Synthesis of Attributed Feature Models From Product Descriptions

Guillaume Bécan
Inria - IRISA - University of Rennes 1
France
guillaume.becan@inria.fr

Razieh Behjati
SIMULA
Norway
behjati@simula.no

Arnaud Gotlieb
SIMULA
Norway
arnaud@simula.no

Mathieu Acher
Inria - IRISA - University of Rennes 1
France
mathieu.acher@inria.fr

ABSTRACT

Many real-world product lines are only represented as non-hierarchical collections of distinct products, described by their configuration values. As the manual preparation of feature models is a tedious and labour-intensive activity, some techniques have been proposed to automatically generate boolean feature models from product descriptions. However, none of these techniques is capable of synthesizing feature attributes and relations among attributes, despite the huge relevance of attributes for documenting software product lines. In this paper, we introduce for the first time an algorithmic and parametrizable approach for computing a legal and appropriate hierarchy of features, including feature groups, typed feature attributes, domain values and relations among these attributes. We have performed an empirical evaluation by using both randomized configuration matrices and real-world examples. The initial results of our evaluation show that our approach can scale up to matrices containing 2,000 attributed features, and 200,000 distinct configurations in a couple of minutes.

1. INTRODUCTION

Many real-world product lines are represented as collections of distinct products, each exhibiting specific configuration values. Users can customize or choose their product according to numerous configuration options for satisfying their functional needs without e.g. reaching a maximum budget. Options (also referred as *features* or *attributes*) are ubiquitous and may refer to functional or non-functional aspects of a system, at different level of granularity – from parameters in a function to a whole service.

Modeling features or attributes of a given set of products is a crucial activity in software product line engineering. The formalism of *feature models* (FMs) is widely employed for this purpose [2, 8, 21]. FMs delimit the scope of a family of related products (i.e., an SPL) and formally document what combinations of features are supported. Once specified, FMs can be used for model checking an SPL [28], automating product configuration [19], computing relevant information [8] or communicating with stakeholders [10]. In many generative or feature-oriented approaches, FMs are central for deriving software-intensive products [2]. *Feature attributes* are a useful extension, intensively employed in practice, for documenting the different values across a range of products [4, 8, 12]. With the addition of attributes, optional behaviour can be made dependent not only on the presence or absence of features, but also on the satisfaction of constraints over domain values of attributes [13]. Recently, languages and tools have emerged to fully support attributes in feature modeling and SPL engineering (e.g., see [4, 8, 12, 17, 19]).

The manual elaboration of a feature model – being with attributes or not – is known to be a daunting and error-prone task [1, 3, 5, 15, 16, 18, 20, 23–27]. The number of features, attributes, and dependencies among them can be very important so that practitioners can face severe difficulties for accurately modeling a set of products. In response, numerous *synthesis* techniques have been developed for synthesising feature models [5, 15, 20, 23, 26, 27, 27]. Until now, the impressive research effort has focused on synthesizing basic, Boolean feature models – without feature attributes. Despite the evident opportunity of encoding quantitative information as attributes, the synthesis of attributed feature models has not yet caught attention.

Developing techniques for synthesizing attributed feature models requires to cope with extra complexity. A synthesis algorithm must decide what becomes a feature, what becomes an attribute and identify the domain of each attribute. Compared to features, the domain of an attribute may contain more than two values. Moreover, the placement of the attributes further increases the number of possible hierarchies in a feature model. Finally, cross-tree constraints over attributes are now possible; this level of expressiveness (beyond Boolean logic) challenges synthesis techniques.

In this paper, we develop the theoretical foundations and algorithms for synthesizing attributed feature models given a set of product descriptions. We present parametrizable, tool-supported techniques for computing hierarchies, feature groups, placements of feature attributes, domain values, and constraints. We describe algorithms for comprehensively[1] computing logical relations between features and attributes. The synthesis is capable of taking knowledge (e.g., about the

[1]The interest reader can find more details – including proofs of soundness and completeness, theoretical complexity – in the technical report accompanying the paper [6].

hierarchy and placement of attributes) into account so that users can specify, if needs be, a hierarchy or some placements of attributes. Our work both strengthens the understanding of the formalism and provides a tool-supported solution.

Furthermore we perform a complexity analysis of our synthesis procedure with regards to the number of configurations, features, attributes, and domain values. We evaluate the practical scalability of the synthesis using randomized configuration matrices and real-world examples. Our results show that our approach can scale up to matrices containing 2,000 attributed features, and 20,000 distinct configurations in less than a couple of minutes.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 further motivates the need of synthesising attributed FMs. Section 4 exposes the problem of synthesizing attributed FMs. Section 5 presents our algorithm targeting this problem. Section 6 evaluates the synthesis techniques from a practical point of view. In Section 8 we discuss threats to validity. Section 9 summarizes the contributions and describes future work.

2. RELATED WORK

Numerous works address the synthesis or extraction of FMs. Despite the availability of some tools and languages supporting attributes, no prior work consider the synthesis of attributed FMs; they solely focus on Boolean FMs.

Techniques for synthesising an FM from a set of dependencies (e.g., encoded as a propositional formula) or from a set of products (e.g., encoded in a matrix) have been proposed [15, 20, 22, 26, 27]. In [27], the authors calculate a diagrammatic representation of all possible FMs from a propositional formula (CNF or DNF). In [5], we propose a set of techniques for synthesizing FMs that are both correct w.r.t input propositional formula and present an appropriate hierarchy. The algorithms proposed in [20, 23] take as input a set of configurations. As in the case of [5, 20, 22, 26, 27], the considered configurations only contain Boolean values. Furthermore the generated feature diagram may be an over-approximation of the input configurations. Our work aims to study whether similar properties arise in the context of attributed feature models.

She *et al.* [26] proposed heuristics to synthesize an FM presenting an appropriate hierarchy. Janota *et al.* [22] developed an interactive editor, based on logical techniques, to guide users in synthesizing an FM from a propositional formula. In prior works [5], we develop techniques for taking the so-called ontological semantics into account when synthesizing feature models. Our work shares the goal of interactively supporting users – this time in the context of synthesizing *attributed* feature models.

Considering a broader view, reverse engineering techniques have been proposed to extract FMs from various artefacts (*e.g.,* product descriptions [3, 16, 18, 25], architectural information [1] or source code [24]). However, they do not support the synthesis of attributes despite the presence of non Boolean data in some of these artefacts. In this paper, we do not consider such a broad view; we focus solely on the synthesis of attributed FMs.

In addition to synthesis techniques, there are numerous existing academic (or industrial) languages and tools for specifying and reasoning about FMs [4, 8, 11–13, 19]. None of the existing tools propose support for synthesizing attributed FMs.

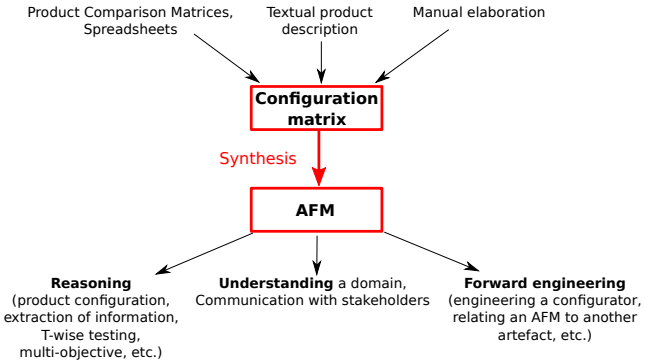


Figure 1: Core problem: synthesis of attributed feature model from configuration matrix

3. MOTIVATION AND BACKGROUND

In this section, we motivate the need for an automated encoding of product descriptions as *attributed feature models (AFMs)*. We also introduce background related to AFMs.

3.1 Product Descriptions and Feature Models

Many modern companies provide solutions for customization and configuration of their products to match the needs of each specific customer. From a user's perspective, it means a large variety of products to choose from. It is therefore crucial for companies not only to provide comprehensive descriptions of their products, but also to do it in an easily navigable manner.

Product descriptions are usually represented in tabular formats, such as spreadsheets and product comparison matrices. The objective of such formats is to describe the characteristics of a set of products in order to document and differentiate them. From now on, we will use the term *configuration matrix* to refer to these tabular formats. Configuration matrices provide an enumerative description of a set of products and are by definition not succinct. Feature models provide an alternative format with a compact and formalized view of a set of products.

Figure 1 summarizes our motivation for synthesizing an AFM from a configuration matrix. As shown in the upper part of Figure 1, the input to the synthesis algorithm is a configuration matrix (see Definition 1).

Definition 1 (Configuration matrix)**.** *Let* $\mathbf{c}_1, ..., \mathbf{c}_M$ *be a given set of configurations. Each configuration* $\mathbf{c}_i$ *is an* N*-tuple* $(c_{i,1}, ..., c_{i,N})$*, where each element* $c_{i,j}$ *is the value of a* variable V_j*. A variable represents either a feature or an attribute. Using these configurations, we create an* $M \times N$ *matrix* $\mathbf{C}$ *such that* $\mathbf{C} = [\mathbf{c}_1, ..., \mathbf{c}_M]^t$*, and call it a* configuration matrix.

Configuration matrices act as a formal, intermediate representation that can be obtained from various sources, such as (1) spreadsheets and product comparison matrices (*e.g.,* see [7]), (2) disjunction of constraints, or (3) simply through a manual elaboration (e.g., practitioners explicitly enumerate and maintain a list of configurations [10]).

For instance, let us consider the domain of Wiki engines. The list of features supported by a set of Wiki engines can be documented using a configuration matrix. Figure 2 is a very simplified configuration matrix, which provides information about eight different Wiki engines.

Id	License Type	License Price	Language Support	Lang uage	WYSI WYG
W1	Commercial	10	Yes	Java	Yes
W2	NoLimit	20	No	–	Yes
W3	NoLimit	10	No	–	Yes
W4	GPL	0	Yes	Python	Yes
W5	GPL	0	Yes	Perl	Yes
W6	GPL	10	Yes	Perl	Yes
W7	GPL	0	Yes	PHP	No
W8	GPL	10	Yes	PHP	Yes

Figure 2: A configuration matrix for Wiki engines.

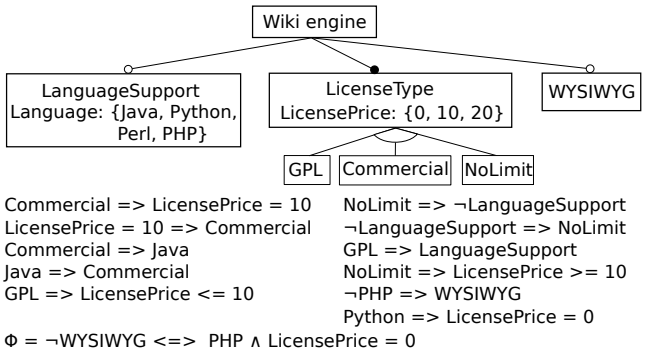


Figure 3: One possible attributed feature model for representing the configuration matrix in Figure 2

The resulting AFM (see lower part of Figure 1) can as well be used to document a set of configurations and open new perspectives. First, state-of-the-art reasoning techniques for AFM can be reused (e.g., [4, 13, 19, 28]). Second, the hierarchy helps to structure the information and a potentially large number of features into multiple levels of increasing detail [14]; it helps to understand a domain or communicate with other stakeholders [8, 10, 14]. Finally, an AFM is central to many product line approaches and can serve as basis for a forward engineering [2] (*e.g.,* through a mapping with source code or design models).

Overall, configuration matrices and feature models are semantically related and aim to characterize a set of configurations. The two formalisms are complementary; we aim to better understand the gap and switch from one representation to the other. For instance, Figure 3 depicts an attributed feature diagram as well as constraints that together provide one *possible* representation of the configuration matrix of Figure 2.

3.2 Attributed Feature Models

Several formalisms supporting attributes exist [4,9,12,17]. In this paper, we consider an extension of FODA-like FMs including attributes and inspired from the FAMA framework [8, 9]. An AFM is composed of a feature diagram (see Definition 2 and Figure 4) and an arbitrary constraint (see Definition 3).

Definition 2 (Attributed Feature Diagram)**.** *An attributed feature diagram FD is a tuple* $\langle F, H, E_M, G_{MTX}, G_{XOR}, G_{OR}, A, D, \delta, \alpha, RC\rangle$ *such that:*

- F *is a finite set of boolean features.*
- $H = (F, E)$ *is a rooted tree of features where* $E \subseteq F \times F$ *is a set of directed child-parent edges.*
- $E_M \subseteq E$ *is a set of edges that define mandatory features.*
- $G_{MTX}, G_{XOR}, G_{OR} \subseteq P(E \backslash E_m)$ *are sets of feature groups. Each feature group is a set of edges. The feature groups of* G_{MTX}, G_{XOR} *and* G_{OR} *are non-overlapping and all edges in a group share the same parent.*
- A *is a finite set of attributes.*
- D *is a set of possible domains for the attributes in* A.
- $\delta \in A \rightarrow D$ *is a total function that assigns a domain to an attribute.*
- $\alpha \in A \rightarrow F$ *is a total function that assigns an attribute to a feature.*
- RC *is a set of constraints over* F *and* A *that are considered as human readable and may appear in the feature diagram in a graphical or textual representation (*e.g., *binary implication constraints can be represented as an arrow between two features).*

A domain $d \in D$ *is a tuple* $\langle V_d, 0_d, <_d\rangle$ *with* V_d *a finite set of values,* $0_d \in V_d$ *the null value of the domain and* $<_d$ *a partial order on* V_d*. When a feature is not selected, all its attributes bound by* α *take their null value,* i.e., $\forall (a, f) \in \alpha$ *with* $\delta(a) = \langle V_a, 0_a, <_a\rangle$*, we have* $\neg f \Rightarrow (a = 0_a)$.

RC	::=	*bool_factor* '⇒' *bool_factor*
bool_factor	::=	**feature_name** | '¬' **feature_name** | *rel_expr*
rel_expr	::=	**attribute_name** *rel_op* **num_literal**
rel_op	::=	'>' | '<' | '≥' | '≤' | '='

Figure 4: The grammar of readable constraints.

For the set of constraints in RC, formally defining what is human readable is essential for automated techniques. In this paper, we define RC as the constraints that are consistent with the grammar in Figure 4. Some examples of such constraints can be found in the bottom of Figure 3. We consider that these constraints are small enough and simple enough to be human readable. In this grammar, each constraint is a binary implication, which specifies a relation between the values of two attributes or features. Feature names and relational expressions over attributes are the boolean factors that can appear in an implication. Further, we only allow natural numbers as numerical literals (num_literal).

The grammar of Figure 4 and the formalism of attributed feature *diagrams* (see Definition 2) are not expressively complete regarding propositional logics, and therefore cannot represent any set of configurations (more details are given in Section 4.2). Therefore, to enable accurate representation of any possible configuration matrix, an AFM is composed of an attributed feature diagram and a propositional formula:

Definition 3 (Attributed Feature Model)**.** *A feature model is a pair* $\langle FD, \Phi\rangle$ *where FD is an attributed feature diagram and* Φ *is an arbitrary constraint over* F *and* A *that represents the constraints that cannot be expressed by RC.*

Example. Figure 3 shows an example of an AFM describing a product line of Wiki engines. The feature *WikiMatrix* is the root of the hierarchy. It is decomposed in 3 features: *LicenseType* which is mandatory and *WYSIWYG* and *LanguageSupport* which are optional. The xor-group composed of *GPL*, *Commercial* and *NoLimit* defines that the wiki engine has exactly 1 license and it must be selected among

these 3 features. The attribute *LicensePrice* is attached to the feature *LicenseType*. The attribute's domain states that it can take a value in the following set: {0, 10, 20}. The readable constraints and Φ for this AFM are listed below its hierarchy (see Figure 3). The first one restricts the price of the license to 10 when the feature *Commercial* is selected.

The main objective of an AFM is to define the valid configurations of a product line. A configuration of an AFM is defined as a set of selected features and a value for every attribute. A configuration is valid if it respects the constraints defined by the AFM (*e.g.,* the root feature of an AFM is always selected). The set of valid configurations corresponds to the *configuration semantics* of the AFM (see Definition 4).

Definition 4 (Configuration semantics)**.** *The configuration semantics $[\![m]\!]$ of an AFM m is the set of valid configurations represented by m.*

4. SYNTHESIS FORMALIZATION

Two main challenges of synthesizing an AFM from a configuration matrix are (1) preserving the configuration semantics of the input matrix; and (2) producing a maximal and readable diagram (for a further exploitation by practitioners, see Figure 1). Synthesizing an AFM that represents the exact same set of configurations (*i.e.,* configuration semantics) as the input configuration matrix is primordial; a too permissive AFM would expose the user to illegal configurations. To prevent this situation, the algorithm must be sound (see Definition 5). Conversely, if the AFM is too constrained, it would prevent the user from selecting available configurations, resulting in unused variability. Therefore, the algorithm must also be complete (see Definition 6).

Definition 5 (Soundness of AFM Synthesis)**.** *A synthesis algorithm is sound if the resulting AFM (afm) represents only configurations that exist in the input configuration matrix (cm),* i.e., $[\![afm]\!] \subseteq [\![cm]\!]$.

Definition 6 (Completeness of AFM Synthesis)**.** *A synthesis algorithm is complete if the resulting AFM (afm) represents at least all the configurations of the input configuration matrix (cm),* i.e., $[\![cm]\!] \subseteq [\![afm]\!]$.

To avoid the synthesis of a trivial AFM (*e.g.,* an AFM with the input matrix encoded in the constraint Φ and no hierarchy, *i.e.,* $E = \emptyset$), we target a maximal AFM as output (see Definition 7). Intuitively, we enforce that the feature diagram contains as much information as possible. Definition 8 formulates the AFM synthesis problem.

Definition 7 (Maximal Attributed Feature Model)**.** *An AFM is maximal if its hierarchy H connects every feature in F and if none of the following operations are possible without modifying the configuration semantics of the AFM:*

- *add an edge to E_M*
- *add a group to G_{MTX}, G_{XOR} or G_{OR}*
- *move a group from G_{MTX} or G_{OR} to G_{XOR}*
- *add a constraint to RC that is not redundant with other constraints of RC.*

TODO (Guillaume): validate precision on rendundancy of RC

The restriction on the redundancy of RC constraints avoids a potentially infinite number of constraints. Without this restriction, a synthesis algorithm would be forced to synthesize all possible constraints respecting the configuration semantics of the AFM.

Definition 8 (Attributed Feature Model Synthesis Problem)**.** *Given a set of configurations sc, the problem is to synthesize an AFM m such that $[\![sc]\!] = [\![m]\!]$ (*i.e., *the synthesis is sound and complete) and m is maximal.*

4.1 Synthesis Parametrization

In Definition 8, we enforce the AFM to be maximal to avoid trivial solutions to the synthesis problem. Despite this restriction, the solution to the problem may not be unique. Given a set of configurations (*i.e.,* a configuration matrix), multiple maximal AFMs can be synthesized.

This property has already been observed for the synthesis of Boolean FMs [5, 26, 27]. Extending boolean FMs with attributes exacerbates the situation. In some cases, the place of the attributes and the constraints over them can be modified without affecting the configuration semantics of the synthesized AFM.

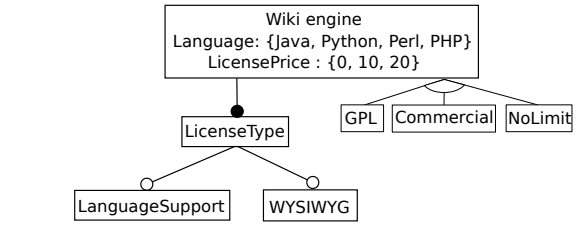
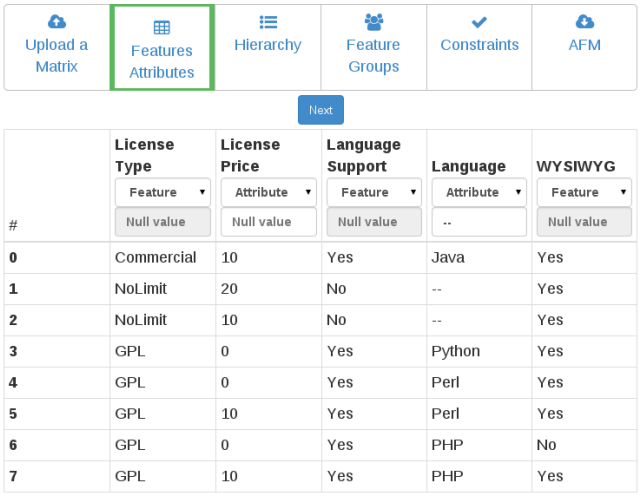


Figure 5: Another attributed feature model representing the configuration matrix in Figure 2

Example. Figures 3 and 5 depict two AFMs representing the same configuration matrix of Figure 2. They have the same configuration semantics but their attributed feature diagrams are different. In Figure 3, the feature *WYSIWYG* is placed under *Wiki engine* whereas in Figure 5, it is placed under the feature *LicenseType*. Besides the attribute *LicensePrice* is placed in feature *LicenseType* in Figure 3, whereas it is placed in feature *Wiki engine* in Figure 5.

To synthesize a unique AFM, our algorithm uses *domain knowledge*, which is extra information that can come from heuristics, ontologies or a user of our algorithm. This domain knowledge can be provided interactively during the synthesis or as input before the synthesis. Our synthesis tool (Figure 6 shows the workflow) provides an interface for collecting the domain knowledge so that users can:

- decide if a column of the configuration matrix should be represented as a feature or an attribute;
- give the interpretation of the cells (type of the data, partial order);
- select a possible hierarchy, including the placement of each attribute among their legal possible positions;



#	License Type (Feature)	License Price (Attribute)	Language Support (Feature)	Language (Attribute)	WYSIWYG (Feature)
	Null value	Null value	Null value	--	Null value
0	Commercial	10	Yes	Java	Yes
1	NoLimit	20	No	--	Yes
2	NoLimit	10	No	--	Yes
3	GPL	0	Yes	Python	Yes
4	GPL	0	Yes	Perl	Yes
5	GPL	10	Yes	Perl	Yes
6	GPL	0	Yes	PHP	No
7	GPL	10	Yes	PHP	Yes

Figure 6: Web-based tool for gathering domain knowledge during the synthesis

- select a feature group among the overlapping ones;
- provide specific bounds for each attribute in order to compute meaningful and relevant constraints for RC.

All steps are optional; in case the domain knowledge is missing, the synthesis algorithm (see next section) takes arbitrary yet sound decisions (e.g., random hierarchy).

Example. The domain knowledge that leads to the synthesis of the AFM of Figure 3 can be collected with our synthesis tool. Users specify what constitute an attribute or a feature. For instance, the column *Language* represents an attribute (for which the null value is "- -"). In further step, users can specify hierarchy and also precise that, *e.g.,* "10" is an interesting value for *LicensePrice* when synthesizing constraints.

4.2 Over-approximation of the Attributed Feature Diagram

A crucial property of the output AFM is to characterize the exact same configuration semantics as the input configuration matrix (see Definition 8). In fact, the attributed feature *diagram* may over-approximate the configuration semantics, i.e., $[\![cm]\!] \subseteq [\![FD]\!]$. Therefore the additional constraint Φ of an AFM (see Definition 3) is required for providing an accurate representation for any arbitrary configuration matrix.

Example. The diagrammatic part of the AFM in Figure 3 characterizes two additional configurations that are not in the matrix of Figure 2: {LicenseType = GPL, LicensePrice = 0, LanguageSupport = Yes, Language = PHP, WYSIWYG = Yes} and {LicenseType = GPL, LicensePrice = 10, LanguageSupport = Yes, Language = PHP, WYSIWYG = No}. To properly encode the configuration semantics of the configuration matrix, the AFM has to rely on a constraint Φ. This particular constraint cannot be expressed by an attributed feature *diagram* as defined in Definition 2. Therefore, if Φ is not computed the AFM would represent an over-approximation of the configuration matrix.

A basic strategy for computing Φ is to directly encode the configuration matrix as a constraint, i.e., $[\![cm]\!] = [\![\Phi]\!]$. Such a constraint can be achieved using the following equation, where N is the number of columns and M is the number of rows.

$$\Phi = \bigvee_{i=1}^{M} \bigwedge_{j=1}^{N} (V_j = c_{ij}) \quad (1)$$

An advantage of this approach is that the computation is immediate and Φ is, by construction, sound and complete w.r.t. the configuration semantics. The disadvantage is that some constraints in Φ are likely to be (1) redundant with the attributed feature diagram; (2) difficult to read and understand for a human.

Ideally, Φ should express the exact and sufficient set of constraints not expressed in the attributed feature diagram, i.e., $[\![\Phi]\!] = [\![cm]\!] \setminus [\![FD]\!]$. Synthesizing a minimal set of constraint may require complex and time-consuming computations. The development of efficient techniques for simplifying Φ and investigating the usefulness and readability of arbitrary constraints[2] are out of the scope of this paper.

5. SYNTHESIS ALGORITHM

Algorithm 1 presents a high-level description of our approach for synthesizing an AFD. The two inputs of the algorithm are a configuration matrix and some domain knowledge for parametrizing the synthesis. The output is a maximal AFD. In complement to the AFD, we compute the constraint Φ (as described by Equation 1). The addition of Φ and the AFD forms the AFM.

Algorithm 1 ATTRIBUTED FEATURE DIAGRAM SYNTHESIS

Require: A configuration matrix **MTX** and domain knowledge **DK**
Ensure: An attributed feature diagram **AFM**

Extract the features, the attributes and their domains
1 $(F, A, D, \delta) \leftarrow$ extractFeaturesAndAttributes(MTX, DK)

Compute binary implications
2 $BI \leftarrow$ computeBinaryImplications(MTX)

Define the hierarchy
3 $(BIG, MTXG) \leftarrow$ computeBIGAndMutexGraph(F, BI)
4 $H \leftarrow$ extractHierarchy(BIG, DK)
5 $\alpha \leftarrow$ placeAttributes(BI, F, A, DK)

Compute the variability information
6 $E_M \leftarrow$ computeMandatoryFeatures(H, BIG)
7 $FG \leftarrow$ computeFeatureGroups$(H, BIG, MTXG, DK)$

Compute cross tree constraints
8 $RC \leftarrow$ computeConstraints(BI, DK, H, E_M, FG)

Create the attributed feature diagram
9 **return** $AFD(F, H, E_M, FG, A, D, \delta, \alpha, RC)$

5.1 Extracting Features and Attributes

The first step of the synthesis algorithm is to extract the features (F), the attributes (A) and their domains (D, δ). This step essentially relies on the domain knowledge to decide how each column of the matrix must be represented as a feature or an attribute. We also discard all the dead

[2]We recall that *relational* constraints as defined by the grammar of Figure 4 are already part of the synthesis. Arbitrary constraints represent other forms of constraints involving more than two features or attributes – hence questioning their usefulness or readability by humans.

features, *i.e.,* features that are always absent. The domain knowledge specifies which values in the corresponding column indicate the presence (*e.g.,* by default "Yes") or absence (*e.g.,* "No") of the feature. For each attribute, all the distinct values of its corresponding column form the first part of its domain (V_d). The other parts, the null value 0_d, and the partial order $<_d$, are computed accordingly. Some predefined heuristics are implemented to populate the domain knowledge and consist in (1) determining whether a column is a feature or an attribute or (2) typing the domain values. If needs be, users can override the default behaviour of the synthesis and specify domain knowledge.

Example. Let us consider the variable *LanguageSupport* in the configuration matrix of Figure 2. Its domain has only 2 possible Boolean values: *Yes* and *No*. Therefore, the variable *LanguageSupport* is identified as a feature. Following the same process, *WYSIWYG* and *LicenseType* are also identified as features while the other variables are identified as attributes. For instance, *LicensePrice* is an attribute and its domain is the set of all values that appear in its column: $\{0, 10, 20\}$.

5.2 Extracting Binary Implications

An important step of the synthesis is to extract binary implications between features and attributes. A binary implication, for a given configuration matrix $\mathbf{C}$, is of the form $(v_i = u) \Rightarrow (v_j \in S_{i,j,u})$, where i and j indicate two distinct columns of $\mathbf{C}$, and $S_{i,j,u}$ is the set of all values in the jth column of $\mathbf{C}$, for which the corresponding cell in the ith column is equal to u. We denote the set of all binary implications of $\mathbf{C}$ as $BI(\mathbf{C})$. Algorithm 2 computes this set. The algorithm iterates over all pairs (i, j) of columns and all configurations c_k in C to compute $S(i, j, c_{k,i})$ (i.e., $S_{i,j,c_{k,i}}$).

Algorithm 2 computeBinaryImplications

Require: A configuration matrix $\mathbf{C}$
Ensure: A set of binary implications BI
1 $BI \leftarrow \emptyset$
2 **for all** (i, j) such that $1 \leq i, j \leq N$ and $i \neq j$ **do**
3 **for all** c_k such that $1 \leq k \leq M$ **do**
4 **if** $S(i, j, c_{k,i})$ does not exists **then**
5 $S(i, j, c_{k,i}) \leftarrow \{c_{k,j}\}$
6 $BI \leftarrow BI \cup \{\langle i, j, u, S(i, j, c_{k,i})\rangle\}$
7 **else**
8 $S(i, j, c_{k,i}) \leftarrow S(i, j, c_{k,i}) \cup \{c_{k,j}\}$
9 **return** BI

In line 4, the algorithm tests if $S(i, j, c_{k,i})$ already exists. If so, the algorithm simply adds $c_{k,j}$, the value of column j for configuration c_k, to the set $S(i, j, c_{k,i})$. Otherwise, $S(i, j, c_{k,i})$ is initialized with a set containing $c_{k,j}$. Then, a new binary implication is created and added to the set BI. At the end of the inner loop, BI contains all the binary implications of all pairs of columns (i, j).

5.3 Defining the Hierarchy

The hierarchy H of an AFD is a rooted tree of features such that $\forall (f_1, f_2) \in E, f_1 \Rightarrow f_2$, *i.e.,* each feature implies its parent. As a result, the candidate hierarchies, whose parent-child relationships violate this property, can be eliminated upfront. To guide the selection of a legal hierarchy for the AFD, we rely on the *Binary Implication Graph* (BIG) of a configuration matrix:

Definition 9 (Binary Implication Graph (BIG))**.** *A binary implication graph of a configuration matrix* $\mathbf{C}$ *is a directed graph* (V_{BIG}, E_{BIG}) *where* $V_{BIG} = F$ *and* $E_{BIG} = \{(f_i, f_j) \mid$ '$f_i \Rightarrow f'_j \in BI(\mathbf{C})\}$.

The BIG aims to represent all the binary implications, thus representing every possible parent-child relationships in a legal hierarchy. We compute the BIG as follows: for each constraint in BI (see Algorithm 2) involving two features we add an edge in the BIG. Hierarchy H of the AFD is a rooted tree inside the BIG. In general, it is possible to find many such trees in a BIG. As part of the tool (see Figure 6), users can specify domain knowledge to select a tree. The BIG is exploited to interactively assist users in the selection of the AFD's hierarchy. In case the domain knowledge is incomplete, a branching of the graph (the counterpart of a spanning tree for undirected graphs) is randomly computed [5].

After choosing the hierarchy of features, we focus on the placement of attributes. An attribute a can be placed in a feature f if $\neg f \Rightarrow (a = 0_a)$. As a result, the candidate features verifying this property are considered as legal positions for the attribute. We promote, according to the domain knowledge, one of the legal positions of each attribute.

Example. In Figure 2, the attribute *Language* has a domain d with "–" as its null value, *i.e.,* $0_d =$ "–". This null value restricts the place of the attribute. The property $\neg f \Rightarrow (a = 0_a)$ of Definition 2 holds for the attribute *Language* and the feature *LanguageSupport*. However, the configuration W7 forbids the attribute to be placed in feature *WYSIWYG*. The value of *Language* is not equal to its null value when *WYSIWYG* is not selected.

5.4 Computing the Variability Information

As the hierarchy of an AFD is made of features only, attributes do not impact the computation of the variability information (optional/mandatory features and feature groups). Therefore, we can rely on algorithms that have been developed for Boolean FMs [5, 26, 27].

First, for the computation of mandatory features, we rely on the BIG as it represents every possible implication between two features. For each edge (c, p) in the hierarchy, we check that the inverted edge (p, c) exists in the BIG. In that case, we add this edge to E_M.

For feature groups, we reuse algorithms from the synthesis of Boolean FMs [5,26,27]. For the sake of self-containedness, we briefly describe the computation of each type of group.

For mutex-groups (G_{MTX}), we compute a so-called *mutex graph* that contains an edge whenever two features are mutually exclusive. The maximum cliques of this mutex graph are the mutex-groups [26, 27].

For or-groups (G_{OR}), we translate the input matrix to a binary integer programming problem [27]. Finding all solutions to this problem results in the list of or-groups.

For xor-groups (G_{XOR}), we consider mutex-groups since a xor-group is a mutex-group with an additional constraint stating that at least one feature of the group should be selected. We check, for each mutex-group, that its parent implies the disjunction of the features of the group. For that, we iterate over the binary implications (BI) until we find that the property is inconsistent. To ensure the maximality of the resulting AFM, we discard any or-group or mutex-group that is also an xor-group.

Finally, the features that are not involved in a mandatory

relation or a feature group are considered optional.

5.5 Computing Cross Tree Constraints

The final step of the AFD synthesis algorithm is to compute cross tree constraints (RC). We generate three kinds of constraints: *requires*, *excludes* and *relational* constraints.

A requires constraint represents an implication between two features. All the implications contained in the BIG (*i.e.,* edges) that are not represented in the hierarchy or mandatory features, are promoted as requires constraints. An example of this is the implication *Commercial* $\Rightarrow$ *Java* in Figure 3.

Excludes constraints represent the mutual exclusion of two features. Such constraints are contained in the mutex graph. As the previously computed mutex-groups may not represent all the edges of the mutex graph, we promote the remaining edges of the mutex graph as excludes constraints. For example, the features *NoLimit* and *LanguageSupport*, in Figure 3, are mutually exclusive. This relation is included in RC as an excludes constraint: *NoLimit* $\Rightarrow \neg$*LanguageSupport*.

Finally, relational constraints are all the constraints following the grammar described in Figure 4 and involving at least one attribute. Admittedly there is a huge number of possible constraints that respect the grammar of RC. Our algorithm relies on some domain knowledge (see Figure 6) to restrict the domain values of attributes considered for the computation of constraints. Formally, the knowledge provides the information required for merging binary implications as (a_i, k) pairs, where a_i is an attribute, and k belongs to D_i. In case the knowledge is incomplete (e.g., users do not specify a bound for an attribute), we randomly choose a value among the domain of an attribute.

Then the algorithm proceeds as follows. First, we transform each constraint referring to one feature and one attribute to a constraint that respects the grammar of RC. Then, we focus on constraints in BI that involve two attributes and we merge them according to the domain knowledge. Using the pairs (a_i, k) of the domain knowledge, we partition the set of all binary implications with $a_i = u$ on the left hand side of the implication into three categories: those with $u < k$, those with $u = k$, and those with $u > k$. Let $b_{j,1}, b_{j,2}, ..., b_{j,p}$ be all such binary implications, belonging to the same category, and involving a_j (i.e., each $b_{j,r}$ is of the form $(a_i = u_r \Rightarrow a_j \in S_r)$). We merge these binary implications into a single one: $(a_i \in \{u_1, u_2, ..., u_p\} \Rightarrow a_j \in S_1 \cup S_2 \cup ... \cup S_p)$ whenever the conformance of the grammar of RC holds.

Example. From the configuration matrix of Figure 2, we can extract the following binary implication: *GPL* $\Rightarrow$ *LicensePrice* $\in \{0, 10\}$. We also note that the domain of *LicensePrice* is $\{0, 10, 20\}$. Therefore, the right side of the binary implication can be rewriten as *LicensePrice* $<= 10$. As this constraint can be expressed by the grammar of RC, we add *GPL* $\Rightarrow$ *LicensePrice* $<= 10$ to RC (see Figure 3).

6. EVALUATION

We developed a tool[3] that implements Algorithm 1. The tool is mainly implemented in Scala programming language. For instance, Algorihtm 2 is written in pure Scala and use appropriate data structures (*e.g.,* HashMap and HashSet) to improve its scalability. For the computation of or-groups, we rely on the SAT4j solver.

TODO (Guillaume): TO VALIDATE : Algorithm 2, Scala, ILP (or-groups), etc.

To provide an insight of the scalability of our procedure, we experimentally evaluate the runtime complexity of our AFD synthesis procedure. For this purpose, we have developed a random matrix generator, which takes as input three parameters:

- number of variables (features and attributes)
- number of configurations
- maximum domain size (*i.e.,* maximum number of distinct values in a column)

The type of each variable (feature or attribute) is randomly selected according to a uniform distribution. An important property is that our generator does not ensure that the number of configurations and the maximum domain size are reached at the end of the generation. Any duplicated configuration or missing value of a domain is not corrected. Therefore, the parameters entered for our experiments may not reflect the real properties of the generated matrices. To avoid any misinterpretation or bias, we present the concrete numbers in the following results.

Moreover, to reduce fluctuations caused by the random generator, we perform the experiments at least 100 times for each triplet of parameters. In order to get the results in a reasonable time, we used a cluster of computers. Each node in the cluster is composed of two Intel Xeon X5570 at 2.93 Ghz with 24GB of RAM.

6.1 Initial Experiments with Or-groups

We first perform some initial experiments on random matrices. We quickly notice that computation of the or-groups posed a scalability problem. It is not surprising since this part of the synthesis algorithm is NP-hard, leading to some timeouts even for Boolean FMs (e.g., see [27]).

Specifically, we measured the time needed to compute the or-groups from a matrix with 1000 configurations, a maximum domain size of 10 and a number of variables ranging from 5 to 70. To keep a reasonable time for the execution of the experiment, we set a timeout at 30 minutes. Results are presented in Figure 7. The red dots indicate average values in each case. The results confirm that the computation of or-groups quickly becomes time consuming. The 30 minutes timeout is reached with matrices containing only 30 variables. With at least 60 variables, the timeout is systematically reached. Therefore, we deactivated the computation of or-groups in the following experiments.

6.2 Scalability w.r.t the number of variables

To evaluate the scalability with respect to the number of variables, we perform the synthesis of random matrices with 1000 configurations, a maximum domain size of 10 and a number of variables ranging from 5 to 2000. In Figure 8, we present the square root of the time needed for the whole synthesis compared to the number of variables. As shown by the linear regression line, the square root of the time grows linearly with the number of variables, with a correlation coefficient of 0.997.

[3]The complete source code can be found in `https://github.com/gbecan/FOReverSE-AFMSynthesis`

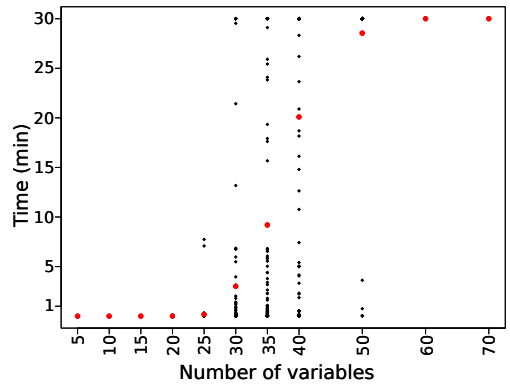


Figure 7: Scalability of or-groups computation w.r.t the number of variables

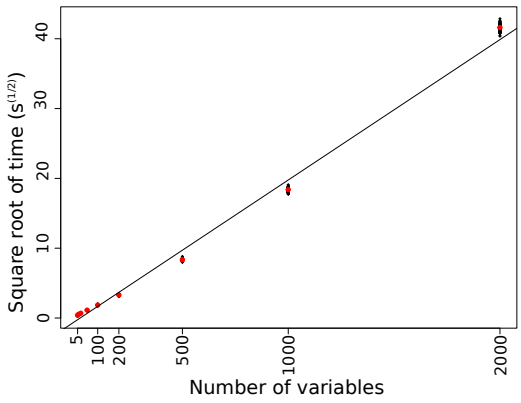


Figure 8: Scalability w.r.t the number of variables

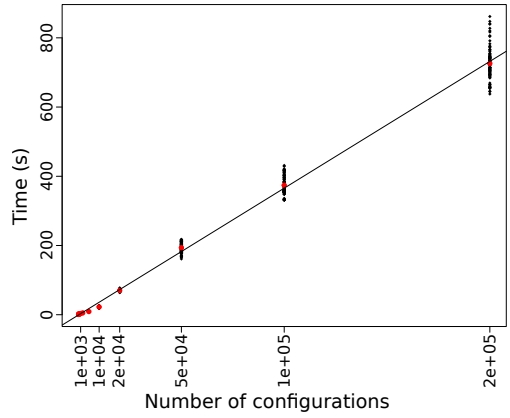


Figure 9: Scalability w.r.t the number of configurations

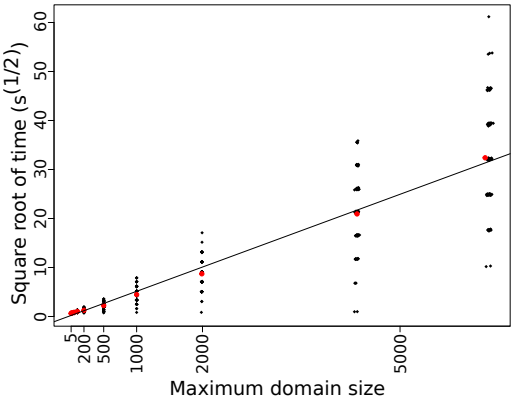


Figure 10: Scalability w.r.t the maximum domain size

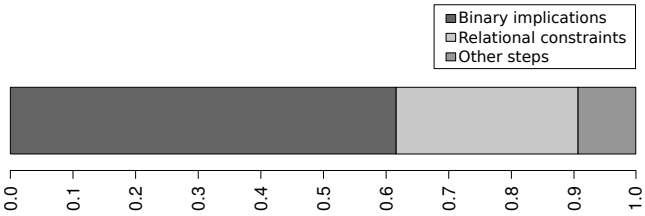


Figure 11: Time complexity distribution for all experiments without or-groups computation

6.3 Scalability w.r.t the number of configurations

To evaluate the scalability with respect to the number of configurations, we perform the synthesis of random matrices with 100 variables, a maximum domain size of 10 and a number of configurations ranging from 5 to 200,000. With 100 variables, and 10 as the maximum domain size, we can generate 10^{100} distinct configurations. This number is big enough to ensure that our generator can randomly generate 5 to 200,000 distinct configurations.

Figure 9 reports the synthesis time in each case. As shown in this figure, the time grows linearly with the number of configurations, and the correlation coefficient is 0.997.

6.4 Scalability w.r.t the maximum domain size

To evaluate the scalability with respect to the maximum domain size, we perform the synthesis of random matrices with 10 variables, 10,000 configurations and a maximum domain size ranging from 5 to 10,000[4].

TODO (Guillaume): TO VALIDATE : footnote for explaining the 5000 / 10000 diff

Figure 10 presents the square root of the synthesis time. We notice that for each value of the domain size, the points are distributed in small groups. For instance, we can see nine groups of points for a maximum domain size of 2000. Each group represents the execution of our algorithm with

[4]The random matrix generator cannot ensure that the maximum domain sizes are always reached. For instance, with a target domain size of 5000 (resp. 10,000), the generator produces configuration matrices with domains containing approximately 4,385 values (resp. 6,416). However, this difference does not impact the validity of the experiment.

matrices that have the same number of attributes. However, we see that the number of attributes does not significantly affect the maximum domain size (the maximum domain size is approximately the same for all groups of results).

A linear regression line fits the average square root of the time, with a correlation coefficient of 0.932. This implies that the synthesis time grows quadratically with the maximum domain size.

6.5 Time Complexity Distribution

To further understand the overall time complexity, we analyze its distribution over different steps of the algorithm.

TODO (Guillaume): TO VALIDATE : theoretical complexity

According to our theoretical analysis (see [6] for details), the two hard parts of the synthesis algorithm are the computation of mutex-groups (exponential complexity) and or-groups (NP-complete). The rest of the algorihtm has a complexity of $O(v^4.d+v^2.d^2+v^2.c.d)$. We note that 93.8% of the configuration matrices in our dataset produce mutex graphs that contain absolutely no edges. In such cases, computing mutex-groups is trivial. Therefore, our experiments confirm the polynomial complexity of the rest of the algorihtm.

In Figure 11, we depict the average distribution for all previous experiments that do not contain the computation of or-groups. The results clearly show that the major part of the algorithm is spent on the computation of binary implications, and relational constraints for RC. The rest of the synthesis represents less than 10% of the total duration. Optimizing these two main steps would significantly decrease the time necessary for synthesizing an AFM.

7. EVALUATION ON REALISTIC DATA

TODO (Guillaume): Find a title

To provide an insight of the scalability of our approach on realistic configuration matrices, we executed our algorithm on configuration matrices extracted from *Best Buy* website. *Best Buy* is a well known American company that sells consumer electronics.

On their website, the description of each product is completed with a matrix that describes technical information. Each matrix is formed of two columns in order to associate a feature or an attribute of a product to its value. For instance, the matrix describing a particular laptop may have a line that associates the caracteristics *Operating System* to its value *Windows 8.1*. The website offers a way to compare products that consists in merging the matrices to form a single configuration matrix which is similar to the one in Figure 2.

7.1 Experimental settings

We developed an automated technique to extract configuration matrices from *Best Buy* website. The procedure is composed of 3 steps. First, it selects a set of products whose matrices have a significant amount of feature and attributes in common. Then, it merges the matrices of each selected product to obtain a configuration matrix. The resulting configuration matrix may contain empty cells. Such cells have no meaningful information from a variability point of view. The last step of the procedure consists in giving an interpretation to these cells. If a feature or an attribute contain only integers, the empty cells are interpreted as "0". Otherwise, the empty cells are interpreted as "No" which means that the feature or attribute is not present.

With this procedure, we extracted 22 configurations matrices from the website that form our dataset for the experiment. The matrices contained at most X% of empty cells before interpretation. Table 1 reports statistics on the dataset about the number of variables, configurations, the maximum domain size and the number of empty cells before interpretation.

Table 1: Statistics on the Best Buy dataset.

	Min	Median	Mean	Max
Variables	24	56.5	56.6	91
Configurations	11	27.0	47.1	203
Max domain size	11	27.0	47.1	203
Empty cells before interpretation	4.1%	24.7%	20.9%	25.0%

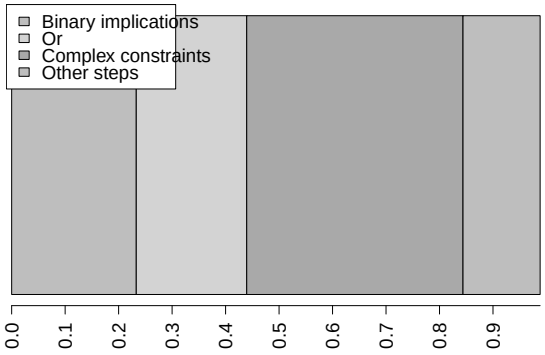


Figure 12: Time complexity distribution of Algorithm 1 on the *Best Buy* dataset

7.2 Scalability on realistic configuration matrices

To evaluate the scalability of our synthesis algorithm on the *Best Buy* dataset, we measured the execution time of Algorithm 1 with the computation of or-groups activated. To have comparable results with previous experiments on random matrices, we executed the algorithm on the same cluster of computers.

On the *Best Buy* dataset, the execution time of Algorithm 1 is 1.8s in average with a median of 0.6s. The most challenging configuration matrix has 73 variables and 203 configurations. The synthesis of the associated AFM takes 25.4s. Figure 12 reports the average distribution for the dataset. It shows that the computation of binary implications, or-groups and the relational constraints are the most time consuming tasks. It confirms the results of the experiments on random matrices. However, we note that on the *Best Buy* dataset, the computation of or-groups is executed in a reasonable time.

To further compare with previous experiments, we performed the same experiment but with the computation of or-groups deactivated. In these conditions, the execution time of Algorithm 1 is 0.6s in average with a median of 0.5s. This time, the synthesis of an AFM from the most challenging configuration matrix takes only 2.0s. The experiment

further confirms that the computation of or-groups is the computationally hardest part of the algorithm. It also shows that the algorithm scales on realistic configuration matrices with similar execution time as with random matrices.

8. THREATS TO VALIDITY

TODO (Mathieu): Modify

An external threat is that the evaluation of Section 6 is based on the generation of random matrices. Using such matrices may not reflect the practical complexity of our algorithm. To mitigate this threat we complement with a realistic data set.

Evaluating the scalability on a cluster of computers instead of a single one may impact the scalability results and is an internal threat to validity. We limited this threat to validity by using a cluster composed of identical nodes. Even if the nodes do not represent a standard computer, we only modify the absolute values of the experiments. The practical complexity of the algorithm is not influenced by this gain of processing power. Moreover, all the necessary data for the experiments are present in the local disks of the nodes thus avoiding any network related issue. Finally, we performed 100 runs for each set of parameters in order to reduce any variation of performance.

Another threat to internal validity is related to our implementation of the algorithm. To check the correctness of the implementation, we have manually reviewed some resulting AFMs. We also tested the algorithm against a set of manually designed configuration matrices. Each matrix represents a minimal example of a construct of an AFM (*e.g.,* one of the matrices represents an AFM composed of a single xor-group). The test suite covers all the concepts in an AFM. None of these experiments revealed any anomalies in our implementation.

9. CONCLUSION

We presented the foundations for synthesizing attributed feature models (AFMs) from product descriptions. We introduced the formalism of *configuration matrix* for documenting a set of products along different Boolean and numerical values. We then sought to understand the relationship between configuration matrices and AFMs. The key contributions of the paper can be summarized as follows:

- We described formal properties of AFMs (over-approximation, equivalence) and established semantic correspondences with the formalism of configuration matrices (Section 4);
- We designed and implemented a tool-supported, parameterizable synthesis algorithm (Section 5);
- We empirically evaluated the scalability of the synthesis algorithm (Section 6).

The synthesis techniques presented in this paper open avenues for investigating novel reverse engineering scenarios involving attributes. Numerous approaches for mining and extracting features or constraints [1, 3, 16, 18, 24–26] can be used to process different types of artefacts and eventually fed our synthesis algorithm – this time with the support of attributes.

As future work, we also plan to further study some properties of the synthesis – like scalability (performance), readability and usefulness of computed constraints, and the over-approximation effect. We are currently investigating the use of AFM synthesis in practical settings.

Acknowledgements

The second author is funded by the Research Council of Norway (the ModelFusion Project - NFR 205606).

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

10. REFERENCES

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *SoSyM*, 2013.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.

[3] E. Bagheri, F. Ensan, and D. Gasevic. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering*, 19(3):335–377, 2012.

[4] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In *SLE'10*, pages 102–122. Springer, 2011.

[5] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr. Breathing ontological knowledge into feature model synthesis: an empirical study. *Empirical Software Engineering*, 2015.

[6] G. Bécan, R. Behjati, A. Gotlieb, and M. Acher. Synthesis of attributed feature models from product descriptions: Foundations. *arXiv preprint arXiv:1502.04645*, 2015.

[7] G. Bécan, N. Sannier, M. Acher, O. Barais, A. Blouin, and B. Baudry. Automating the formalization of product comparison matrices. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 433–444. ACM, 2014.

[8] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[9] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés. Fama: Tooling a framework for the automated analysis of feature models. *VaMoS*, 2007:01, 2007.

[10] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski. Three cases of feature-based variability modeling in industry. In *MODELS*, pages 302–319, 2014.

[11] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In *VaMoS'13*. ACM, 2013.

[12] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program.*, 76(12), 2011.

[13] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *ICSE'13*, pages 472–481, 2013.

[14] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature models are views on ontologies. In *SPLC '06*, 2006.

[15] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *SPLC'08*, pages 22–31, 2008.

[16] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *ESEC/FSE'13*, 2013.

[17] H. Eichelberger and K. Schmid. Mapping the design-space of textual variability modeling languages: a refined

analysis. *International Journal on Software Tools for Technology Transfer*, pages 1–26, 2014.

[18] A. Ferrari, G. O. Spagnolo, and F. dell'Orletta. Mining commonalities and variabilities from natural language documents. In *SPLC*, 2013.

[19] J. Guo, E. Zulkoski, R. Olaechea, D. Rayside, K. Czarnecki, S. Apel, and J. M. Atlee. Scaling exact multi-objective combinatorial optimization by parallelization. In *ASE '14*, pages 409–420, 2014.

[20] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. On extracting feature models from sets of valid feature combinations. In *FASE'13*, 2013.

[21] A. Hubaux, T. T. Tun, and P. Heymans. Separation of concerns in feature diagram languages: A systematic survey. *ACM Computing Surveys*, 2013.

[22] M. Janota, V. Kuzina, and A. Wasowski. Model construction with external constraints: An interactive journey from semantics to syntax. In *MODELS'08*, 2008.

[23] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 2014.

[24] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.

[25] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of feature models from formal contexts. In *FOSD'11*, 2011.

[26] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*, pages 461–470. ACM, 2011.

[27] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki. Efficient synthesis of feature models. *Information and Software Technology*, 56(9), 2014.

[28] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014.