Detecting and Explaining Conflicts in Attributed Feature Models*

Uwe Lesta Ina Schaefer Tim Winkelmann

Institute of Software Engineering and Automotive Informatics
Technische Universität Braunschweig
Braunschweig, Germany

Lesta@sbs-softwaresysteme.de i.schaefer@tu-braunschweig.de t.winkelmann@tu-braunschweig.de

Product configuration systems are often based on a variability model. The development of a variability model is a time consuming and error-prone process. Considering the ongoing development of products, the variability model has to be adapted frequently. These changes often lead to mistakes, such that some products cannot be derived from the model anymore, that undesired products are derivable or that there are contradictions in the variability model. In this paper, we propose an approach to discover and to explain contradictions in attributed feature models efficiently in order to assist the developer with the correction of mistakes. We use extended feature models with attributes and arithmetic constraints, translate them into a constraint satisfaction problem and explore those for contradictions. When a contradiction is found, the constraints are searched for a set of contradicting relations by the QuickXplain algorithm.

1 Introduction

Matys [16] describes a product line (PL) as a group of related products which fulfill similar functions, complement one another or are destined for the same target group. Variability models are used to support the product configuration process and to analyze the variability of the existing PL. In the context of software product lines (SPLs) [17], feature models (FMs) [14] are a widely used formalism for variability modeling. Product configuration is an optimization task in order to find the most suitable product that meets a set of predetermined requirements and restrictions. The definition of an objective function for this optimization in a pure FM, i.e., a feature model without any attributes, can only refer to the section of as many or as few features as possible. In attributed feature models (AFMs) [3], where each feature can have multiple integer-valued attributes, optimization during product configuration becomes much more interesting. In particular, as attributes can have arithmetic constraints between each other which need to be satisfied in a configured product. Defining large and complex variability models, such as AFMs with integer-valued attributes and arithmetic constraints, is error-prone. Errors in the variability model may lead either to an undesired configurable product or to a realizable product which cannot be configured. Another type of modeling error is to relate features or attribute values in a wrong way which may either lead to a contradiction between the model and the reality or to a contradiction within the variability model. Contradictions within the variability model are hard to find and are mostly accidentally discovered without tool support. Furthermore, it is a time consuming and difficult task to find the reason for such a contradiction manually in order to resolve it.

To counter this problem, we present an approach to detect contradictions within AFMs in an efficient way. We further provide means to explain the cause of the contradiction to support the modeler to build a consistent AFM. There has been a lot of work on finding contradictions in the FM community on

*This work was partially supported by the DFG (German Research Foundation) under grants SCHA1635/2-1 and SCHA1635/4-1 and by the European Commission within the project HyVar (grant agreement H2020-644298).

J. Atlee and S. Gnesi (Eds.): FMSPLE 2015
EPTCS 182, 2015, pp. 31–43, doi:10.4204/EPTCS.182.3

pure FMs [8, 29, 27, 22, 4], but to the best of our knowledge there were only suggestions to do this on AFMs with integer-valued attributes and general arithmetic constraints. The analysis of AFMs with integer-valued attributes and arithmetic constraints is particularly complex [4], since not only attribute-value pairs have to be considered, but also arithmetic constraints. The number of products derived from the model can change drastically if such arithmetic constraints are involved. We describe how an AFM with integer attributes can be translated into a constraint satisfaction problem (CSP) for configuration and analyses purposes. We show how the detection of contradictions can be realized efficiently by reducing the size of the AFM, by reducing the number of checks and by making the checks themselves more efficient. In addition, we present how contradictions within the AFM can be detected automatically. To this end, we adapt the QuickXplain algorithm [12, 13] in order to efficiently find a contradicting set of constraints in the CSP. We implement our approach in SWI-Prolog [10] with the library CLP(FD), a constraint solver for finite domains written by Triska [24]. This solver provides powerful constraints for integer arithmetic, determines and is correct [25]. For evaluation we use a case example from a medium-sized mechanical engineering PL with 497 features and 1990 attributes.

The outline of this paper is as follows: In Section 2, we describe the considered variability model and its translation into a CSP. In Section 3, the different kinds of contradictions are defined and an efficient approach for their detection is proposed. Afterwards, we describe in Section 4 how the contradicting relations can be found and presented to the modeler in a way that helps him to resolve the contradiction. A case study for the developed tool support is presented in Section 5. In Section 6, related approaches are described, before the paper is summarized in Section 7.

2 The variability model

The variability of a PL can be described by a FM. To make optimization possible, each feature may have some integer attributes. The variability model we consider in this paper is called AFM [3] if it contains features with integer-valued attributes and consists of the following two parts:

- A (graphical) notation which represents the hierarchically arranged set of features and attributes with the relationships between parent- and child features called feature-tree.
- cross-tree constraints (CTCs) that specify conditions, i.e., relations over the integer values of the attributes, that a product configuration must satisfy. These are usually written in a textual notation in Figure 1.

As an example, Figure 1 shows a feature-tree with CTCs of a simplified robot PL from a real world AFM for configuration purposes. A *Robot* consists of a *Motor*, a *Tool* and an optional *Protective_grid*. The *Motor* attribute *Motor:pwr* allows a motor with either 10, 20 or 30 kW driving power. The alternative-

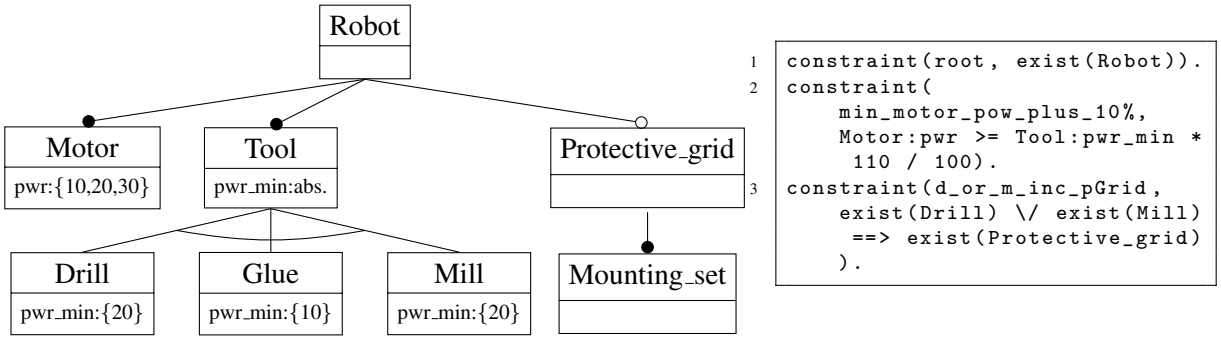


Figure 1: AFM of a Robot

```
Robot = Tool      // Mandatory
Robot = Motor
Protective_grid = Mounting_set
Robot = 0 ==> Protective_grid = 0 // Optional
Protective_grid = 1 ==> Robot = 1
sum([Mill, Glue, Drill], =, Tool) // Alternative
Drill = 0 <==> Drill:pwr_min = nil // Attribute
Glue = 0  <==> Glue:pwr_min = nil
Mill = 0  <==> Mill:pwr_min = nil
Motor = 0 <==> Motor:pwr = nil
Tool = 0  <==> Tool:pwr_min = nil
// Abstract attribute
element(_, [Drill:pwr_min, Glue:pwr_min, Mill:pwr_min], Tool:pwr_min)
Tool:pwr_min \= 20 ==> Mill:pwr_min  \= 20
Tool:pwr_min \= 10 ==> Glue:pwr_min  \= 10
Tool:pwr_min \= 20 ==> Drill:pwr_min \= 20
// cross-tree-constraints
Robot = 1
Motor:pwr >= (Tool:pwr_min) * 110 / 100
Drill = 1 \/ mill = 1 ==> Protective_grid = 1
```

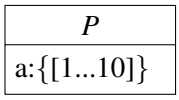
Listing 1: AFM from Figure 1 as constraints

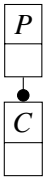
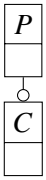
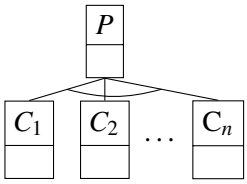
group *Tool* can be specialized either as a *Drill* or a *Glue* or a *Mill* tool. Each possible tool defines its minimum input power by an attribute *pwr_min*. The abstract attribute *Tool:pwr_min* groups the child attributes. The optional *Protective_grid* requires also a *Mounting_set*. These mandatory feature chains, which do not change the variability of the AFM, are typical for real-world configuration models. They reflect the hierarchically whole-part relations of a machine. Often, they are extended by attributes like price or weight which are summarized up to the root by CTCs for possible optimisation. The CTCs are specified in a textual form by the keyword `constraint` followed by a name and the actual constraint. The syntax and semantics of the constraints in this case follow SWI-Prolog [10]. Feature- and attribute names are translated into variables of the constraints. The construct `exist` requires that the feature variable is set to true, while the construct `nonexist` demands the opposite.

A FM can be translated, e.g., into propositional logic (SAT solver), description logic, constraint logic, binary decision trees [4]. The translation of an AFM into a CSP is the most suitable representation for product configuration purposes and analyses of contradictions, because most CSP solvers offer good support for optimisation and can handle arithmetic constraints. Also explaining a contradiction in terms of constraints is best done in the same language the modeller defines the AFM. A CSP is described by Tsang [26] as a Triple $csp = \{Z, D, \mathscr{C}\}$ where Z is a finite set of variables, D is a set of possible domain values for each variable and $\mathscr{C}$ is a finite, possibly empty, set of constraints which are restricting the domain values of the variables. A CSP solver takes a CSP and searches for a possible solution which contains assignments for each variable from its corresponding domain which satisfies all constraints. The solving process is divided into two substantial steps. The first step tries to reduce the solution space of the CSP by value propagation where some, but generally not all, invalid domain values are removed from D. In the second step, all remaining variable assignments are explored, mostly by backtracking, to find a solution. This step is known as the labeling process of a CSP-Solver. A CSP is called consistent if a solution exists, otherwise it is called inconsistent. We translate the AFM into a CSP. The mapping of the feature-tree is shown in Table 2 and is based on the work of John [11]. For each feature, a variable with the possible domain values 0 and 1 is created where 1 represents the existence of the feature and 0 its absence. For each attribute of a feature, a variable is generated with the given possible domain values of the attribute and an additional *nil* value. The value *nil* indicates that no domain value is valid for this attribute. In this case, the feature of the attribute must also be absent. This is captured by the constraint

$P = 0 \Leftrightarrow a = nil$ which sets the value *nil* to the attribute variable when its feature does not exist and vice versa. Since every value of the attribute can be chosen they are known as an optional attribute values.

	Relation	CSP-Translation

Feature		`P in 0..1`
		`a in 1..10 \/ nil`
		`P = 0 <==> a = nil`

Mandatory		`C <==> P.`
Optional		`C = 1 ==> P = 1.` `P = 0 ==> C = 0.`
Alternative		`sum([C1,C2..Cn],=, P).`

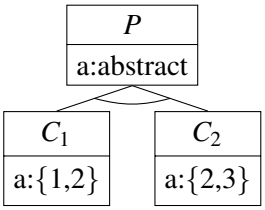
Abstract Attribute		`element(_, [C1:a, C2:a ], P:a).`
		`P:a \= 1 ==> C1:a \= 1.`
		`P:a \= 2 ==> C1:a \= 2 \/ C2:a \= 2.`
		`P:a \= 3 ==> C2:a \= 3.`

Table 1: Mapping graphical AFM elements to a CSP

For a mandatory feature relation where the feature C is a part of the feature P, a constraint $P = C$ is defined. This constraint ensures that both features or neither of them must exist. For an optional feature C which may be part of the feature P, the following constraint is created $(C = 1 \Rightarrow P = 1) \wedge (P = 0 \Rightarrow C = 0)$. This constraint ensures that, if the feature C exists, the feature P must also exist and if P does not exist the child feature C cannot exist. For an alternative-group where only one child feature of P can exist, the constraint $\texttt{sum}([C_1, C_2, ...C_n], =, P)$ is added. This constraint ensures that the sum of the existence variables $C_1...C_n$ is equal to the existence variables of P. So neither P nor any children of P exist or P and only one child of P can exist.
For the parent feature P of an alternative-group, an abstract attribute a can be defined if all children $C_1...C_n$ define an attribute a with the same name and type, similar to abstract super-classes in OOP. The set of possible domain values for the abstract attribute a at the feature P is the union of all domain values of all attributes a from the children $C_1...C_n$. For the abstract attribute a in Table 1, the domain values would be $\{1,2,3\}$.

This can be expressed by the constraint $\texttt{element}(\_, [C_1{:}a, C_2{:}a], P{:}a)$. If a domain value is removed from the abstract attribute, this value must also be removed from all child attributes. A constraint of the form $P{:}a \neq dv \Rightarrow C_i{:}a \neq dv$ is created for each domain value dv that occurs in the attribute a of C_i based on John [11]. John shows that this translation rules lead to a tree-like CSP in which the nodes represent variables and the undirected arcs represent the existence of an constraint between them. Freuder [9] shows that tree-like constraint graphs can be solved without any backtracking. So the CSP obtained by translating the feature tree can be solved by arc-consistency [15] propagation mechanism of a solver.

The CSP generated from the AFM in Figure 1 is shown in Listing 1. Each feature name represents a CSP variable. In lines 1 to 3, the constraints express the mandatory features. If *Robot* exists, *Motor* and *Tool* must also exist, and if *Protective_grid* exists, the *Mounting_set* must be also present. The constraints in line 5 and 6 are derived from the optional feature *Protective_grid*. The `sum` constraint in line 8 ensures that only one *Tool* is selected, if it exists. In lines 7 to 11, the constraints ensure consistency between the

existence of a feature and the attribute values. For the abstract attribute *Tool:pwr_min*, the constraints in lines 13 to 16 are generated. The CTCs are shown in lines 18 to 20.

3 Contradictions

The development of an AFM is an error-prone task. Benavides et al. [4] propose the most complete collection of analysis tasks we are aware of up to now. Searching for contradictions in an AFM is one of these analysis tasks. It is also called detection of anomalies. There are two kinds of contradictions that occur in a variability model: A **model contradiction** (aka. model anomaly) is a contradiction between the semantics of the feature tree and the CTCs (e.g. an optional feature which is present in all possible products due to cross-tree-constraints). An **inconsistent model** (aka. void feature model) exists if no product can be derived from the model at all (e.g., a cross-tree constraint requires the absence of a full mandatory feature). Model contradictions can be described as properties of the AFM and its elements [23]. The following properties can be determined automatically because these contradictions always consist of at least one cyclic dependency of the relations in the AFM:

- *Void feature model*: This property of a FM is true if the FM is inconsistent which means that it represents no product at all.
- *Dead feature*: This property is true if a feature cannot be selected in any product, therefore, it appears in no product.
- *False optional feature*: The *false optional* feature property becomes true if a feature is included in all products of the product line despite not being modeled as mandatory.

In addition, these properties can be adapted to attributes domain values of AFMs aswell.

- *Dead attribute values*: This property of a domain value becomes true if the value cannot be assigned to its attribute in any product. (E.g., in Figure 1, the domain value 10 cannot be assigned to *Motor:pwr* because a cross-tree constraint requires a minimum motor power of *Tool:pwr_min* plus 10 percent). This property is the same as *non-attainable domain values* introduced by Salinesi and Mazo [18].
- *False optional attribute values*: An attribute value has the property *false optional* if it is included in all products of the product line despite being modeled as an optional attribute.

The reasons that makes any of the above properties true are always related to cross-tree constraints. For an AFM without cross-tree constraints, none of these properties can be true because the CSP, constructed from the feature tree is a tree-like CSP which contains no cyclic dependencies. Imagine a tree where the nodes are variables or values of the CSP and the arcs are the relation between them. In this case, a solution of the CSP always exists. A contradiction is only possible if the tree becomes a graph by an additional relation from a CTC.

Detection of Contradictions The main operation to detect a contradiction is a method which takes an AFM and derives a product from it. In other terms, a constraint solver needs to calculate a solution of a CSP which represents the AFM. This is exactly the case for detecting a *void feature model*. In order to detect the other contradictions, we need the possibility to add additional constraints to the AFM, e.g., to detect a *dead feature*, a constraint which selects this feature must be added. For this reason, Trinidad [22] proposes to distinguish between a FM and a stateful feature model (SFM). A SFM can also contain additional constraints, corresponding to user decisions and feature selections. A naive approach to find contradictions is to check all features and attribute values. After checking the AFM for a solution (no *void feature model*), for each feature the following steps are executed to detect contradictions:

1. Select a feature. A constraint is created where the variable which represent the feature in the CSP is set to 1. This constraint is added to a copy of the FM resulting in a SFM.

2. Find a solution. The constraint solver is used to derive a product solution of the SFM. If no solution is found, the feature is marked as a *dead feature*.
3. If the feature is optional or part of a child in an alternative-group: Deselect this feature. A constraint is created where the CSP variable of the feature is set to 0 and the constraint is added to a fresh copy of the FM to get a SFM.
4. Find a solution. If no solution is found for the SFM from the step above, this feature is marked as a *false optional feature*.

The process to check the attribute values is similar. First all attributes are collected. For each attribute, all possible domain values must be checked. So for each domain value, the following steps are executed:

- Check for a *dead attribute value*. A constraint is created which sets the attribute to the domain value, adds the constraint to a copy of the AFM and searches for a solution. If there is no solution, this attribute value is marked as a *dead attribute value*.
- Check for a *false optional attribute value*. A constraint is created where the domain value is removed from the attribute and added to a copy of an AFM. If no solution is found, this attribute value is a *false optional attribute value*.

This approach is inefficient because not all features and attribute values have to be checked. Some properties for certain features or attributes cannot be true because these features or attributes are not involved in cross-tree constraints. Other properties must not be checked due to the semantics of the feature tree. E.g., a full mandatory feature must not be checked for the *dead feature* property if the property *void feature model* is false. Additionally, some contradictory properties for certain features or attributes can be true as a side-effect if this feature or attribute is connected to another feature or attribute by a feature-tree or cross-tree constraint for which a contradictory property is true.

Efficient Contradiction Detection To decrease the computation efforts during the search for contradictions, we reduce the CSP which represents the AFM and propose additional rules to decrease the amount of checked features.

The basic idea is to reduce the variability of the AFM by deleting all features and attributes which cannot be part of any contradiction. As a contradiction needs at least a cyclic path of its relations, we can safely delete all leaf-features and attributes which are not part of an alternative-group from the AFM which do not appear in any cross-tree constraint. Only cross-tree constraints can create cyclic relation paths in the AFM. In detail, we remove: (1) Each set of attributes marked as abstract in an alternative-group where none of the attributes appears in any cross-tree constraint. (2) Any other attribute which does not appear in any cross-tree constraint. (3) The features of an alternative-group if none of the features has an attribute or a child-feature and none of them appears in any cross-tree-constraint. (4) And any other leaf feature not appearing in any cross-tree constraint.

The analysis process benefits from the reduced AFM in two ways: (1) Less features and attributes have to be checked for a possible contradiction. (2) A smaller CSP can be used in the checks to search for a possible product. In addition, the modeler benefits from the reduced AFM because only the root of a contradiction is found. Possible aftereffects are not detected, only the head features of mandatory feature chains are interesting for contradiction checking. Consider the feature *Mounting_set* in Figure 1. If the feature *Protective_grid* is a *dead feature*, it is clear that *Mounting_set* is also a *dead feature*. Therefore, it is only necessary to find and check the heads of those feature chains.

Apart from the reduced AFM, there are two more ways to increase the performance of contradiction analyses. The first is to investigate how the number of checks for a solution can be reduced since finding a solution in a CSP is an expensive task. The second is to improve the solution finding process itself.

Reduce Number of Checks While Salinesi and Mazo [18] only mention the possibility to reduce the amount of solution checks by comparing the set of features to be checked against the features of a derived product from a check, we implement this approach and improve it further as follows:

From the reduced AFM, we collect all optional features and children of an alternative-group from the feature-tree into a set of features to be checked. This set is duplicated to a set called `check_dead` for the *dead feature* checks and a set called `check_false_opt` for the *false optional feature* checks. All features of these sets are compared with the features from the derived product after the *void feature model* check of the AFM. A feature from the set `check_dead` will be removed if it is selected in the product. Features from the set `check_false_opt` will be removed if they are not in the product. This halves the amount of checks because each feature from the AFM is either selected or deselected in the derived product. For each remaining feature of the `check_dead` set, a *dead feature* check is performed starting with the features from the alternative-groups where only one feature can be selected at a time. Afterwards, the features from both sets are compared against the derived product from the *dead feature* check. If the FM contains large alternative-groups, a huge amount of *false optional feature* checks can be pruned because most features of an alternative-group are deselected in a product.

For further reduction of *dead feature* checks, we are interested in a derived product that contains as many features as possible from the features in the set `check_dead`. It can be obtained by an explicit labeling strategy for the CSP when solving the *dead feature* check. The variables representing the existence of the optional features from the `check_dead` set are labeled with a strategy which tries the highest domain values first. This results in a derived product where the maximum of optional features can be removed from the set `check_dead`. During the *false optional feature* checks, the labeling strategy tries the smallest domain values first for the CSP variables representing the optional features. During the *dead feature* checks, the labeling strategy for the optional features can be switched to the *false optional feature* check strategy if all optional features are removed from the set `check_dead`. Then, the *dead feature* checks derive products which contain the minimum number of optional features which are removed from the `check_false_opt` set. Depending on the amount of features in alternative-groups mostly the set of `check_false_opt` features is empty when all *dead feature* checks are done.

Reduce Checking Efforts The second way to increase performance is to speed up the CSP solver in finding a solution. In general, finding a solution of a CSP is a NP-complete problem [26]. With the reduced AFM, we decrease space, not the time complexity which comes with the cross-tree-constraints. This leads to less propagation steps in the CSP-Solver in the labeling process. In practical examples the performance gain from using the reduced AFM is about linear to the reduced size, as the amount of propagation steps are nearly linear to the amount of constraints generated from the feature-tree. The labeling options of the CSP-solver are the key to speed up solution finding. Most solvers offer the possibility to define the order of the variables, the domain value order and the branching strategy for the labeling process. Unfortunately, the fastest labeling strategy depends on the specific problem. In general, it is a good idea to store the labeling options within the AFM. As a starting point, we use an order where the variable with the smallest domain is labeled first. Then, the detection of infeasible values happens at an early stage. Additionally, for variables representing the existence of the features, an order is chosen where the variable participating in most constraints are labeled first with an enumerating branching strategy. For attribute variables, a bisection method as a branching strategy is chosen. These options are problem-specific and can change the performance heavily.

4 Explaining Contradictions

After detecting a *void feature model* or an other property indicating a contradiction, the modeler needs a clue to change this mistake in the variability model. The essential task for removal of contradictions

within the AFM is changing relationships between features or attribute values. Therefore, the contradicting relations must be detected and shown to the modeler. Afterwards the modeler must decide which relation needs to be changed in order to resolve the contradiction.

```
1 protective_grid = 0          % false_optional_feature
2 robot = 1
3 robot = tool                 % mandatory
4 sum([mill, drill], =, tool)  % alternative-group
5 drill = 1 #\/ mill = 1 #==> protective_grid = 1
6           %drill_or_mill_include_protective_grid1
```

Listing 2: Defiant constraints for the *false optional feature* *Protective_grid* if the feature *Glue* is removed

Consider for example the AFM shown in Figure 1, if the modeler removes the feature *Glue* from the alternative-group *Tool*, the feature *Protective_grid* becomes a *false optional feature*. Listing 2 shows the contradicting constraints for the *false optional feature* *Protective_grid* in this AFM. Line 1 shows the constraint to check the *false optional feature* property for the feature *Protective_grid*. The feature *Protective_grid* is disabled and the CSP becomes inconsistent which means that the CSP has no solution (the AFM is a *void feature model*). The constraint in line 2 expresses that a robot exists and the constraint in line 3 requires the existence of the alternative-group *Tool* if a robot exists. In line 4, the constraint determined that one feature from the alternative-group *Tool* must exist. The CTC in line 5 requires the feature *Protective_grid* if the feature *Mill* or *Drill* exists. Hence, the selection of feature *Protective_grid* contradicts the constraint in line 1.

For a given inconsistent CSP, a simple approach to find the contradicting constraints is to check all possible subsets of constraint combinations for consistency and select the smallest inconsistent set. The effort for this, however, is exponential. In real world scenarios, the modeler does not need the smallest conflict set. An arbitrary conflict set, which has no smaller subset of conflicting constraints, contains in most cases enough information for the modeler to solve the conflict.

QuickXPlain Junker developed the QuickXPlain algorithm [12, 13] which calculates an inconsistent set of constraints for a given inconsistent CSP. The number of consistency checks for the QuickXPlain algorithm is $O(n,k) = 2k * log_2(n/k) + 2k$ [13], with n the number of all constraints and k the number of constraints in the resulting conflict set. The number of constraints in the conflict set k in relation to the number of all constraints n has a vital impact on the number of consistency checks. The conflict set computed by QuickXPlain is one of possibly many, whereas the simple approach described above finds the smallest conflict set, but with $O(n) = 2^n$ consistency checks.

Modularization of AFM Finding an explanation for a contradiction is a time consuming task. We decrease the computation efforts by reducing the number of constraints to check and constraints in the conflict set. The basic idea is to call the QuickXPlain algorithm twice in a cascading way. Therefore, we group the constraints representing the AFM into large conjunctions of constraints and call the QuickXPlain algorithm on this constraint-groups. Afterwards, we split up the constraint-groups from the resulting conflict set back into their native constraints and run QuickXPlain again on this set of constraints. All constraint-groups which are not part of the initial conflict found by the first call of QuickXPlain can be ignored. This decrease the number of constraints for the second computation. The main challenge is using a good heuristic to build the constraint-groups for the first call of QuickXPlain. A good gouping has a small amount of constraint-groups and leads to small conflict set which contains in the best case with one constraint-group.

During development the modeler maps the structure of the PL into the AFM similar to multi software product lines [6]. Thereby, the modeler encapsulates subunits, respectively assembly units, of the PL into AFM-modules. That reflect the structure of the PL into the AFM for separation of concerns and reuse. This existing module structure can be exploited as a good heuristic for grouping of the constraints for efficient explanation of contradictions.

Scen-ario	Con. in the Contra-diction	Simple Search				Cascading Search			
		AFM 3402 Con.		reduced AFM 1161 Con.		AFM $1_{st}+2_{nd}$ search		reduced AFM $1_{st}+2_{nd}$ search	
		Time	Checks	Time	Checks	Time	Checks	Time	Checks
s1	7	16.7 s	61	5.3 s	57	0.49 s	44	0.47 s	41
s2	2	7.9 s	19	2.4 s	18	0.42 s	16	0.25 s	11
s3	2	9.4 s	22	2.1 s	16	0.40 s	15	0.22 s	12
s4	2	8.4 s	20	2.0 s	15	0.42 s	17	0.25 s	12
s5	2	9.1 s	21	2.3 s	17	0.42 s	16	0.31 s	15
s6	7	8.9 s	69	2.9 s	64	0.87 s	57	0.80 s	61
s7	8	9.1 s	78	2.8 s	75	1.06 s	67	0.85 s	72

Table 2: Amount of checks and execution times for finding explanations

All constraints of a module can be consolidated to one conjunctive constraint. It reduces the number of constraints for the first call of QuickXPlain to the number of modules. Furthermore, the resulting conflict set is also small because most conflicts occur within a single module or between two modules. This grouping with cascading calls of the QuickXPlain algorithm leads to ca. 10 times faster computation for the explaining conflict set (see Section 5).

If a module structure is not available, another idea to group the constraints is finding all clusters of connected CTCs and add all generated constraints from the feature-tree bottom up to these clusters. Other groupings, e.g., by the type of the relation in the feature-tree give a performance gain of less than 10 percent. The evaluation of different grouping approaches is left future work.

Implementation Details of a modularized AFM In order to build an AFM-module, all external dependencies of the module can be added as attributes in the module's root feature. This can be done, e.g., by adding CTCs between the root attribute and the features or attributes internal to the module. In our approach, we introduce feature and attribute references at the root feature which behave like the original feature or attribute, but are accessible from other modules via the module's root feature. The module's root feature becomes an interface of the module, containing all external dependencies between the features and attributes of the used module and of a super-ordinate module. For reusable modules, we support the marking of features and attribute values as *false optional* or *dead* by a '`pragma`' directive. This is necessary because when using one module multiple times in a greater AFM, the modeler does some pre-configuration of the generic module with cross-tree constraints which leads to model contradictions. These intentional contradictions are ignored at the stage of contradiction detection.

5 Case Study and Tool Support

The approach developed in this paper is implemented in SWI-Prolog. The translation of the feature-tree into constraints is written in constraint handling rules (CHR) [19] and SWI-Prolog. As case study for evaluating our approach, we use a configuration model for generating quotes for a medium-sized mechanical engineering application. This model contains 497 features with 1990 attributes of which 237 are optional features, 49 alternative-groups and 256 cross-tree-constraints. The feature-tree results in 3402 constraints in the CSP. Using this case study, we evaluate our proposed approach for detecting and explaining contradictions. First, we show that the reduction of the AFM leads to reduced computation times, both for detecting and explaining contradictions. Second, we show that the cascaded approach for explaining contradictions also decreases computation times. Randomly generated models are not suitable

for evaluation purposes as randomly generated constraints typically do not reflect realistic relationships.

The reduced AFM with the process described in Section 3 is about 50 percent smaller. It contains 222 features with 712 attributes, 90 optional features, 35 alternative-groups and 1161 generated constraints. In the AFM, there are 396 features to check for the *false optional* property (237 optional features plus 159 children from the alternative-groups) and 497 features to check for the *dead feature* property. With the efficient approach presented in Section 3, we need only 48 consistency checks for the remaining 161 features to discover the contradictions. Thereby, each consistency check is twice as fast because of the smaller amount of constraints to propagate. Hence, the overall performance improvement for detecting contradictions for the reduced AFM, compared with the native approach, is round a factor of 32.

Table 2 shows the amount of consistency checks and the computation time which is necessary to find a conflict set of constraints to explain a contradiction property. The execution times are determined on a Laptop with an Intel dual core processor T7500 with 2200 MHz and 3 GByte RAM by the SWI-Prolog predicate time/1 in seconds. Each line shows different scenarios, which descibe the effort to explain a model contradiction in the AFM. In column 2, the number of constraints in each of the conflict sets is listed. The block 'simple search' shows, for a single QuickXPlain call, the computation time in seconds and the number of consistency checks. Therein, the first two columns are for the full AFM and the second for the reduced AFM. In the second block 'cascading search' the time and checks are listed for cascaded QuickXPlain calls for the AFM and the reduced AFM. In a cascading search, the constraints are grouped by the module structure for the first call (comp. Sect. 4). These results show that our approach also improves the computation times for explaining contradictions. The best improvements are obtained with the cascading search. If we apply the cascaded search, the additional reduction of the AFM does not matter. If, however, the reduced AFM is used in a single search, the time required for explaining the contradiction is improved by about three times.

6 Related Work

The analysis of FMs is an active area of research [2, 18]. Most approaches consider pure feature models without attributes or just mention that their approach can be adapted to AFMs. Benavides [4] et al. present a classification of different approaches for the analysis of FMs. They mention numerous proposals to detect *void feature models*. Fewer approaches consider the detection of *dead features*, and even less deal with the detection of *false optional features*. In this survey only Trinidad [22] presents an approach on how to find and explain all the above mentioned anomalies in a pure FM. For this he used a constraint solver and Reiter's theory of diagnosis. In contrast to our work, Trinidad does not consider integer-valued attributes and general arithmetic constraints. There are two approaches which deal with detecting and explaining contradictions in AFMs that are related to our work. For explaining contradictions in an AFM, Zaid et al. [28] propose an approach based on description logic. This approach covers detection and explaination of *dead features* using the Pellet Reasoner [21]. However, in contrast to our work, they only use constraints on attribute values containing smaller-, greater- or equal-relations and no general arithmetic constraints as we do. Osman et al. [7] detect and explain anomalies in a FM by a knowledge-based method using propositional logic. He extends a FM by so called variation points. The variation points are cardinalities which allow a more complex grouping of features. However, there is no possibility to encode attributes with the possibility to express arithmetic cross-tree constraints.

Felfernig et al. [8] also use constraint sets to find inconsistencies in FMs which is very similar to our approach, but also lacks the handling of arithmetic constraints and integer-attributes. Zhang et al. [29] introduced refinement paths, which decreases the number of checks needed by a satisfiability (SAT) solver. But they only do this on a simplied version of pure FMs. A more practical approach is studied by Xiong et al. [27], where they provide a list of solutions on encountered configuration problems in the eCos

configurator. The eCos configurator also contains numeric attributes in its configuration domain. Their approach aims to support customers during their configuration and not the developer during modeling the AFM. There are other tools and languages which are designed for modeling and configuration of AFMs. For instance, *TVL* a text-based formal language for modeling AFMs designed by Classen et al. [5]. TVL contains structuring mechanisms for FMs, such as user-defined types or imports and AFMs with artihmetic constraints. Unfortunately checking such AFMs is still under construction and currently supports only pure FMs. Similarly, *FAMILIAR* by Acher et al. [1] is a textual language for representing FMs which allows (amongst other operations) im-/ex-porting, (de-)composing, slicing and diffing of (multiple) FMs. FAMILIAR supports reasoning about FMs, but still only FMs without attributes. The *SPLConqueror* from Siegmund et al. [20] uses attributed FMs and optimization functions to help stakeholders during the configuration. They also describe the use of constraints for feature restriction, but it is unclear how errors in the model can be detected and explained to the modeler. Another approach is the *Clafer* modeling language from GSD Lab at the University of Waterloo[1]. It is a DSL for modeling FMs with attributes and constraints. They support detection of void FMs, but direct support is still missing to find e.g. dead attribute values. Summarizing, none of the existing approaches (cf. [4]) supports the detection and explanation of contradictions, in the sense of all of the above mentioned properties, in AFM with integer-value attributes and arithmetic constraints.

7 Conclusion

In this paper, we presented an approach for the detection and explanation of contradictions in AFMs with integer-values attributes and general arithmetic constraints. Based on the translation of the feature-tree into a CSP, we presented how the labeling process of a constraint solver can be adapted for the efficient detection of contradictions. The QuickXPlain algorithm [12, 13] is used for computing a conflict set of contradicting constraints which explains the reason of a contradiction. For larger AFMs, we used a modularization concept similar to MSPLs to improve the computation time by cascading calls of QuickXPlain. For future work, we will further investigate possible tool integrations, heuristics to determine possible clusters of constraints in order to speed up the time for explaining contradictions. Additionally we want to compare our approach with an implementation of the Felfering et al. [8] algorithm FASTDIAG combined with a diagnosis model.

References

[1] Mathieu Acher, Philippe Collet, Philippe Lahire & Robert B. France (2013): *FAMILIAR: A Domain-specific Language for Large Scale Management of Feature Models. Sci. Comput. Program.* 78(6), pp. 657–681, doi:10.1016/j.scico.2012.12.004. Available at `http://dx.doi.org/10.1016/j.scico.2012.12.004`.

[2] Don Batory, David Benavides & Antonio Ruiz-Cortes (2006): *Automated Analysis of Feature Models: Challenges Ahead. Commun. ACM* 49(12), pp. 45–47, doi:10.1145/1183236.1183264. Available at `http://doi.acm.org/10.1145/1183236.1183264`.

[3] David Benavides, Pablo Trinidad Martín-Arroyo & Antonio Ruiz Cortés (2005): *Automated Reasoning on Feature Models.* In: *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, pp. 491–503, doi:10.1007/11431855_34. Available at `http://dx.doi.org/10.1007/11431855_34`.

[4] David Benavides, Sergio Segura & Antonio Ruiz-Cortés (2010): *Automated analysis of feature models 20 years later: A literature review. Inf. Syst.* 35(6), pp. 615–636, doi:10.1016/j.is.2010.01.001.

[1] `http://www.clafer.org/`

[5] Andreas Classen, Quentin Boucher & Patrick Heymans (2011): *A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL.* *Sci. Comput. Program.* 76(12), pp. 1130–1143, doi:10.1016/j.scico.2010.10.005. Available at `http://dx.doi.org/10.1016/j.scico.2010.10.005`.

[6] Ferruccio Damiani, Ina Schaefer & Tim Winkelmann (2014): *Delta-oriented Multi Software Product Lines.* SPLC '14, ACM, New York, NY, USA, pp. 232–236, doi:10.1145/2648511.2648536.

[7] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk & Chin Kuan Ho (2008): *Knowledge Based Method to Validate Feature Models.* In: *SPLC (2)*, pp. 217–225.

[8] A. Felfernig, M. Schubert & C. Zehentner (2012): *An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets.* *Artif. Intell. Eng. Des. Anal. Manuf.* 26(1), pp. 53–62, doi:10.1017/S0890060411000011.

[9] Eugene C. Freuder (1982): *A Sufficient Condition for Backtrack-Free Search.* *J. ACM* 29(1), pp. 24–32, doi:10.1145/322290.322292.

[10] T. Fruehwirth, J. Wielemaker & L. De Koninck (2012): *SWI Prolog Reference Manual 6.2.2.* Available at `http://books.google.nl/books?id=q6R3Q3B-VC4C`.

[11] Ulrich John (2002): *Konfiguration und Rekonfiguration mittels constraint-basierter Modellierung.* *DISKI* 255, Infix Akademische Verlagsgesellschaft.

[12] Ulrich Junker (2001): *QUICKXPLAIN: Conflict Detection for Arbitrary Constraint Propagation Algorithms.* In: *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, doi:10.1.1.23.3472. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.3472`.

[13] Ulrich Junker (2004): *QUICKXPLAIN: Preferred Explanations and Relaxations for Over-constrained Problems.* In: *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, AAAI Press, pp. 167–172. Available at `http://dl.acm.org/citation.cfm?id=1597148.1597177`.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak & A. S. Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report, SEI.

[15] Alan K. Mackworth (1975): *Consistency in Networks of Relations.* Technical Report, Vancouver, BC, Canada, Canada.

[16] E. Matys (2011): *Praxishandbuch Produktmanagement: Grundlagen und Instrumente.* Campus-Verlag.

[17] Klaus Pohl, Günter Böckle & Frank J. van der Linden (2005): *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[18] Camille Salinesi & Raúl Mazo (2012): *Defects in Product Line Models and how to Identify them.* In Abdelrahman Elfaki, editor: *Software Product Line - Advanced Topic*, InTech editions.

[19] Tom Schrijvers & Bart Demoen (2004): *The K.U.Leuven CHR system: Implementation and application.* In Th. Frühwirth & M. Meister, editors: *CHR '04: 1st Workshop on Constraint Handling Rules: Selected Contributions.*

[20] Norbert Siegmund, Marko Rosenmller, Martin Kuhlemann, Christian Kstner, Sven Apel & Gunter Saake (2012): *SPL Conqueror: Toward optimization of non-functional properties in software product lines.* *Software Quality Journal* 20(3-4), pp. 487–517, doi:10.1007/s11219-011-9152-9. Available at `http://dx.doi.org/10.1007/s11219-011-9152-9`.

[21] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur & Yarden Katz (2007): *Pellet: A practical OWL-DL reasoner.* *Web Semant.* 5(2), pp. 51–53, doi:10.1016/j.websem.2007.03.004.

[22] P. Trinidad (2012): *Automating the Analysis of Stateful Feature Models.* Ph.D. thesis, University of Seville, http://www.lsi.us.es/ trinidad/docs/tesis.pdf.

[23] Pablo Trinidad, David Benavides & Antonio Ruiz Cortés (2006): *Isolated Features Detection in Feature Models.* In: *The 18th Conference on Advanced Information Systems Engineering (CAiSE'06), Forum Proceedings, Theme: Trusted Information Systems, Luxembourg, June 5-9, 2006.* Available at `http://www.ceur-ws.org/Vol-231/Paper19.pdf`.

[24] Markus Triska (2012): *The Finite Domain Constraint Solver of SWI-Prolog.* In Tom Schrijvers & Peter Thiemann, editors: *Functional and Logic Programming*, *Lecture Notes in Computer Science* 7294, Springer

Berlin Heidelberg, pp. 307–316, doi:10.1007/978-3-642-29822-6_24. Available at http://dx.doi.org/10.1007/978-3-642-29822-6_24.

[25] Markus Triska (2014): *Correctness Considerations in CLP(FD) Systems*. Ph.D. thesis, Vienna University of Technology.

[26] Edward P. K. Tsang (1993): *Foundations of constraint satisfaction*. Computation in cognitive science, Academic Press.

[27] Yingfei Xiong, Arnaud Hubaux, Steven She & Krzysztof Czarnecki (2012): *Generating Range Fixes for Software Configuration*. In: *ICSE '12*, pp. 58–68, doi:10.1109/ICSE.2012.6227206.

[28] Lamia Abo Zaid, Frederic Kleinermann & Olga De Troyer (2009): *Applying Semantic Web Technology to Feature Modeling*. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, ACM, New York, NY, USA, pp. 1252–1256, doi:10.1145/1529282.1529563. Available at http://doi.acm.org/10.1145/1529282.1529563.

[29] Wei Zhang, Haiyan Zhao & Hong Mei (2011): *Binary-Search Based Verification of Feature Models*. In Klaus Schmid, editor: *Top Productivity through Software Reuse*, *Lecture Notes in Computer Science* 6727, Springer Berlin Heidelberg, pp. 4–19, doi:10.1007/978-3-642-21347-2_2. Available at http://dx.doi.org/10.1007/978-3-642-21347-2_2.