Automated Analysis of Dependent Feature Models. In *VaMoS*, 9:1–9:5. doi:10.1145/2430502.2430515.

Tartler, R., Lohmann, D., Sincero, J., and Schröder-Preikschat, W. (2011). Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the Conference on Computer systems*, 47–60. ACM.

Trinidad, P. (2012). *Automating the Analysis of Stateful Feature Models*. Ph.D. thesis, University of Seville.

von der Maßen, T. and Lichter, H. (2004). Deficiencies in Feature Models. In *Proceedings of the Workshop on Software Variability Management for Product Derivation-Towards Tool Support*.