On extracting relevant and complex variability information from software descriptions with pattern structures
Jessie Carbonnel, Marianne Huchard, Clémentine Nebut

To cite this version:

HAL Id: lirmm-01872807
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01872807
Submitted on 12 Sep 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License
ABSTRACT
The migration from existing software variants to a software product line is an arduous task that necessitates to synthesise a variability model based on already developed softwares. Nowadays, the increasing complexity of software product lines compels practitioners to design more complex variability models that represent other information than binary features, e.g., multi-valued attributes. Assisting the extraction of complex variability models from variant descriptions is a key task to help the migration towards complex software product lines. In this paper, we address the problem of extracting complex variability information from software descriptions, as a part of the process of complex variability model synthesis. We propose an approach based on Pattern Structures to extract variability information, in the form of logical relationships involving both binary features and multi-valued attributes.

CSC CONCEPTS
• Information systems → Information extraction; Software and its engineering → Software product lines; Software reverse engineering;

KEYWORDS
Software Product Line, Reverse Engineering, Variability Extraction

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference format:

2 METHOD OVERVIEW
The proposed extraction method is based on Pattern Structures (PS) [2], a mathematical framework used for knowledge discovery on a set of objects described by complex data called patterns, allowing to extract all logical relationships that are true for the considered set of objects. In this paper, we propose a method that uses Pattern Structures to extract logical relationships involving both features and multi-valued attributes from complex variability descriptions, i.e., depicting not only binary features. It is a first step towards the automated synthesis of complex variability models from existing software systems. The aim of this research is to ease the transition of more and more complex software variants that have been individually developed, to software reuse and mass-customisation approaches. Our method is sound and complete, and offers theoretical foundations to complex variability information extraction.

Figure 1: Extracting complex variability information from variant descriptions using Pattern Structures.
Table 1: Excerpt of a multi-valued matrix representing software variant descriptions taken from wikipedia

<table>
<thead>
<tr>
<th>Corporate</th>
<th>Personal</th>
<th>First release</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Confluence</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>GrokOla</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Jive</td>
<td>X</td>
<td>2006</td>
</tr>
<tr>
<td>4</td>
<td>Wagon</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>Wiki.js</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Identifying patterns. Patterns can be values of atomic types, e.g., a Date, a String or a number. But, one can choose to compose several patterns to form a vector of patterns, that can be considered a pattern as well. In our case, a variant description may be represented as a pattern vector depicting the values of each characteristic of the matrix. This first step consists in identifying in the variant descriptions the set of component patterns that will form the pattern vectors. As binary characteristics have a true, false domain, the set of binary characteristics having a true value of a variant can be considered as one multi-valued characteristic. This representation choice does not alter the extracted information and permits to simplify the processed data.

The pattern vectors for the software variants of Table 1 are thus of the form \( v = (P_F, P_f, P_1) \), where \( P_F \) is the set of features (i.e., binary characteristics), followed by the value of the attribute First Release (FR) \( P_f \) and then the value of the attribute Language \( P_1 \). The two following pattern vectors respectively represent the softwares Confluence and GrokOla:

\[
\begin{align*}
&v_1 = (\{ \text{Corporate, Personal} \}, \text{FR} = 2004, \text{Language} = \text{Java EE}) \\
&v_2 = (\{ \text{Corporate} \}, \text{FR} = 2014, \text{Language} = \text{JavaScript})
\end{align*}
\]

Defining taxonomies. Now that we have specified the pattern vectors, we have to define similarities between the elements of each component pattern. A similarity operator is associated to a subsumption relation \( \subseteq \), allowing the set of patterns \( P \) to be partially ordered in a taxonomy, in which each pair of elements has a unique most specific common generalisation pattern. In other words, it structures the patterns by specialisation, and \( p_1 \subseteq p_2 \iff p_1 \cap p_2 = p_1, P_{p1}, p_2 \in P \). For example, let us consider \( P_f \) the first component of our vector, representing the binary features owned by the variants. The similarity between a variant described by the feature set \( \{ \text{Corporate, Personal} \} \) and another variant described by \( \{ \text{Corporate} \} \) is that they are both described by the feature set \( \{ \text{Corporate} \} \) (denoted \( \{ \text{Corporate, Personal} \} \cap \{ \text{Corporate} \} = \{ \text{Corporate} \} \)). Also, a way to define the similarity of a variant first released in 2004 and a variant first released in 2014 is that they have both been released between 2004 and 2014 (2004 ∩ 2014 = [2004, 2014]). Once we have defined the similarity operators of the component pattern characterising our software variants, we can compute similarity between these pattern vectors:

\[
\begin{align*}
&v_1 \cap v_2 = (\{ \text{Corporate} \}, \text{FR} = [2004, 2014], \text{Language} = \neg \text{Ruby})
\end{align*}
\]

Building the lattice. The set of objects \( O \), the pattern taxonomy \((P, \cap)\) and the mapping \( \delta : O \rightarrow P \) associating each object of \( O \) to its pattern in \( P \), form a Pattern Structure. Given a Pattern Structure \( PS = (O, (P, \cap), \delta) \), a pattern concept of \( PS \) is a tuple \((O', p)\) such that \( O' \subseteq O \) and \( p \in (P, \cap) \), which represents a maximal set of objects and the most specific pattern corresponding to these objects. This means that there is no other object than the ones in \( O' \) that corresponds to the pattern \( p \), and that there is no other pattern more specific than \( p \) which corresponds to all the objects from \( O' \). These pattern concepts can be partially ordered: given two pattern concepts \( C_1 = (O_1, p_1) \) and \( C_2 = (O_2, p_2) \) of \( PS \), \( C_1 \leq_{ps} C_2 \) if and only if \( O_1 \subseteq O_2 \) and \( p_1 \subseteq p_2 \). The set of all pattern concepts of \( PS \) provided with the order \( \leq_{ps} \) forms a lattice structure called a pattern concept lattice.

Knowledge discovery. The pattern concept lattice emphasises several types of variability information, i.e., logical relationships between features and/or attribute values, that state how these elements can be combined to compose a software of the software family. The most represented and studied variability relationship types in the literature are implications between elements, co-occurring elements, mutually exclusive elements and feature-groups. All these types of variability information can be extracted from \( PS \) using the proper algorithms [5]. By theoretical definition, variability extraction with Pattern Structures is complete, in the sense where they allow to extract all logical relationships (among the 4 aforementioned types) that are true for the considered set of variants.

3 CONCLUSION AND PERSPECTIVES

We presented an approach to extract complex variability information from software variant descriptions based on Pattern Structures. It allowed us to extract variability information by considering not only the software variants’ features, but also multi-valued attributes. This work is a first step toward a more generic approach to assist practitioners into extracting complex variability models.

Our extraction method is complete and therefore extracts an important number of relationships. It appears that a significant part of them are “accidental”, i.e., true for the considered set of variants but not regarding the domain. In future work, we plan to study existing techniques to lower the number of considered extracted relationships by deepening the separation of the meaningful ones from the accidental ones. Another future work will be to consider the extraction of other kinds of variability information that could be useful to synthesise complex variability models from software descriptions. Particularly, relationships between several independent but connected software families are to be studied, allowing applications in the field of multiple software product lines.

REFERENCES


