Conflict Detection for Edits on Extended Feature Models using Symbolic Graph Transformation

Frederik Deckwerth Géza Kulcsár Malte Lochau Gergely Varró

Andy Schürr

TU Darmstadt
Real-Time Systems Lab

{frederik.deckwerth|geza.kulcsar|malte.lochau|gergely.varro|andy.schuerr}@es.tu-darmstadt.de

Feature models are used to specify variability of user-configurable systems as appearing, e.g., in software product lines. Software product lines are supposed to be long-living and, therefore, have to continuously evolve over time to meet ever-changing requirements. Evolution imposes changes to feature models in terms of edit operations. Ensuring consistency of concurrent edits requires appropriate conflict detection techniques. However, recent approaches fail to handle crucial subtleties of extended feature models, namely constraints mixing feature-tree patterns with first-order logic formulas over non-Boolean feature attributes with potentially infinite value domains. In this paper, we propose a novel conflict detection approach based on symbolic graph transformation to facilitate concurrent edits on extended feature models. We describe extended feature models formally with symbolic graphs and edit operations with symbolic graph transformation rules combining graph patterns with first-order logic formulas. The approach is implemented by combining eMoflon with an SMT solver, and evaluated with respect to applicability.

1 Introduction

In many of nowadays' application domains, software systems must be extensively user-configurable in order to meet diverse customer needs. The inherent *variability* of those systems imposes new kinds of challenges to developers throughout the entire product life cycle. Software product line engineering constitutes a promising paradigm to cope with the additional complexity arising in variant-rich software systems [11]. As a consequence, software product lines (SPLs) have recently found their way from academia into real-world application domains, such as mobile devices, automotive and security [30]. An SPL comprises a family of similar, yet well-distinguished (software) products, whose commonality and variability are defined in terms of *features*. Each feature, therefore, corresponds to (1) a user-visible configuration option in the problem domain of the SPL, as well as (2) dedicated engineering artifacts within the solution space, composable into automatically derivable implementation variants. Feature models are frequently used to specify the set of relevant features of an SPL, together with dependencies among the features, constraining their valid combinations within product configurations. In particular, FODA feature diagrams are widely used during domain analysis, as they provide an intuitive graphical layout in terms of a tree-like hierarchical structuring of feature nodes [16]. Over 20 years of research has been spent to date on developing techniques for efficiently validating crucial consistency properties of feature models in an automated way [5]. Most of those approaches are limited to feature models with Boolean feature parameters by applying respective constraint-solving capabilities, cf., e.g., [20, 6].

Julia Rubin and Thomas Thüm (Eds.):
7th International Workshop on Formal Methods and Analysis
in Software Product Line Engineering (FMSPLE'16)
EPTCS 206, 2016, pp. 17–31, doi:10.4204/EPTCS.206.3

However, various extensions to feature models have been proposed in order to capture all issues that are relevant for the configuration of real-world applications. One major extension of feature models adds complex cross-tree constraints involving non-Boolean feature attributes, e.g., to denote numerical configuration information such as extra-functional properties [24]. Until now, no generally accepted definition of feature models extended with non-Boolean feature attribute constraints exists. Instead, most recent approaches rely on ad-hoc representations of feature attribute constraints, e.g., being restricted to cases that can be encoded into well-known constraint satisfaction problems [6, 17, 9].

Designing an SPL for a particular application domain from scratch usually requires enormous domain analysis efforts and respective upfront investments. Hence, an SPL is supposed to be inherently long-living and, therefore, has to evolve over time to prevent *software aging*, imposed by ever-changing customer needs, platform changes, new legal restrictions etc. [23]. SPL evolution, first of all, induces *edit* operations to feature model specifications by means of changes applied to the feature diagram [19]. Moreover, certain changes that occur frequently during evolution might be extracted as reusable change patterns [25]. In order to ensure consistency and to prevent structural decay, a formal framework for feature model evolution by means of continuous edit operation chains is required. This becomes even more challenging in the presence of non-Boolean feature attribute constraints, as potential conflicts among edit operations involving both feature-tree patterns and complex cross-tree constraints are hard to detect.

In this paper, we propose a formalization of feature models with complex constraints involving non-Boolean feature attributes using symbolic graphs as introduced in [22]. Symbolic graphs constitute a natural extension to typed graphs for a concise representation of modeling languages mixing graphical syntax and first-order logic formulas. This way, we are able to handle feature diagrams with cross-tree constraints incorporating (1) logic formulas combined with arbitrary background theories, e.g., linear and non-linear arithmetics, and (2) attributes with potentially infinite value domains like integers, real numbers, strings, etc. As a consequence, edit operations applied to extended feature models affecting both the feature diagram and the complex cross-tree constraints are seamlessly expressible in terms of symbolic graph transformation rules. Thereupon, we apply a recently proposed conflict notion [18] for symbolic graph transformation rules to ensure consistency of concurrent feature model edits.

To summarize, we make the following contributions:(i) We formalize extended feature models using symbolic graphs, naturally integrating graph patterns and logic formulas. (ii) We define edit operations on extended feature models by means of symbolic graph transformation rules. (iii) We apply a recent conflict detection notion to analyze potential conflicts among concurrent edits. (iv) We evaluate the applicability of our approach, based on an implementation combining our graph transformation tool eMoflon with the Z3 SMT solver.

2 Background and Motivation

2.1 Feature Models

The variability of an SPL is commonly expressed in terms of features. A *feature* represents a distinct user-visible configuration option in the problem domain of an SPL [11]. A *feature model* is used to restrict the possible feature combinations by introducing logical dependencies among features. A feature combination that fulfills these dependencies is called a *valid configuration*. The *configuration space* is the set of all valid configurations defined by a feature model. Feature models are frequently denoted graphically as FODA feature diagrams [16], which organize features into a hierarchical tree structure that can be enriched with additional cross-tree edges between features.

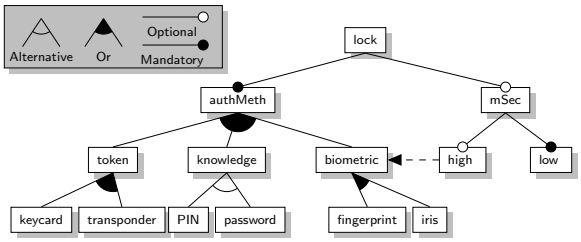


Figure 1: Feature model of an SPL for an electronic lock

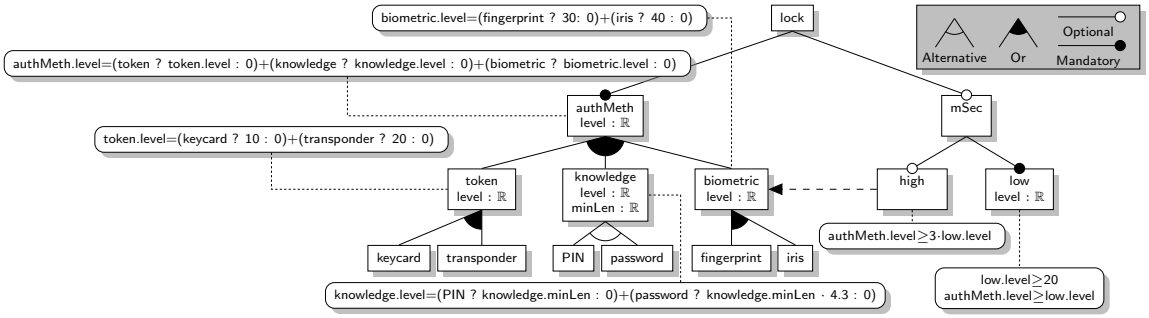


Figure 2: Extended feature model of an SPL for an electronic lock

Figure 1 shows a feature model representing an electronic lock in FODA notation. The feature lock can be equipped with different authentication devices for unlocking: token-based devices using a keycard or a transponder, knowledge-based devices for entering a PIN or a password, as well as biometric devices using fingerprint or iris-scan. The *parent-child relationship* between feature nodes induces configuration constraints, i.e., the selection of a child feature requires the selection of its parent feature. Sibling features may be arranged in *groups* introducing further restrictions on valid combinations.

- An *alternative-group* ensures that exactly one of the child features is present whenever the parent feature is present in a configuration (e.g. either PIN or password).
- An *or-group* claims that at least one child feature has to be present if the parent feature is present in a configuration. In the example, an or-group is used to describe that at least a token-based, a knowledge-based or a biometric device has to be present in a valid lock.
- An *optional-group* consists of a single child feature that may be selected if the parent feature is selected in a configuration. For example, the feature mission security (mSec) may be selected to ensure that a valid lock configuration complies with high or low security regulations.
- A *mandatory-group* also consists of one child feature and requires this feature to be present whenever its parent feature is present, e.g., a lock always has an authentication method (authMeth).

In addition, *cross-tree edges* define dependencies between hierarchically unrelated features, e.g., in terms of binary *require* and *exclude* edges. For example, if the feature high is present in a configuration, a biometric authentication device must be part of the configuration, too.

2.2 Extended Feature Models

In many real-world applications, it is often required to capture configuration options going beyond Boolean features of FODA feature diagrams. For example, it might be desirable to explicitly include the security levels of different authentication devices into the configuration process by introducing numerical

measures for these characteristics. To handle extensions like non-Boolean configuration information, *extended feature models* (EFMs) have been proposed [11]. EFMs extend basic FODA feature diagrams by feature attributes to augment features with additional non-Boolean configuration information. A further extension of EFMs is feature multiplicity denoted by cardinality annotations, which is out of scope here.

A *feature attribute* has a possibly infinite value domain. As a generalization of binary cross-tree edges, dependencies among feature attributes are denoted as complex cross-tree constraints. A *complex cross-tree constraint* is expressed by a first-order logic formula over features and feature attributes that has to evaluate to *true* in a valid configuration. A *valid EFM configuration* consists of two parts: (i) a feature selection corresponding to a valid configuration and (ii) a value assignment to the attributes of the selected features, satisfying all complex cross-tree constraints relevant for those attributes.

In Figure 2, the EFM of the electronic lock is presented, extending the feature diagram from Figure 1 with feature attributes and complex cross-tree constraints. Therein, feature attributes together with their value domains are shown in the boxes of their corresponding feature nodes. We denote complex cross-tree constraints as rounded boxes containing the first-order logic formulas. In our example, each complex cross-tree constraint is related to a feature and has to evaluate to *true* only if its related feature is selected in a configuration. This relation may be expressed by adding further guard expressions over the related features to the constraint [17, 6, 9]; however, for better readability, we omit those additional clauses in our example, but rather indicate those relationships by dashed lines. In particular, the feature knowledge is extended by a feature attribute minLen, the value of which can be explicitly selected by the user during configuration to adjust the minimal length required for a PIN or a password. In addition, each feature representing an authentication method is extended with a real-valued feature attribute level. In contrast to minLen, the values of those feature attributes are not explicitly assignable by the user during configuration, but are rather implicitly derived from the selected authentication devices through complex cross-tree constraints. The complex cross-tree constraints in Figure 2 restrict the accumulated level value of the selected authentication devices (≥ 20 if only low is configured, $3 \cdot$ low.level if high is selected). As feature attributes are only present in a configuration if the corresponding feature is selected, we use a *default expression* to define values for the case if the corresponding feature is not selected in a configuration. For example, the default expression token ? token.level : 0 evaluates to the value of token.level if the feature token is selected and it evaluates to 0 otherwise. By using complex cross-tree constraints, subtle dependencies among features and their attributes are expressible. As an example, a lock with a single keycard device is only valid if the option mission security (mSec) is unselected. This is because authMeth.level would evaluate to 10 which is smaller than the security level 20 required by the constraints low.level ≥ 20 and authMeth.level $\geq$ low.level. As a further example, if high mission security is selected, the lock requires at least one further device in addition to a biometric device in order to fulfill the complex cross-tree constraint authMeth.level $\geq 3 \cdot$ low.level.

2.3 Concurrent Evolution of Extended Feature Models

As illustrated by our running example, the non-Boolean feature attributes and their corresponding cross-tree constraints add additional complexity of EFM compared to FODA diagrams, to be handled by the product-line engineer. This gets even worse by the fact that a product line is meant to be long-living, thus, requiring continuous feature model evolution [19]. The respective evolution steps (e.g., reflecting change requests from the customer, etc.) lead to edit operations on feature models for adding, removing or changing features, attributes and constraints [29].

Figures 3 and 4 show examples for three different feature model edits performed on an excerpt of the EFM presented in Figure 2, referred to as *FM*. The intention of the edit (a) (shown on the left of

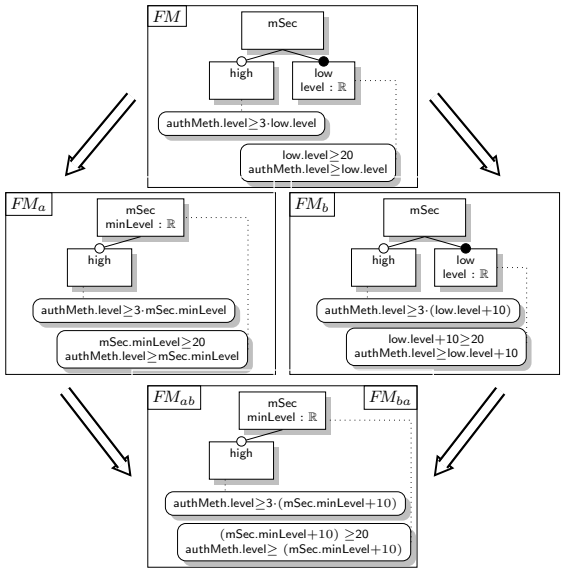


Figure 3: Non-conflicting evolutions of lock EFM

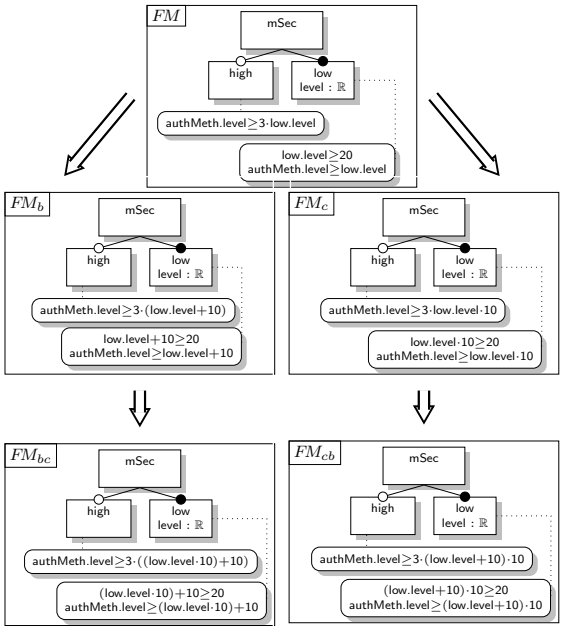


Figure 4: Conflicting evolutions of lock EFM

Figure 3) is to replace the low feature and its level feature attribute by a new feature attribute minLevel with the same assertions, but assigned to the feature mSec. The resulting EFM FM_a (left of Figure 3) is a more compact EFM, with a similar configuration semantics as FM. The edit (b) (right of Figure 3) adds a constant value 10 to the value of low.level. The resulting EFM is FM_b (right of Figure3). The aim of the edit (c) (on the right of Figure 4) is to refactor the feature model by scaling a feature attribute value without altering the conditions imposed by the other complex cross-tree constraints. The result of scaling the low.level feature attribute by factor 10 is FM_c, shown on the right of Figure. 4. In practice, evolution scenarios of extended feature models usually comprise many of such presumably local edits, potentially even applied concurrently by different stakeholders [1]. It is inevitable to provide means that helps the developers to distinguish between concurrent edits whose results can be merged without obstructing feature diagram consistency, and those being potentially conflicting and require manual intervention. This situation gets even more demanding in the presence of complex cross-tree constraints, as handling concurrent EFM edits involves both feature tree patterns as well as logical reasoning on complex cross-tree constraints. For example, the edits (a) and (b) can be merged such that they result in the same EFM FM_{ab}, independent of their application order. This is not obvious because, after edit (a), the low.level attribute is removed and edit (b) is applied to the new mSec.minLevel attribute to obtain the same result. On the contrary, edits (b) and (c) are not arbitrarily serializable and, thus, manual prioritization might be required.

3 Symbolic Graphs and Graph Transformation for EFM Evolution

In this section, we propose the use of a formal, symbolic technique to describe extended feature models and their evolution to reason about the consistency of concurrent edit operations performed on arbitrary

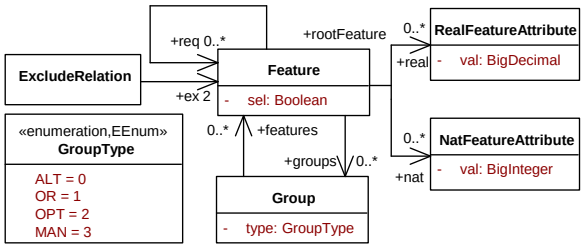


Figure 5: The EFM metamodel

extended feature models. The modeling process consists of the following two phases: (i) Extended feature models are described as symbolic graphs [22], which combine a graph with a first-order logic formula, to concisely represent the feature tree structure together with the complex cross-tree constraints (Section 3.1). (ii) Edit operations on extended feature models are formalized as symbolic graph transformation rules, which provide a declarative technique to manipulate symbolic graphs (Section 3.2).

3.1 Defining Extended Feature Models as Symbolic Graphs

To define a graph-based modeling language for EFM, we specify a metamodel for feature models with feature attributes. We extend this language with first-order logic for complex cross-tree constraints. A *metamodel* defines the abstract syntax for a graph-based modeling language. Figure 5 shows the *EFM metamodel* denoted as a class diagram that provides the building blocks for specifying EFMs. Nodes in the metamodel define *classes* (e.g., Feature). Classes can have *attributes* of a certain *sort* specifying the value domain of the attribute (e.g., attribute sel is of sort Boolean). Classes can be connected by *associations* which are denoted as arrows. Associations have *cardinalities* to restrict the number of participating instances. The EFM metamodel (Figure 5) contains the class Feature, which has an attribute sel of sort Boolean representing whether a feature is selected in a configuration. Groups are modeled by the class Group, which has an attribute type of enumeration-sort GroupType that defines the type of the group, i.e., alternative (ALT), or (OR), optional (OPT), or mandatory (MAN). The classes RealFeatureAttribute and NatFeatureAttribute for modeling feature attributes have an attribute val for representing the infinite domain of real and natural numbers, respectively. Further classes for feature attribute domains may be added to the metamodel if required, e.g., StringFeatureAttribute. The class Group has association features to the contained child features. A parent feature is modeled by association groups to the contained groups. The association req is used to define *require* edges. The *exclude* edges are modeled by the class ExcludeRelation that has the association ex to the two excluding features.

A diagram specified in the modeling language defined by a metamodel is referred to as an *instance model of the metamodel*. The nodes and edges in an instance model define *objects* and *links*, being instances of classes and associations of the metamodel defining their corresponding *types*. Instances of attributes are called *attribute slots* providing a location (e.g., in the memory) for a concrete *attribute value* of the domain defined by the sort of the corresponding attribute in the metamodel. By defining instance models of the EFM metamodel, we can model configurations of an EFM by creating instances of Feature, Group, Real- and NatFeatureAttribute, etc., and assigning attribute values to each attribute slot. However, to represent an EFM instance prior to its configuration, attribute values must be defined symbolically in terms of logic formulas, rather than by concrete value assignments. This leads to the notion of symbolic instance models. A *symbolic instance model* constitutes a natural extension of models by combining their graph-based syntax with first-order logic formulas. We denote a symbolic instance

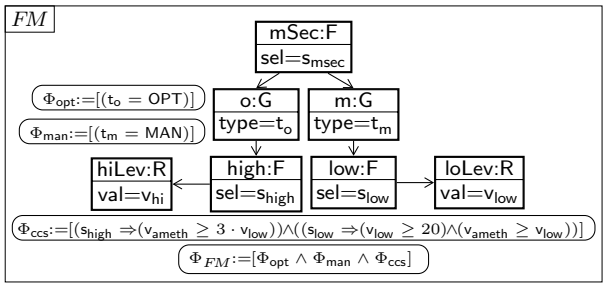


Figure 6: Symbolic instance model of lock EFM

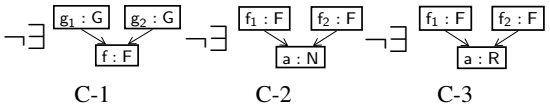


Figure 7: EFM well-formedness constraints

model as a pair $\langle M, \Phi_M \rangle$ consisting of an instance model M, whose attribute values are replaced by *variables*, and a first-order logic formula Φ_M to constrain those variables.

Figure 6 shows a symbolic instance model representing an excerpt of the EFM shown in Figure 2. The model is denoted as an object diagram where objects are represented as nodes labeled with an identifier for the object followed by a colon and the corresponding class of the object. Attribute slots of an object are denoted below the horizontal line and are labeled with the corresponding attribute followed by the assigned variable. Due to space restrictions, we abbreviate class and object names, where F stands for Feature, G for Group, R for RealFeatureAttribute and ameth for authMeth. The corresponding types of the links are not shown as they are unambiguously defined by the classes of the source and target objects. For better readability, the formula Φ_{FM} is partitioned in the conjunction of the terms Φ_{opt}, Φ_{man}, and Φ_{ccs}, where $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ denotes the logical connectives as usual. The term Φ_{opt} sets the types of the group o to optional (OPT) and the term Φ_{man} sets the type of group m to mandatory (MAN). The complex cross-tree constraints are defined in Φ_{ccs}. To guarantee that a symbolic model of an EFM is well-formed, we define further well-formedness constraints to forbid certain patterns in a symbolic instance model. The *EFM well-formedness constraints* are shown in Figure 7:

C-1 No Feature is contained in two Groups.

C-2 No NatFeatureAttribute is contained in two different Features.

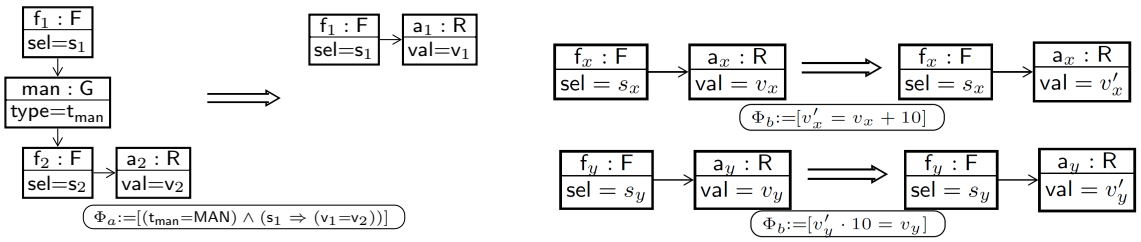
C-3 No RealFeatureAttribute is contained in two different Features.

In general, a well-formedness constraint $\neg\exists C$ consists of a *pattern* C. A symbolic EFM instance $\langle FM, \Phi_{FM} \rangle$ satisfies a well-formedness constraint $\neg\exists C$ if no *matching* for C in FM exists, i.e., no mapping of the elements of C to the elements of a subgraph of FM with identical graph structure and types. For instance, the symbolic EFM instance model in Figure 6 is well-formed.

3.2 Defining EFM Edits by Symbolic Graph Transformation

We define EFM edits by the rule-based technique of graph transformation. Graph transformation (GT) provides a pattern-based manipulation of graph-based models [12]. Applying a transformation rule to an instance model replaces a part of that instance model. A *symbolic GT rule* $r = (LHS, RHS, \Phi)$ consists of a *left-hand side* pattern (*LHS*), a *right-hand side* pattern (*RHS*) and a first-order logic formula Φ [22]. The *LHS* of a rule defines the application context to be matched in a symbolic instance model for applying the rule. Figure 8 depicts the *EFM edit rules* specifying the symbolic graph transformation rules for the EFM edits presented in Section 2.3. The *LHS* pattern of rule r_a, depicted in Figure 8a (left to the arrow), specifies that a symbolic instance model has to contain at least one feature (to match f_1) that is parent of a group (to match man) that has a child feature (to match f_2) with a RealFeatureAttribute (to match a_2).

The application of a rule r at a matching of the *LHS* in a symbolic instance model replaces the



(a) Edit rule r_a

(b) Edit rules r_b (top) and r_c (bottom)

Figure 8: The EFM edit rules

matching of the *LHS* by the *RHS* and adding the formula of the rule by conjuncting its image with the formula of the model. More specifically, a rule r is applied at a matching m to a symbolic instance model $\langle M, \Phi_M \rangle$ leading to the symbolic instance model $\langle M', \Phi'_M \rangle$ (denoted as $\langle M, \Phi_M \rangle \overset{r@m}{\Longrightarrow} \langle M', \Phi'_M \rangle$) by

(i) removing elements from M that can be mapped to *LHS* but not to *RHS*,
(ii) adding elements that can be mapped to *RHS* but not to *LHS* resulting in the model M', and
(iii) constructing the resulting formula Φ'_M as the conjunction $\Phi_M \wedge \sigma'(\Phi)$ of the formula Φ_M and the image $\sigma'(\Phi)$ of the formula Φ that is obtained by substituting the variables in Φ according to the mapping of the variables from *RHS* to M'.

If the resulting formula Φ'_M is satisfiable and the resulting model M' does not contain *dangling links* (i.e., links whose source or target object was deleted), the rule application returns the symbolic instance model $\langle M', \Phi'_M \rangle$ and the rule application is invalid otherwise. Figure 9 shows the result of applying edit rule r_a (Figure 8a) to the EFM *FM* (Figure 6). The rule r_a is applied at the matching given by the following object mapping: $(\mathsf{f}_1 \to \mathsf{mSec})$, $(\mathsf{man} \to \mathsf{m})$, $(\mathsf{f}_2 \to \mathsf{low})$, and $(\mathsf{a}_2 \to \mathsf{loLev})$. The rule is applied by

(i) removing objects m, low, and lowLev and their links and attribute slots with no image in *RHS*,
(ii) creating a new RealFeatureAttribute minLevel and assigning the new variable v_{min} to minLevel.val,
(iii) obtaining new formula Φ_{FM_a} as conjunction of Φ_{FM} and $\sigma'(\Phi_a)$.

The formula $\sigma'(\Phi_a)$ is obtained by substituting the variables in the formula Φ_a of the rule according to the following mapping: $(s_1 \to s_{\mathsf{msec}})$, $(t_{\mathsf{man}} \to t_{\mathsf{m}})$, $(s_2 \to s_{\mathsf{low}})$, and $(v_2 \to v_{\mathsf{low}})$. The rule application is valid as there are no dangling links and the resulting formula Φ_{FM_a} is satisfiable. Note that although we delete the objects with their attribute slots, variables and their corresponding formulas are not deleted. Attribute values are changed by assigning a new variable to an attribute slot, whose value is defined by adding a new first-order clause. Based on the representation of EFM edits as symbolic graph transformation rules, we can apply a recently proposed criterion [18] to detect conflicting pairs of edit operations. Two (edit) rules r_1 and r_2 are *non-conflicting* if

(i) for all symbolic models M and all matchings m_1 and m_2, such that rules r_1 and r_2 can be applied to M, the rule applications lead to the symbolic models M_1 and M_2, respectively, and
(ii) there exist matchings m'_1 and m'_2 for applying r_1 to M_2 and r_2 to M_1, such that the resulting sequences $M \overset{r_1@m_1}{\Longrightarrow} M_1 \overset{r_2@m'_2}{\Longrightarrow} M_{12}$ and $M \overset{r_2@m_2}{\Longrightarrow} M_2 \overset{r_1@m'_1}{\Longrightarrow} M_{21}$ lead to equivalent results M_{12} and M_{21}.

Two rules are *conflicting* if there exists a model and matchings with no such sequences.

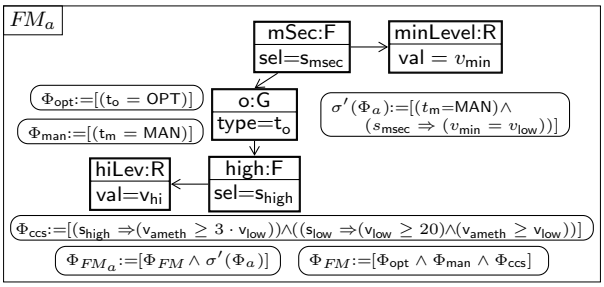


Figure 9: The result FM_a, from applying rule r_1 to FM

4 A Conflict Detection Approach

In this section, we present a conflict detection approach that operates on rules at specification time by performing an analysis on all pairs of rules and categorizing them as conflicting or non-conflicting. The analysis is carried out as follows: (i) For each pair of rules, (symbolic) instance models (i.e., minimal application contexts [14]) are constructed by gluing together the left-hand sides of the rules along all their possible type-conforming subgraphs. These instance models are considered as representatives for a possible conflict (Section 4.1). (ii) The pair of rules under investigation is applied in both possible orders on each instance model representative (minimal application context), and the equivalence of the results of these rule application sequences is checked. If any of these checks fails (i.e., reveals the non-equivalence of the resulting instance models), then the analyzed pair of rules is conflicting (Section 4.2).

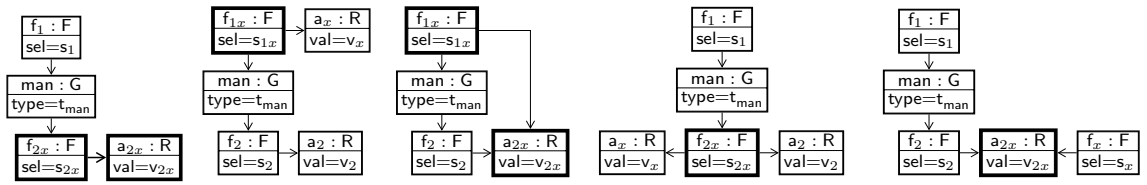
4.1 Constructing Minimal Instance Models

For each pair of rules, we construct instance models as minimal application contexts by gluing together the left-hand sides of the rules along all their possible type-conforming subgraphs. Formally, a *minimal application context* (AC, m_1, m_2) *of rules* $r_1 = (LHS_1, RHS_1, \Phi_1)$ *and* $r_2 = (LHS_2, RHS_2, \Phi_2)$ is a minimal instance model AC together with the matchings m_1 and m_2 of the left-hand sides LHS_1 and LHS_2 to the minimal application context AC, respectively. The construction process might produce minimal instance models that violate the EFM well-formedness constraints. These invalid instance models are filtered out and not considered anymore in the further analysis. Figure 10 shows all minimal application contexts for the rules r_a and r_b constructed as possible gluings (shown by the bold framed elements) of the left-hand sides LHS_a and LHS_b. For example, the minimal instance model AC^a (Figure 10a) is built by gluing the nodes f_2 and a_2 (and the edge between them) in the left-hand side LHS_a to the nodes f_x and a_x (and to the corresponding edge) of the left-hand side LHS_b, respectively. The remaining part (i.e., f_1 and man) of that minimal instance model AC^a originates from the left-hand side LHS_a and these elements do not have corresponding elements in LHS_b. The minimal instance model AC^e is not a well-formed EFM as features f_2 and f_x share the same feature attribute a_{2x}, thus, violating the constraint C-3.

4.2 Applying Rules on Minimal Contexts

For each minimal application context $(\langle AC, true\rangle, m_1, m_2)$ of the rules r_1 and r_2, we have to check whether we can derive application sequences

$$\langle AC, true\rangle \overset{r_1@m_1}{\Longrightarrow} \langle AC_1, \Phi_1\rangle \overset{r_2@m_2'}{\Longrightarrow} \langle AC_{12}, \Phi_{12}\rangle \text{ and } \langle AC, true\rangle \overset{r_2@m_2}{\Longrightarrow} \langle AC_2, \Phi_2\rangle \overset{r_1@m_1'}{\Longrightarrow} \langle AC_{21}, \Phi_{21}\rangle$$



(a) AC^a (b) AC^b (c) AC^c (d) AC^d (e) AC^e

Figure 10: Minimal application contexts of the rule r_a and r_b

such that the resulting symbolic instance models $\langle AC_{12}, \Phi_{12}\rangle$ and $\langle AC_{21}, \Phi_{21}\rangle$ are equivalent. If such a sequence exists for all minimal application contexts of the two rules, the rules are non-conflicting.

Deriving application sequences. First, the rules r_1 and r_2 are applied at the matchings m_1 and m_2 to $\langle AC, true\rangle$. If at least one rule application is invalid, the presence of this minimal application context in a model cannot lead to a conflict. Otherwise the rule applications lead to the symbolic instance models $\langle AC_1, \Phi_1\rangle$ and $\langle AC_2, \Phi_2\rangle$. In the second step, the application sequences are derived by finding matchings m_1' and m_2' such that r_1 and r_2 are applicable to $\langle AC_2, \Phi_2\rangle$ and $\langle AC_1, \Phi_1\rangle$, leading to the symbolic instance models $\langle AC_{21}, \Phi_{21}\rangle$ and $\langle AC_{12}, \Phi_{12}\rangle$, respectively. If at least one of the matchings m_1' or m_2' does not exist such that the rules can be applied, r_1 and r_2 are conflicting and the analysis can be stopped.

Checking equivalence of the results. Two symbolic instance models are equivalent if they have isomorphic graph parts and equivalent logic formulas. As the mapping of the graph parts determines the mapping of the variables as well, we have to bind those variables in $\langle AC_{12}, \Phi_{12}\rangle$ and $\langle AC_{21}, \Phi_{21}\rangle$ that are not assigned to any attribute slot and do not originate from $\langle AC, true\rangle$. These *auxiliary variables* can potentially be assigned to any value (i.e., the values are only constrained by the formula) and are bound in the formulas by existential quantification. Figure 11 shows this technique for rules r_a and r_b with minimal application context $(\langle AC^a, true\rangle, m_a, m_b)$. The application sequences

$$\langle AC^a, true\rangle \overset{r_a@m_a}{\Longrightarrow} \langle AC^a_a, \Phi_a\rangle \overset{r_b@m_b'}{\Longrightarrow} \langle AC^a_{ab}, \Phi_{ab}\rangle \text{ and } \langle AC^a, true\rangle \overset{r_b@m_b}{\Longrightarrow} \langle AC^a_b, \Phi_b\rangle \overset{r_a@m_a'}{\Longrightarrow} \langle AC^a_{ba}, \Phi_{ba}\rangle$$

are derived. To compare the resulting symbolic instance models, v_1 and v_c' in $\langle AC^a_{ab}, \Phi_{ab}\rangle$ and $\langle AC^a_{ba}, \Phi_{ba}\rangle$, respectively, have to be bound as they are auxiliary variables, i.e., they are not assigned to an attribute slot and do not appear in $\langle AC^a, true\rangle$. Binding v_1 leads to the expression $\exists v_1 : \Phi_{ab}$ being equivalent to

$$(t_{\text{man}}\text{=MAN}) \wedge (s_1 \Rightarrow (v_x' - 10 = v_{2x})),$$

as there exists a value for v_1 only if $v_1 = v_x' - 10$. Binding v_x' leads to $\exists v_x' : \Phi_{ba}$ equivalent to

$$(t_{\text{man}}\text{=MAN}) \wedge (s_1 \Rightarrow (v_1 = v_{2x} + 10)).$$

Thereupon, we check whether AC^c_{ab} and AC^c_{ba} are isomorphic, which is the case (simply mapping f_1 to f_1 and a_1 to a_1). Based on the mapping of the objects, we can find a variable mapping $\sigma_{ab\to ba}$: $(s_1 \to s_1)$, $(t_{man} \to t_{man})$, $(v_{2x} \to v_{2x})$, and $(v_x' \to v_1)$. In order to show that the formulas are equivalent, we have to check $\sigma_{ab\to ba}(\exists v_1 : \Phi_{ab}) \Leftrightarrow \exists v_x' : \Phi_{ab}$ that is

$$(t_{\text{man}}\text{=MAN}) \wedge (s_1 \Rightarrow (v_1 - 10 = v_{2x})) \Leftrightarrow (t_{\text{man}}\text{=MAN}) \wedge (s_1 \Rightarrow (v_1 = v_{2x} + 10)),$$

which always holds. For the minimal application context AC^b, it can be shown in the same way that no conflict occurs. AC^d does not lead to a conflict as the application of rule r_a is invalid as it would produce

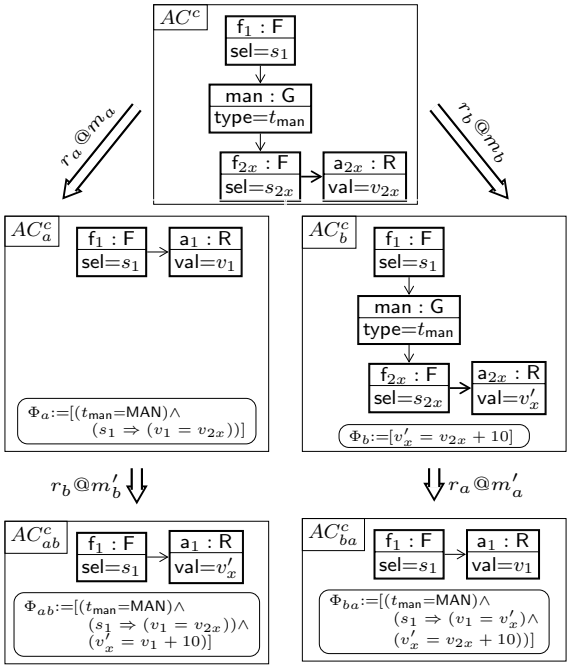


Figure 11: Conflict detection for rules r_a and r_b

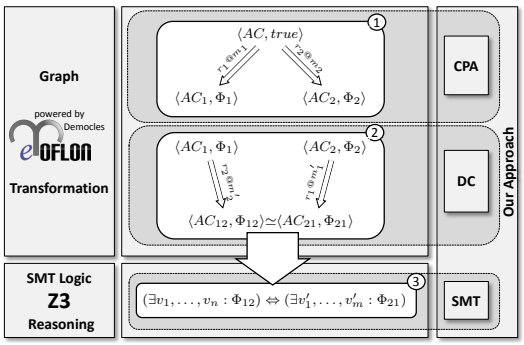


Figure 12: Implementation architecture

dangling links: the link between f_{2x} and a_x becomes dangling after deleting f_{2x} by the rule application. The minimal application contexts AC^c and AC^e are not well-formed as both violate well-formedness constraint C-3, i.e. they share a feature attribute. Consequently, rules r_a and r_b are not conflicting as there exists no minimal application context that causes a conflict. The proposed approach is based on the fact that the application of a graph transformation rule only affects the part of a model that is included in the application context of the rule. In order to ensure that symbolic graph transformation rules applied to symbolic instance models have only local effects, we require that the application of a rule to a symbolic instance model only adds constraints concerning fresh variables. More specifically, a symbolic graph transformation rule $r = (LHS, RHS, \Phi)$ can be handled by our approach if the expression $\forall v_1, \ldots, v_n : \Phi$ is satisfiable, where $v_1, \ldots, v_n$ are those variables appearing in the *LHS* part of the rule. The domain of any fresh variable is unaffected by this restriction. Hence, we can modify the value of any attribute by assigning a fresh variable to the corresponding attribute slot that can be constrained arbitrarily.

5 Implementation and Evaluation

The proposed approach is implemented by combining graph transformation with an SMT solver. We used our model transformation tool eMoflon [3] to apply rules and the Z3 SMT solver [21] to carry out logic reasoning. The Z3 SMT solver supports quantification and equality for non-linear arithmetics over real and natural numbers. Figure 12 shows the architecture of our implementation, which consists of three basic modules. The first module performs *critical pair analysis* (CPA) ①, which is a standard conflict detection technique [12] to derive the initial pair of rule applications. In our approach, CPA serves as a filter to reduce the number of minimal application contexts to be further analysed. The direct confluence (DC) module ② searches for a second pair of rule applications that lead to isomorphic

	r_a	r_b	r_c
r_a	✗		
r_b	(✗)	(✗)	
r_c	(✗)	✗	(✗)

(a) CPA

	r_a	r_b	r_c
r_a	✗		
r_b	✓	✓	
r_c	(✗)	✗	✓

(b) Our approach

Table 1: Conflict detection accuracy results

	CPA [%]	DC [%]	SMT [%]	Total [ms]
$r_a - r_a$	97	3	0	17
$r_a - r_b$	3	31	66	201
$r_a - r_c$	1	1	98	10924
$r_b - r_b$	1	32	67	97
$r_b - r_c$	1	30	69	97
$r_c - r_c$	1	32	67	91

Table 2: Execution times of conflict detection

graphs. If such a second rule application pair is found, the variable mapping is handed over to the SMT module ③ that creates the formula and invokes the SMT solver for the equivalence check. We evaluate our approach by comparing it to CPA, the prevalent standard approach to conflict detection. In our evaluation, the accuracy (i.e., the number of recognized non-conflicting rule pairs) and the execution time of both approaches have been assessed.

Table 1 shows the accuracy results for the two conflict detection approaches. A pair of non-conflicting rules is denoted by ✓, a ✗ marks a conflicting pair of rules, and (✗) denotes false positives, namely, pairs of rules that are recognized as conflicting by the algorithms, although they are non-conflicting according to our definition. The upper right of the tables has been grayed out as conflict analysis is order-insensitive. As shown in Table 1a, CPA categorizes all pairs of the edits r_a, r_b and r_c as conflicting. Note that CPA only considers the graph part and reports a conflict whenever one of the two analyzed edit rules reassigns a variable that is in the matching of the other rule. In this way, CPA is sound (i.e., it recognizes every conflicting pair of rules) but less precise as it does not take the semantics of the changes into account. The accuracy results of our approach are presented in Table 1b. Our technique categorizes 3 pairs of rules as non-conflicting. Only the pair $r_a - r_c$ is incorrectly considered as a conflict. In this case, the SMT solver was unable to determine the equivalence of the formulas, i.e., the SMT solver returns *UNKNOWN* (due to undecidability of first-order logics). Such cases are reported as conflicts (even if edit rules may be non-conflicting) in order to guarantee the soundness of our approach.

Table 2 presents the measured execution times for each pair of edits. Measurements were performed on a computer with an Intel Core i-7-2600-3.4GHz processor. Analysis was run 100 times for each pair of rules. To compensate the just-in-time optimization performed by the Java virtual machine, only the last 50 runs were included in the presented measurement results. The first three columns show the percentage of overall execution time for the three modules, and the last column contains the average of overall execution times in the last 50 runs. The improved conflict detection approach is in average 4 times slower than CPA, where approximately $2/3$ of the additional time is used by the SMT solver. However, the overall execution times are in most cases (except for $r_a - r_c$) below 200 ms. The outlier $r_a - r_c$ is caused by the fact that the SMT solver is unable to check the equivalence of the formulas and timeouts after 10 seconds. The pair $r_a - r_a$ constitutes another extreme, where the most time is spent for CPA and the SMT solver is not even executed. The considerable time spent on CPA can be explained by the fact that the left-hand side of edit rule r_a has the largest pattern in our setup, thus, gluing the pattern with itself produces the highest number of minimal application contexts. For $r_a - r_a$, no logic reasoning is required as accidentally, for the first minimal context, the derivation of the second pair of rule applications is not possible due to dangling links. Thus, the rules are in conflict.

Threats to validity. We conducted our experiments on a selected case study from the security system domain, including sample edit operations observed in a real-world application scenario. However, concerning the significance of those results with respect to other application domains, further experiments

have to be conducted. Regarding the scalability of the approach, the complexity of the underlying analysis problem is mainly caused by the number and size of the rules under consideration. We assume that in a typical application scenario, the number of the rules might be much larger than in our experiments, but the size of the rules is presumably similar. Note that the approach operates on rule level, therefore, its run-time complexity is only influenced by the rule size but is independent of the size of the feature model. Concerning the overall soundness of the approach, our improved conflict detection used in this paper has been proven sound in our previous work [18]. Finally, threats to external validity may arise from the usage of off-the-shelf SMT solver capabilities. However, Z3 is a well-established SMT solver which is widely used in many projects and known for producing reliable results.

6 Related Work

Formalization of Extended Feature Models. Recent approaches for formalizing feature model configuration semantics either rely on translations into equivalent constraint problems, including SAT [4, 20], CSP [6], and BDD [1], or on algebraic representations, e.g., using set theory [27]. Extending feature models with non-Boolean feature attributes and constraints already has been proposed by Kang et al. in the initial FODA feasibility study in [16] and was further elaborated by Czarnecki et al. [11]. In Passos et al. [24], a systematic study on the usage of non-Boolean features in various case studies is presented. However, no generally accepted syntax and semantics for non-Boolean features exist.

Benavides et al. [6] propose a direct translation of feature models with non-Boolean feature attributes into an equivalent CSP representation. In contrast, Bürdek et al. [9], as well as Karatas et al. [17] propose a transformation (of a restricted sublanguage) of non-Boolean feature constraints into Boolean feature model fragments for applying existing constraint solvers [9, 17]. Both approaches are limited to attributes over finite value domains and a restricted set of algebraic operations on attributes within constraints.

Edits on Feature Models. McGregor was one of the first who pointed out the necessity of continuous evolution of software product lines due to their inherently long-living nature [19]. Following this observation, various researchers have proposed approaches for systematically evolving software product lines, starting from changes in terms of edit operations to the underlying feature model. For instance, Elsner et al. identify different types of product-line evolution scenarios based on frequent changes observed in evolving real-world systems [13]. Similar to our approach, the EvoFM approach of Botterweck et al. support the specification and modularization of feature model edits in terms of change rules [8, 7]. In contrast, Seidl et al. define modify patterns on feature models in terms of model deltas and provide a mapping onto solution space artifacts affected by the changes [28]. However, both approaches are limited to predefined collections of basic syntactic edit operations on feature models with no support for Boolean features and respective constraints.

Concerning the semantic impact of feature model changes, the approach of Alves et al. ensures the preservation of feature model configuration semantics by proposing a catalog of sound feature model refactoring patterns [2]. More generally, Thüm et al. present an approach for reasoning about the semantic impact of arbitrary feature model edits using a SAT solver [29]. Henard et al. present a framework for feature model mutation aiming at generating effective product samples for product-line testing [15]. The framework comprises basic mutation operators to inject local changes into the SAT-based representation of feature models for simulating faulty product-line changes. Again, all those approaches are limited to feature models with Boolean features and corresponding constraints. Concerning extended feature models, Quinton et al. recently proposed an approach for ensuring consistency-preserving evolutions of cardinality-based feature models, again, on the basis of a translation into SAT-based representations,

whereas non-Boolean attributes are out of scope [26].

Conflict Detection on Graph Transformation Rules with Attributes. The approach of Cabot et al. [10] presents a fully-fledged graph transformation tool framework which also incorporates a conflict detection technique for attributed graph transformation rules. Nevertheless, the approach is based on a preceding translation of the rules into OCL expressions and, consequently, the used formalism and techniques are not suitable in our symbolic setting. Critical Pair Analysis has been extended to attributed graph transformation on term-attributed graphs in [14]. Although this approach can handle arbitrary attribute domains, the transformation of term-attributed graphs requires term unification to be performed at every derivation step, which restricts the practical applicability of the approach. Contrary, in the symbolic case, the formula is constructed stepwise at the syntactic level and is validated afterwards using SMT solvers.

7 Conclusion

In this paper, we presented a systematic approach for detecting conflicts of concurrent edit operations on extended feature models based on symbolic graph transformation to support the consistent evolution of long-living software product lines. The approach has been implemented by combining the graph transformation tool eMoflon with the Z3 SMT solver. Our experiments show a promising improvement concerning accuracy. We observed a remarkable reduction of false positives compared to a conventional conflict detection approach based on Critical Pair Analysis. For future work, we plan to conduct experiments using larger rule sets gained from an industrial case study from the automation domain [9]. Also, we plan to investigate the possible conflicts of more than two parallel edit operations. In the unattributed case, pairwise analysis suffices, however, it is an open problem in the presence of attributes.

References

[1] Ebrahim Khalil Abbasi, Arnaud Hubaux & Patrick Heymans (2011): *A Toolset for Feature-Based Configuration Workflows*. In: *Proc. of SPLC'11*, IEEE, pp. 65–69, doi:10.1109/SPLC.2011.41.

[2] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba & Carlos Lucena (2006): *Refactoring Product Lines*. In: *Proc. of GPCE'06*, doi:10.1145/1173706.1173737.

[3] A. Anjorin, M. Lauder, S. Patzina & A. Schürr (2011): *eMoflon: Leveraging EMF and Professional CASE Tools*. In: *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*.

[4] Don Batory (2005): *Feature Models, Grammars, and Propositional Formulas*. In: *Proc. of SPLC'05*, pp. 7–20, doi:10.1007/11554844_3.

[5] David Benavides, Sergio Segura & Antonio Ruiz-Cortés (2010): *Automated Analysis of Feature Models 20 Years later: A Literature Review*. *Information Systems* 35, doi:10.1016/j.is.2010.01.

[6] David Benavides, Pablo Trinidad & Antonio Ruiz-Cortés (2005): *Automated Reasoning on Feature Models*. In: *CAiSE*, pp. 491–503, doi:10.1007/11431855_34.

[7] G. Botterweck & A. Pleuss (2014): *Evolution of Software Product Lines*. In: *Evolving Software Systems*, pp. 265–295, doi:10.1007/978-3-642-45398-4_9.

[8] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer & S. Kowalewski (2010): *EvoFM: Feature-driven Planning of Product-line Evolution*. In: *ICSE Workshop on Product Line Approaches in Software Engineering*, ACM, pp. 24–31, doi:10.1145/1808937.1808941.

[9] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz & Andy Schürr (2013): *Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints*. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, ACM, New York, NY, USA, pp. 16:1–16:8, doi:10.1145/2556624.2556627.

[10] Jordi Cabot, Robert Clarisó, Esther Guerra & Juan de Lara (2010): *A UML/OCL Framework for the Analysis of Graph Transformation Rules*. *SoSyM* 9(3), pp. 335–357, doi:10.1007/s10270-009-0129-0.

[11] Krysztof Czarnecki & Ulrich Eisenecker (2000): *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional.

[12] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer, doi:10.1007/3-540-31188-2.

[13] Christoph Elsner, Goetz Botterweck, Daniel Lohmann & Wolfgang Schröder-Preikschat (2010): *Variability in Time - Product Line Variability and Evolution Revisited*. In: *VaMoS'10*, ACM.

[14] Reiko Heckel, Jochen Malte Küster & Gabriele Taentzer (2002): *Confluence of Typed Attributed Graph Transformation Systems*. In: *Proc. of ICGT'02*, Springer, pp. 161–176, doi:10.1007/3-540-45832-8_14.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein & Y. Le Traon (2013): *Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing*. In: *ICST, Verification and Validation Workshops (ICSTW'13)*, pp. 188–197, doi:10.1109/ICSTW.2013.30.

[16] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak & Spencer A. Peterson (1990): *Feature Oriented Domain Analysis (FODA)*. Technical Report, CMU.

[17] Ahmet Serkan Karataş, Halit Oğuztüzün & Ali Doğru (2010): *Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains*. In: *SPLC'10*, Springer, pp. 286–299, doi:10.1007/978-3-642-15579-6_20.

[18] Géza Kulcsár, Frederik Deckwerth, Malte Lochau, Gergely Varró & Andy Schürr (2015): *Improved Conflict Detection for Graph Transformation with Attributes*. In: *Proc. of GaM'15*, EPTCS, pp. 97–112, doi:10.4204/EPTCS.181.7.

[19] John McGregor (2003): *The Evolution of Product Line Assets*. Technical Report CMU/SEI-2003-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[20] Marcílio Mendonça, Andrzej Wasowski & Krzysztof Czarnecki (2009): *SAT-based Analysis of Feature Models is Easy*. In: *13th SPLC*, pp. 231–240, doi:10.1145/1753235.1753267.

[21] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[22] Fernando Orejas & Leen Lambers (2012): *Lazy Graph Transformation*. *Fundam. Inf.* 118(1-2), pp. 65–96, doi:10.3233/FI-2012-706.

[23] David Lorge Parnas (1994): *Software Aging*. In: *Proc. of ICSE'94*, Los Alamitos, CA, USA, pp. 279–287, doi:10.1109/ICSE.1994.296790.

[24] Leonardo Passos, Thorsten Berger, Marko Novakovic, Krzysztof Czarnecki, Yingfei Xiong & Andrzej Wasowski (2011): *A Study of non-Boolean Constraints in Variability Models of an Embedded Operating System*. In: *SPLC WS*, pp. 21–28, doi:10.1145/2019136.2019139.

[25] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba & Jianmei Guo (2015): *Coevolution of Variability Models and Related Software Artifacts*. *Empirical Software Engineering*, pp. 1–50, doi:10.1007/s10664-015-9364-x.

[26] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien & Goetz Botterweck: *Consistency Checking for the Evolution of Cardinality-based Feature Models*. In: *Proc. of SPLC'14*, ACM, pp. 122–131, doi:10.1145/2648511.2648524.

[27] Pierre-Yves Schobbens, Patrick Heymans & Jean-Christophe Trigaux (2006): *Feature Diagrams: A Survey and a Formal Semantics*. In: *Proc. of RE'06*, doi:10.1109/RE.2006.23.

[28] Christoph Seidl, Florian Heidenreich & Uwe Aßmann (2012): *Co-evolution of Models and Feature Mapping in Software Product Lines*. In: *SPLC*, ACM, pp. 76–85, doi:10.1145/2362536.2362550.

[29] Thomas Thüm, Don Batory & Christian Kästner (2009): *Reasoning About Edits to Feature Models*. In: *Proc. of ICSE'09*, IEEE Computer Society, Washington, DC, USA, pp. 254–264, doi:10.1109/ICSE.2009.5070526.

[30] David M. Weiss (2008): *The Product Line Hall of Fame*. In: *SPLC*, IEEE, p. 395, doi:10.1109/SPLC.2008.56.