Implicit Constraints in Partial Feature Models

Sofia Ananieva
FZI Research Center for Information Technology, Germany
ananieva@fzi.de

Matthias Kowal Thomas Thüm
Ina Schaefer
TU Braunschweig, Germany
{m.kowal, t.thuem, i.schaefer}@tu-bs.de

Abstract

Developing and maintaining a feature model is a tedious process and gets increasingly difficult with regard to large product lines consisting of thousands of features and constraints. In addition, these large-scale feature models typically involve several stakeholders from different domains during development and maintenance. We aim at supporting such stakeholders by deriving and explaining implicit constraints for partial feature models. A partial feature model can either be a submodel of a feature model representing the full product line or a specific feature model in a set of interrelated models. For every implicit constraint, we generate an explanation exposing which other model parts and constraints interfere with the partial model of interest. Thus, stakeholders are only confronted with a small part of the feature model reducing the complexity while preserving the necessary information about dependencies. Our approach is implemented in the open-source framework FeatureIDE.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management—Software configuration management; D.2.13 [*Software Engineering*]: Reusable Software

Keywords Configurable Software, Feature Model

1. Introduction

Customers have a rising demand for highly configurable products that can be exactly tailored to their individual requirements [27]. In reverse, manufacturers have to pay more attention to variability and its management for their products to cope with these demands. A famous example is the automotive domain in which almost every car leaving the factory is unique. Software plays an important role not only in today's vehicles, but in many everyday consumer products. Hence software systems also have to deal with variability. The upside of variability is to adapt systems to different kinds of environments and requirements. The downside is a large variant space, which has to be managed, maintained, and tested [21]. With the introduction of product lines several decades ago, engineers tried to mitigate this problem [8, 16]. A product line consists of a set of related systems that share some commonalities and variabilities. For example, every car must have a radio making it a common feature, but a navigation system is only optional. Product lines enhance the reuse potential enabling the generation of a specific variant. They are known to reduce time-to-market, simplify maintenance, and have a better cost-efficiency compared to classical software development [9, 27].

Without restricting the large variant space, a product line can contain feature combinations that are not possible or useful. Kang et al. [16] introduced feature models to explicitly model these restrictions. Feature models are widely used to express the intended variability of a product line [6]. Industrial-size feature models tend to have thousands of features and cross-tree constraints [33]. However, maintainability of a feature model decreases with its size posing a major problem for developers [25]. A possible countermeasure is to use feature model views showing only specific parts of the complete model. Thus, developers can focus on parts most relevant to them reducing the overall complexity [7, 14].

In most cases, the development of a product line is an interdisciplinary undertaking involving countless developers from domains such as mechanical-, electrical-, and software engineering. As a result, the feature model can contain information that may not be relevant for individual developers and can be hidden or split up using concepts such as different views on feature models, feature model decomposition and multi software product lines [1, 11, 20, 29, 30]. It is mandatory that no crucial information is lost using these methods, e.g., dependencies between features of the software domain and the mechanical domain are still respected and visible to the developer in his partial feature model or feature model view. Furthermore, by considering only a part of the model, the developer may be confronted with hidden dependencies. The constraints of the complete feature model can result in restrictions for the partial model that are not directly visible and have to be derived for the developer. We refer to such de-

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

FOSD'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4647-4/16/10...
http://dx.doi.org/10.1145/3001867.3001870

pendencies as implicit constraints. An explanation for these implicit constraints is necessary to support the communication between the different disciplines and to find out which model parts interfere in an unintended way.

Existing work focuses on the introduction of multiple views or models following the separation of concerns principle and preserving consistency in case of feature model evolution for configuration purposes [11, 14, 29, 38]. The actual derivation and additional explanation of implicit constraints is neglected or completely missing [20, 24]. Hence, we propose an approach that provides developers with both aspects without introducing new concepts and notations for feature models or increasing the modeling workload for developers. The approach works on any submodel of a feature model as well as on interrelated feature models. To the best of our knowledge, the derivation of implicit constraints for such models has never been proposed before. In summary, we provide the following contributions:

1. We derive and explain implicit constraints for a partial feature model and its given context.
2. We provide an open-source implementation in FeatureIDE and evaluate the frequency and nature of implicit constraints.

2. Case Study: Pick and Place Unit

A product line from the automation engineering domain, namely the Pick and Place Unit (PPU), serves as running example throughout this paper. The PPU is a universal production demonstrator for studying variability and evolution [19]. Overall 15 variants are documented in detail with different UML model types and source code [37]. In addition, a feature model is available representing the product line from the customer's point of view (cf. Fig. 1).

2.1 Feature Models

A feature model typically has a tree-like graphical representation depicting the hierarchically arranged set of features. A feature model of the PPU is shown in Fig. 1. Relationships in the feature model regarding parent and child features are expressed with the common notation of *mandatory*, *optional* features and *or*-, *alternative* groups and their underlying semantics (cf. legend in Fig. 1 for the graphical representation) [9, 16]. *Abstract* features do not contain realization artifacts and are only used for structural purposes [33]. For instance, the PPU can process up to two different types of workpieces simultaneously with size *Small* and *Large*. The operating environment can either be *Smooth* or *Rough*, but not both at the same time. The additional functionalities of *Selfhealing* and *Diagnosis* are optional. Relationships between features that are not related by a parent-child relationship can be expressed using cross-tree constraints written in propositional logic, e.g., $A \Rightarrow B$. However, the feature model of the PPU in Fig. 1 does not provide us with any cross-tree constraints at the moment.

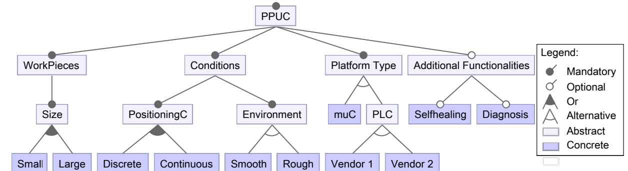
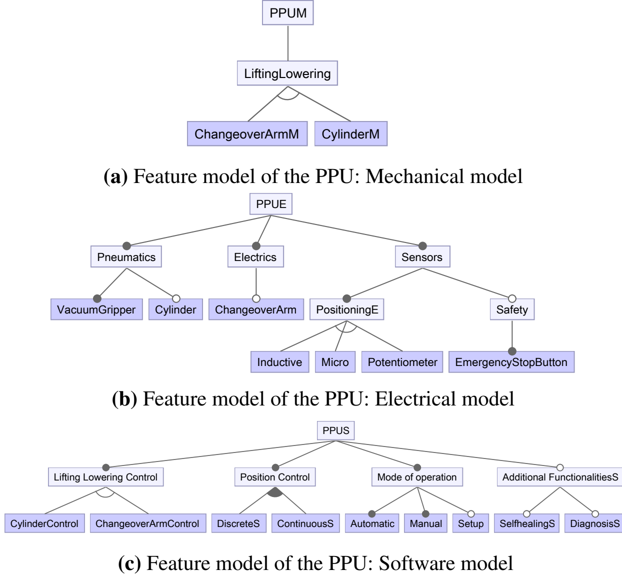


Figure 1: Feature model of the PPU: Customer model [19]

2.2 Interrelated Feature Models

The development and maintenance of the PPU involves multiple domains with software, mechanical and electrical engineering that are not sufficiently represented in the previously described feature model. The model in Fig. 1 is solely for configuration purposes by a potential customer or plant manufacturer. As a result, three additional engineering feature models are available for the PPU covering the different domains [11, 19]. The individual models are depicted in Fig. 2. It is not surprising that some features are similar to the customer feature model such as the positioning (present in the electrical and software model) or additional functionalities that are also visible in the software feature model. However, some features are only present in the engineering models, e.g., *Safety*. Fig. 2a consists of two alternative features describing different mechanical mechanism to lift workpieces. These features are also reflected in Fig. 2b as child features of *Pneumatics* and *Electrics*. The PPU can be equipped with several sensors, namely *Inductive*, *Micro* or *Potentiometer*, to ascertain the positions of workpieces and the crane. For the PPU, an *EmergencyStopButton* is not mandatory. Depending on the selected sensors, the positioning can work in a discrete or continuous manner (cf. Fig. 2c). The PPU can always be controlled manually or operate fully automatic. Finally, the PPU can have a diagnose function and heals itself if a problem is encountered. It is not relevant for our work, to what extent the healing is possible. The goal of separate feature models is to reduce the complexity for the developers and let them focus on important assets for their domain [1, 31].

Several feature models alone do not provide an ultimate solution to reduce complexity, since such an approach is missing a very important aspect. The individual feature models have to be connected to each other in order to be able to fully model the product line. Hence, we need a mechanism to express dependencies between the separate feature models. The developers of the PPU models proposed a mapping matrix between the customer feature model and the engineering models to express such global constraints [11, 19]. For instance, a customer can select the small workpiece type resulting in the selection of the features *ChangeoverArmM* in the mechanics, *VacuumGripper* and *ChangeoverArm* in the electrics and *ChangeoverArmControl* in the software. Several of these dependencies are defined by the developers [11]. Hence, the mapping matrix sets the customer's model into context of the engineering models. Fig. 3 shows an extract of the original matrix. The columns 1-4 (left-side)



(a) Feature model of the PPU: Mechanical model

(b) Feature model of the PPU: Electrical model

(c) Feature model of the PPU: Software model

Figure 2: Engineering feature models of the PPU [11, 19]

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

Figure 3: Extract from the mapping matrix [11].

represent features from the customer feature model. The three top-level rows belong to the engineering models. The matrix is interpreted as follows: Selecting the small work-piece size in the customer model requires changeover arm in the mechanics, vacuum gripper in the electrics and so on. The connection of the individual engineering models to each other is missing in the mapping. Although the definition is rather informal and the matrix does not scale very well, it is a first step contributing to a satisfying solution for interrelated feature models.

A partial feature model can either be a model representing one domain, e.g., the customer model for the PPU or a submodel in a feature model, e.g., the *WorkPieces* represents a submodel with all of its subfeatures. Most automated analysis methods for feature models rely on analyzing a single model. If multiple models exist as in the case of the PPU, they are merged into one model. A common solution is to create a new root feature and rearrange the separate feature models of each domain below this new root feature also resulting in one large feature model for the product line. Our approach also uses the last technique at least for analyses purposes to manage separate feature models.

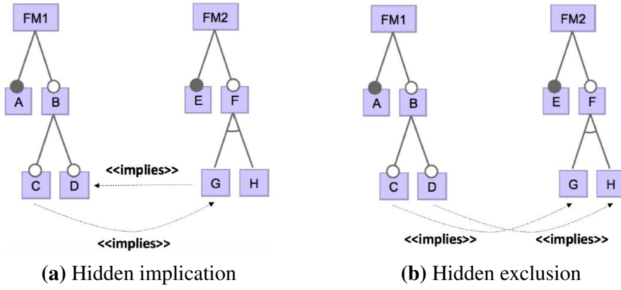
In order to reason about the implicit constraints of the PPU, it is first necessary to convert the mapping matrix in a representation that is easy to analyze. Feature models can be translated to propositional logic and many automated analysis techniques already use this format [4, 6, 22]. In a first step, we translated the matrix into propositional cross-tree constraints. The complete list of constraints is depicted in Fig. 4. They represent our global constraints that must be valid for the complete product line. Highlighted constraints are redundant and provide no additional information, hence it would be possible to remove them without introducing anomalies or false configuration possibilities [12, 17]. We validated the correctness of all cross-tree constraints with the actual developers of the PPU. As previously mentioned, 15 variants of the PPU are described in detail. The actual feature models permit a significant larger number of possible variants, since the mapping matrix is not restrictive enough.

Figure 4: Cross-tree constraints based on the mapping matrix [11, 19]. Highlighted constraints are redundant.

3. Deriving Implicit Constraints

We call a feature model representing the full product line a *complete* feature model. This includes separate feature models for different domains, since they can be merged into one large model under a new root feature. Most of the automated analysis techniques including our own operate at this level [6]. A partial feature model is an arbitrary submodel in this complete model. Implicit constraints occur if we only consider such partial feature models of the product line. Implicit constraints always provide redundancy considering the complete feature model, since the information is already available in the cross-tree constraints of the complete feature model, which we refer to as global constraints. The idea of a partial feature model is to reduce the visible complexity for the developer. Showing globally defined cross-tree constraints does not help if the features are unknown and the number can easily add up to several thousands. Stakeholders need to focus on constraints relevant for the partial feature model that they observe at the moment. In addition, it is mandatory that developers be aware of such hidden dependencies in order to prevent the introduction of inconsistencies or errors during feature model maintenance and development.

Fig. 5 shows four feature models and two cases resulting in implicit constraints. The first example describes a cycle of implications between two feature models FM1, FM2 (cf. Fig. 5a). Feature C from FM1 implies feature G from the FM2. Feature G, on its part, implies feature D from FM1. Both global cross-tree constraints result in a hidden implication between Feature C and D in FM1, namely $C \Rightarrow D$. The second example presents another use case of an implicit constraint in FM1 (cf. Fig. 5b). Feature C from FM1 implies feature G from the FM2. Feature D from FM1 implies feature



(a) Hidden implication **(b)** Hidden exclusion

Figure 5: Two exemplary implicit constraints

H from FM2. An alternative relationship between feature G and H in FM2 results in a hidden mutual exclusion between feature C and D in FM1, i.e., $\neg(C \wedge D)$. It is possible to find similar cases for our running example of the PPU. For instance, consider the two types of workpieces with *Small* and *Large*. Both features imply a different lifting/lowering mechanic in the mechanical feature model. However, these features are part of an alternative relationship resulting in an implicit constraint in the form of $(\neg Small \vee \neg Large)$. The PPU contains several more implicit constraints which we explain in Section 4. To assure the quality of partial models, the automated detection of hidden dependencies has become an important field of research [18, 20, 30]. For our approach, the detection is a necessary prerequisite to actually explain why implicit constraints occur. To the best of our knowledge, the explanation part is often neglected in literature [6].

The first challenge is the creation of a partial model based on a complete one, while preserving all dependencies. The elimination of features must not change dependencies between features in the submodel. A state-of-the-art approach to remove features while maintaining dependencies between other features is feature model slicing [2]. Feature model slicing can be applied for different scenarios, i.e., feature model evolution, removing abstract features and for the decomposition of feature models [2, 35]. Krieter et al. present an efficient algorithm for feature model slicing in FeatureIDE [18]. The algorithm has already been successfully applied in practice [30]. Inputs to the algorithm comprise a feature model in conjunctive normal form (CNF) and a subset of features which shall not appear in the partial feature 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 a complete feature model before the slicing operation. 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. The core of the slicing algorithm is logical resolution. The main idea of logical resolution involves the construction of a new clause, which represents a relationship between features of the sliced feature model. We refer to this clause as *resolvent*, represented by a new cross-tree constraint of the sliced feature model. The construction of a *resolvent* requires two clauses such that the first clause contains the literal to remove in its positive form and the second clause contains the literal to remove in its negated form. Krieter et al. derive the *resolvent* by combining the two clauses and removing the respective literal. The *resolvent*, on its part, represents a transitive relationship between the two clauses. We consider a new *resolvent* as an implicit constraint. A more detailed explanation of the slicing algorithm is available elsewhere [18].

We adapt this feature model slicing algorithm in our approach. A specific feature, that acts as the root of the partial feature model, is selected. The slicing algorithm then removes all features that are not part of the subtree of the new root feature. Given the PPU, the process works as follows: First, the four feature models are merged into one large model. Therefore, we create a new root feature, e.g., *PPU* and the old root features, namely *PPUC*, *PPUM*, *PPUE* and *PPUS*, act as children of *PPU*. Second, we can select any feature in the complete model, e.g., one of the old root features and the output of the slicing algorithm is compared to the complete model. All new constraints are then marked as implicit. For the customer model, we compute two implicit constraints with *(Diagnosis* $\vee$ *¬ Selfhealing)* and *(¬ Small* $\vee$ *¬ Large)*. The sliced feature model might contain cross-tree constraints from the complete feature model. This is the case if a global cross-tree constraint is defined solely with features of the partial/sliced model, e.g., the constraint only includes features from the customer model such as $Small \Rightarrow Discrete$. In order to detect implicit cross-tree constraints, it is necessary to differentiate between old global constraints and new implicit constraints by comparing cross-tree constraints of both feature models. As for the PPU, we have already shown the global constraints in Fig. 4. Since the individual feature models of the PPU have no constraints at all, it is obvious that no global constraint occurs after the slicing process for the customer model. The hidden dependencies are now visible to the developer. At this point, we are still missing the cause why these constraints occur and what the connection to the other feature model parts is.

4. Explaining Implicit Constraints

Deriving and presenting the implicit constraints to the developer is hardly a sufficient solution, since it is laborious to manually identify the cause of such dependencies in large feature models. Hence, we propose an additional step to explain why an implicit constraint holds. The explanation algorithm is adapted from our previous work [17] and is originally used to explain anomalies in feature models, e.g. dead and false-optional features, redundant constraints or void feature models. It is a boolean constraint propagation (BCP) algorithm functioning as inference engine for a logic truth maintenance system. Recalling the implicit constraint for the hidden implication $C \Rightarrow D$ from Fig. 5a, we can generate the following explanation with: *Constraint* $C \Rightarrow D$ *is implicit, because:* $G \Rightarrow D$ *is a constraint and* $C \Rightarrow G$ *is a con-*

straint. Hence, the developer knows exactly why an implicit constraint exists in the partial feature model. The reasons for implicit constraints in the PPU are already more complex and difficult to identify manually. However, before proposing explanations for our running example, we describe the basic concept of our explanation algorithm.

Boolean Constraint Propagation BCP functions as our core principle and uses boolean constraints that are represented by means of boolean formulas. They use connectives such as OR, AND, and NOT to combine variables. Reasoning about boolean constraints is achieved by propagating known 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 must be true.

BCP is also referred to as *Unit Resolution* and uses such propagation techniques to conclude inferences [10]. The input to BCP is specified by a set of variables and a formula in conjunctive normal form (CNF). The truth value of the variables is defined by a three-value logic with (true, false, unknown). Using this specification, 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

Hence, a unit-open clause can be satisfied by setting its unknown literal to true.

Example Regarding the clause $\neg X \vee Y \vee Z$, the different types are demonstrated:

- 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**.

The fundamental idea of BCP is depicted in Fig. 6. The algorithm is invoked with initial truth value assignments, the *premises*. Given the premises, BCP propagates the resulting consequences for the other literals. Selecting truth values for the premises is a crucial step and shapes the core of our explanation algorithm. During the first iteration, BCP pushes all unit-open clauses it finds in the CNF on a stack. Afterwards, a unit-open clause is removed from the stack and BCP deduces the truth value of the unknown literal. Again, BCP searches the CNF for new unit-open clauses and pushes them on the stack. The iterative process is repeated until a violation is encountered during constraint propagation. BCP reports the contradiction and terminates. Finding the violation is crucial to explain an implicit constraint in the partial feature model.

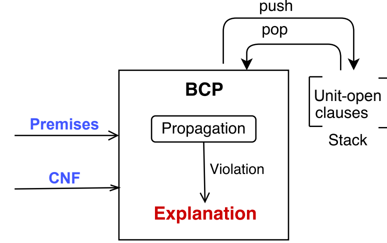


Figure 6: Overview of Boolean Constraint Propagation

BCP stores information in a set of 3-tuples with

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

for every deduced truth value assignment, i.e., the *conclusion* represents an inferred value, while the *reason* contains the unit-open clause leading to the derived assignment. *Antecedents* hold the predecessors of the considered clause. Hence, all variables in the clause that were previously referenced and for which BCP also has a 3-tuple.

Example Consider the formula of a feature model $(X \Rightarrow Y) \wedge (Y \Rightarrow \neg X)$, which is transformed to a CNF $(\neg X \vee Y) \wedge (\neg Y \vee \neg X)$. BCP sets $X = true$ and maintains its reason as *premise*, as presented in Table 1. A premise does not have *antecedents*. BCP pushes unit-open clauses from the CNF on the stack. After examining $(\neg Y \vee \neg X)$, BCP infers $Y = false$, saves its unit-open clause as reason and refers to variable X as its value was referenced. The BCP algorithm discovers the violated clause $(\neg X \vee Y)$ and reports the contradiction which we use to generate an explanation.

ID	Con.	Reason	AC	Stack
#1	X=1	premise		$(\neg X \vee Y), (\neg Y \vee \neg X)$
#2	Y=0	$(\neg Y \vee \neg X)$	#1	$(\neg X \vee Y)$
#3		$(\neg X \vee Y)$	#2	violated clause

Table 1: BCP process. AC = *antecedents*, Con. = *conclusion*.

Batory originally used BCP to support the configuration process of a feature model by giving explanations in propositional logic why features cannot be selected anymore [4]. In our previous work, we adapted BCP to explain dead and false-optional features, void feature models and redundant constraints [17]. We extend this work to also explain implicit constraints in feature models and to give feedback in an user-friendly manner.

BCP for Implicit Constraints The slicing algorithm described in the previous section extracts a new, so called, sliced feature model with fewer features while preserving dependencies between features. Inputs to the slicing algorithm involve a complete feature model, which is used to extract a sliced model, and a set of features to remove. The sliced feature model may contain additional cross-tree constraints, which we refer to as implicit constraints. The detection of implicit constraints requires a comparison between (global) cross-tree constraints of the complete feature model and cross-tree constraints of the sliced model. If a constraint of the sliced model does not appear in the complete feature model, it is implicit.

We put the following thoughts into explaining an implicit constraint: Feature model slicing constructs an implicit constraint by combining two specific clauses. One clause contains a positive form of the literal to remove, while the other contains a negative form of it. The resulting implicit constraint (resolvent) represents a hidden dependency between the involved clauses. Implicit constraints are always redundant ones, since their information is already available regarding the complete feature model. The global cross-tree constraints contain the same information, but are not helpful in a partial feature model as the dependencies are expressed in a much more complex manner and features may by unknown in the partial model. We pass the following input parameters to BCP:

- A complete feature model in CNF. We do not pass the sliced feature model, because an explanation can only arise from the complete feature model.
- A truth value assignment for features from the implicit constraint making the constraint non-satisfiable.

An explanation can involve features from multiple neighboring partial models, which is why the sliced feature model is not sufficient to generate a full explanation. Redundancy occurs if the relationship expressed in the cross-tree constraint is already modeled in some other way in the feature model, which is exactly the case for an implicit constraint. We can use this relationship to explain the constraint. The CNF of the complete feature model does not contain the redundant constraint and forms the first input parameter. During the creation of the CNF, we additionally store information about the tracing of each literal to the feature model. Every literal belongs to a clause which either originates from a cross-tree constraint or from the feature hierarchy. This tracing enables us to provide a user-friendly explanation to the developer.

The crucial part is the selection of premises as the second input. The truth values of the implicit constraint must result in a contradiction in order to generate an explanation with BCP. Hence, we select the premises resulting in the redundant constraint to be violated. A contradiction is bound to occur, since the information of the constraint is still contained in the CNF due to redundancy. However, multiple assignments can lead to a non-satisfiable constraint and it is necessary to analyze all combinations. Every combination can result in a different explanation giving us only a partial solution for the cause of an implicit constraint. A union of all incomplete explanations produces the final result which is able to explain an implicit constraint. Duplicate explanation parts are ignored to reduce the length. A detailed description for explaining redundant constraints is given elsewhere [17].

Again, consider the feature models in Fig. 5a resulting in an implicit constraint $C \Rightarrow D$. In order to apply the slicing algorithm, we need to pass a complete feature model. Tools like VELVET support the composition of interrelated feature models [31]. A composition of feature models results in a new abstract root containing the prior roots as child features. In Table 2, we present the BCP process in order to explain the implicit constraint $C \Rightarrow D$. First, we pass the feature model in CNF and premises to BCP. In this case, we have only one truth value assignment leading to a violated clause with $D = false$ and $C = true$. Then, BCP collects unit-open clauses from the CNF and pushes them on the stack. We refrain from showing the complete CNF due to its length. BCP concludes G to be true and updates all respective literals in the CNF with the truth value. This results in a violated clause $\neg G \vee D$. An explanation can be generated by reporting the reasons: first, we take the violated clause into account and, second, we traverse the reasons for conclusions backwards to the premises. Initial value assumptions do not need to be reported to shorten the explanation.

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).*				

Table 2: Explaining the implicit constraint $C \Rightarrow D$.

Returning to our running example, the PPU, 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)

Table. 3 presents explanations for the implicit constraints above retrieved using our extension of FeatureIDE. The first constraint is a hidden implication in the customer feature model because of two global cross-tree constraints. A similar dependency is detected for the software feature 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. We have two features in the customer model implying different features in an alternative relationship 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 workpiece types *Small* and *Large*. Again, both imply different features even from different feature models. How-

Constraint	Explanation	Partial Models
Diagnosis ∨ ¬ Selfhealing	SelfhealingS → Diagnosis is a constraint Selfhealing → SealfhealingS is a constraint	Customer, Software
¬ Small ∨ ¬ Large	Large → CylinderM is a constraint CylinderM and ChangeoverArmM are alternative children of LiftingLowering Small → ChangeoverArmM is a constraint	Customer, Mechanical
Cylinder ∨ ChangeoverArm	PPU is the root PPUC is a mandatory child of PPU WorkPieces is a mandatory child of PPUC Size is a mandatory child of WorkPieces Small and Large are or children of Size Small → ChangeoverArm is a constraint Large → CylinderM is a constraint CylinderControl ⇔ CylinderM is a constraint CylinderM ⇔ Cylinder is a constraint	Customer, Mechanical, Electrical, Software
Diagnosis ∨ ¬ SelfhealingS	Diagnosis → DiagnosisS is a constraint SelfhealingS → Diagnosis is a constraint	Customer, Software

Table 3: Explaining implicit constraints of the PPU

ever, due to bijections, an implicit dependency occurs in the electrical feature model.

The last column in Table 3 depicts all involved partial feature models for the specific implicit constraint. For the PPU, we mostly have two partial models responsible for a hidden dependency. However, in one case features from all partial models are present in the explanation. A complete explanation often involves features from multiple partial models which is necessary to fully understand the hidden dependency and supports the communication between developers, since it is now obvious which features are responsible.

5. Evaluation

We give a description of our prototypical implementation in FeatureIDE and explore the adapted BCP algorithm in practice with a large-scale feature model. For the evaluation, we investigate the computation time, the number and structure of the implicit constraints. In detail, we look at the following research questions:
RQ1: *How many implicit constraints exist?*
RQ2: *How long does it take to derive them?*
RQ3: *What is their structure?*
RQ4: *How many partial models are involved?*

5.1 Implementation

We implemented our approach in the open-source framework FeatureIDE and it is part since release, FeatureIDE 3.1.0 [1]. It uses the CNF generated from the feature model to infer causes leading to an implicit constraint and the explanations are built by a combination of several reasons. However, BCP in its original form gives us only pure CNF clauses for an explanation leading to a contradiction (cf. Section 3). This representation is hardly comprehensible by developers, since the relation to the feature model is not obvious. As it is our goal to provide useful feedback to the developer, we introduced a tracing between the feature model and the literals in its CNF.

Every feature comprises structural information in a feature model. It always occurs in the tree topology and can take different roles with *child* or *parent* and additionally it can be *mandatory*, *optional* or in *alternative* groups and *or* groups. Furthermore, it may be present in cross-tree constraints. We take advantage of the structural information to present the explanations in a user-friendly manner reflected in the graphical feature model (cf. Section 4).

After modeling a feature model in FeatureIDE, our approach can be executed by simply 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 feature. Below the root feature, all features appear in the same way as in the complete feature model. Global cross-tree constraints relevant 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. The presented partial model is not editable at the moment. Reflecting changes in the partial model back to the complete model is a challenge for future work.

5.2 Application

At first, we focus on performance measurements and evaluations concerning the number of implicit constraints in a large product line to answer RQ2 and one part of RQ1. As case example, we have a feature model from the automotive industry with 2,513 features and 2,833 global cross-tree constraints. The execution time has been measured using an Intel(R) Core(TM) i7-4800MQ CPU with 2.7 GHz and 16-GB RAM.

The feature model slicing algorithm has an average computation time of 0.44 s across all considered partial feature models. Table 4 presents the detailed results of our evaluation. The first column refers to the depth of a feature in

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

the complete model that is selected as the root of the partial feature model. For instance, the automotive example has six child features below its root at depth one. The third column contains the number of features in these partial models, e.g., the largest model at position 5 contains 2,065 features in its subtree. This partial model also contains with a count of eleven most of the implicit constraints at this depth. Overall, there are twelve hidden dependencies which seems to be a rather small amount in comparison to the size of the complete feature model. Furthermore, we evaluate the child features of these six features as well resulting in 25 analyzed partial models. The number in brackets indicates to which parent feature (1-6) the 25 features belong. We can observe that the amount of implicit constraints significantly increased with 189. The computation time for partial models without any implicit constraints is close to the 0.44s of the slicing algorithm, since our approach does not start any explanation attempts. Depending on the number of derived implicit constraints, the computation time rises constantly. However, this is to be expected and the time with at most 170s for 123 constraints is still acceptable. The average number of explanation parts (or length of the explanation) is 15.1 parts at depth one and 16.46 at depth two. Given several hundred features in even in the partial models, we believe that the explanation is still comprehensible and definitely beneficial for the developer in understanding the dependencies. A more detailed reasoning on the explanation length is available in our prior work [17]. Thus, we conclude that implicit constraints are present and relevant for real-world feature models (cf. RQ1) and the computation time of BCP is acceptable (cf. RQ2).

RQ3 questions the structure of the derived implicit constraints. We were able to identify three major groups of logical expressions:

1. Implication: The implicit constraint represents an implication, e.g. $A \Rightarrow B$ or $A \wedge B \Rightarrow C$.
2. Exclusion: The implicit constraint represents a mutual exclusion between features, e.g. $\neg(A \wedge B)$.
3. Negation: The implicit constraint represents a negated feature, e.g., $\neg A$.

Table 5 presents our classification of the derived 201 implicit constraints into the three major groups and an additional group for any another kind of logical expressions. We also give examples of the identified CNF patterns. As indicated by the overall percentage, almost all implicit constraints can be mapped to one of the major categories. Implication forms by far the largest group with about 75%. Finally, we took a closer look to the features occurring in the constraints. Naturally, an implicit constraint involves additional partial feature models meaning that some features in the constraint are part of another submodel. We observed that up to four individual submodels are involved in an implicit constraint. Less than 40% of all features in an explanation are local features meaning features of the partial model

Depth	Partial Model (Parent)	# Features	# IC	Overall (s)
1	1	105	1	0.81
1	2	171	-	0.46
1	3	54	-	0.52
1	4	112	-	0.51
1	5	2065	11	542.6
1	6	5	-	0.43
2	7 (1)	8	-	0.5
2	8 (1)	72	-	0.47
2	9 (1)	4	-	0.48
2	10 (1)	3	2	1.05
2	11 (1)	17	-	0.45
2	12 (2)	167	-	0.48
2	13 (2)	3	-	0.45
2	14 (3)	21	2	2.08
2	15 (3)	18	-	0.46
2	16 (3)	3	-	0.46
2	17 (3)	3	-	0.44
2	18 (3)	5	-	0.44
2	19 (3)	3	-	0.41
2	20 (4)	3	-	0.46
2	21 (4)	88	-	0.45
2	22 (4)	16	-	0.43
2	23 (4)	4	-	0.45
2	24 (5)	684	123	170.83
2	25 (5)	16	-	0.44
2	26 (5)	948	39	75.7
2	27 (5)	231	20	21.24
2	28 (5)	185	3	2.51
2	29 (6)	2	-	0.45
2	30 (6)	1	-	0.44
2	31 (6)	1	-	0.46

Table 4: Implicit constraints (IC) for the automotive model.

Logic	CNF Pattern	Overall
Neg.	$\neg A$	8.6 %
Impl.	$\neg A \vee B$	14.6 %
	$\neg A \vee \neg B \vee C$	58.1 %
	$\neg A \vee B \vee C$	2.5 %
Excl.	$\neg A \vee \neg B$	15.7 %
Other	$A \vee B \vee C$	0.5 %

Table 5: Classification of implicit constraints in logical expressions and representation as CNF patterns.

in which the implicit constraint occurred. An explanation is comprised of 24% up to 56.1% local features. On average, explanations comprise 37.35% local features leading us to the conclusion, that a hidden dependency is mostly caused by relations between features from other submodels, thus answering (RQ4).

Consequently, implicit constraints are necessary to understand partial feature models in isolation.

6. Related Work

A considerate amount of research has been conducted on the automated analysis of feature models and feature model-

ing itself. FeatureIDE, TVL, FAMILIAR, SPLConqueror, Clafer, and pure::variants are only some results of this work [3, 26, 32]. Many approaches already provide support for analysis techniques such as anomaly detection [5, 6, 23], but only some respect dependencies between different partial models [20, 31] and even fewer actually explain the dependencies or anomalies [6, 24]. Furthermore, we are the first to make implicit constraints explicitly visible for the developer during the modeling process.

Removing features from a feature model is called feature model slicing [2] and several approaches exist in literature to perform this task [2, 34]. We take advantage of an approach implemented in FeatureIDE by Krieter et al. [18] with which a larger case study was conducted elsewhere [30].

A different concept to reduce complexity for the developers and present only relevant information is feature model views. Instead of actually removing the features as in slicing, views hide undesired features. Clark et al. [7] provide a first theory for feature model views. Checking the compatibility of different views as well as reconciliation of compatible views is possible. Views are often connected to the configuration process and not the actual development or maintenance of a feature model [14, 29]. Similar to the previous approach, a formalism for defining multiple views and checking the consistency is developed and evaluated. These aspects can be extended to the concept of multi software product lines combining several product lines and expressing their dependencies [38]. VELVET supports the development of such multi software product lines. It also adapts the concept of merging the separate feature models together below an artificial root and lets stakeholders express dependencies between the individual models [31]. Lettner et al. [20] added functionality in FeatureIDE to support different modeling spaces, modeling at different abstraction levels and dependencies between the spaces using inter-space dependencies. Inter-space refers to dependencies between the solution, problem, and configuration space based on the feature model. Revealing and understanding such dependencies between features from different spaces turned out to be extremely challenging. Although, we do not operate in different spaces, this challenge can be exactly mapped to dependencies in partial feature models. Another approach working at the configuration level was presented by Mendonça et al. [24]. Implicit dependencies are also detected with truth value propagation in a CNF and presented in a graph-based manner during configuration time.

A different type of implicit constraint is described in [15]. Modeling a product line with a feature model and an architectural model may lead to inconsistencies, e.g., the feature model allows configurations that are physically not possible due to architecture restrictions. As a result, the feature model is explicitly enhanced with implicit constraints to express these aspects. Hidden dependencies can also have an impact by linking features to code artifacts. Removing a feature might have undesired side effects if the feature was part of an implicit dependency and not known to the developer. Therefore, a tracing is proposed as a possible solution [13].

Considering the explanation process, we are most closely related to our previous work and work done by Batory [4]. The first attempt to relate propositional formulas to product lines was executed by Mannion [22]. Building upon this work, Batory adapted a BCP algorithm to support developers in the configuration process of a variant using a feature model [4]. 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 constraints is present and the explanations are presented in propositional formulas. In our previous work [17], we used this approach as foundation and provided explanations for different anomaly types, e.g., dead and false-optional features, void feature models, and redundant constraints. We extended this approach to explain implicit constraints in partial feature models in an user-friendly manner. Explaining these anomalies is also presented in other works [6, 28, 36]. However, no approach considers the explanation of implicit constraints.

7. Conclusion

We have presented an approach for deriving and explaining implicit constraints in partial feature models. Implicit constraints are numerously present in product lines and cannot be neglected for development and maintenance purposes. Our evaluation with an industrial-size feature model is a major contribution and shows the general feasibility and scalability of the approach. 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.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution. We would like to thank Stefan Feldmann for his support in validating the constraints of the PPU and Sebastian Krieter for his help with FeatureIDE.

References

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Decomposing feature models: Language, environment, and applications. ASE, Washington, DC, USA, 2011. IEEE Computer Society.

[2] M. Acher, P. Collet, P. Lahire, and R. B. France. Slicing feature models. ASE, pages 424–427, Washington, DC, USA, 2011. IEEE Computer Society.

[3] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski. Clafer: Unifying Class and Feature Modeling. *Software & Systems Modeling*, pages 1–35, 2014.

[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 7–20, Berlin, Heidelberg, 2005. Springer.

[5] D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank. *Automated Analysis in Feature Modelling and Product Configuration*. ICSR. Springer, Berlin, Heidelberg, 2013.

[6] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[7] D. Clarke and J. Proença. Towards a theory of views for feature models. In G. Botterweck, S. Jarzabek, T. Kishi, J. Lee, and S. Livengood, editors, *SPLC Workshops*, pages 91–98. Lancaster University, 2010.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, New York, NY, USA, 2000.

[10] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *JACM*, 7(3):201–215, 1960.

[11] S. Feldmann, C. Legat, and B. Vogel-Heuser. Engineering Support in the Machine Manufacturing Domain through Interdisciplinary Product Lines: An Applicability Analysis. *IFAC-PapersOnLine*, 48(3):211 – 218, 2015.

[12] A. Felfernig and M. Schubert. Fastdiag: A Diagnosis Algorithm for Inconsistent Constraint Sets. In *DX 2010, Portland, OR, USA*, pages 31–38, 2010.

[13] Y. Ghanam and F. Maurer. Linking feature models to code artifacts using executable acceptance tests. In *SPLC*, pages 211–225, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. K. Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *Software and System Modeling*, 12(3):641–663, July 2011.

[15] M. Janota and G. Botterweck. *Formal Approach to Integrating Feature and Architecture Models*, pages 31–45, FASE. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] 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 CMU/SEI-90-TR-21, SE Institute, 1990.

[17] M. Kowal, S. Ananieva, and T. Thüm. Explaining Anomalies in Feature Models. GPCE, New York, NY, USA, 2016. ACM.

[18] S. Krieter, R. Schröter, T. Thüm, W. Fenske, and G. Saake. Comparing algorithms for efficient feature-model slicing. In *SPLC*, New York, NY, USA, September 2016. ACM.

[19] C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in industrial plant automation: A case study. In *IECON*, 2013.

[20] D. Lettner, K. Eder, P. Grünbacher, and H. Prähofer. Feature modeling of two large-scale industrial software systems: Experiences and lessons learned. In *MODELS*, 2015.

[21] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*, pages 105–114, Washington, DC, USA, May 2010. IEEE.

[22] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 176–187, Berlin, Heidelberg, 2002. Springer.

[23] J. Meinicke, T. Thüm, R. Schöter, F. Benduhn, and G. Saake. An Overview on Analysis Tools for Software Product Lines. In *SPLatools*, New York, NY, USA, 2014. ACM.

[24] M. Mendonça, D. D. Cowan, W. Malyk, and T. C. de Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *JSW*, 2008.

[25] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer-Verlag, Berlin Heidelberg, 2008.

[26] M. Mernik, B. R. Bryant, G. Cabri, M. Ganzha, M. Acher, P. Collet, P. Lahire, and R. B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming*, 78(6), 2013.

[27] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, Sept. 2005.

[28] L. Rincón, G. L. Giraldo, R. Mazo, and C. Salinesi. An Ontological Rule-Based Approach for Analyzing Dead and False Optional Features in Feature Models. *Electronic Notes in Theoretical Computer Science*, 302:111–132, 2014.

[29] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *MODELS*, pages 252–268, Berlin, Heidelberg, 2012. Springer.

[30] R. Schröter, S. Krieter, T. Tüm, F. Benduhn, and G. Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *ICSE*, pages 667–678, New York, NY, USA, May 2016. ACM.

[31] R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated Analysis of Dependent Feature Models. In *VaMoS*, pages 9:1–9:5, New York, NY, USA, Jan. 2013. ACM.

[32] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal*, 20(3):487–517, 2012.

[33] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Conf. on Computer systems*. ACM, 2011.

[34] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *ICSE*, pages 254–264, Washington, DC, USA, May 2009. IEEE.

[35] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *SPLC*, pages 191–200, Washington, DC, USA, Aug. 2011. IEEE.

[36] P. Trinidad. *Automating the Analysis of Stateful Feature Models*. PhD thesis, University of Seville, 2012.

[37] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Technical Report TUM-AIS-TR-01-14-02, TU München, 2014.

[38] L. A. Zaid, F. Kleinermann, and O. De Troyer. Feature assembly framework: Towards scalable and reusable feature models. In *VaMoS*, pages 1–9, New York, NY, USA, 2011. ACM.