Towards detecting redundancy in domain engineering process using first order logic rules

Abdelrahman Osman Elfaki*, Sim Liew Fong, Kevin Loo Teow Aik and Md Gapar Md Johar

Faculty of Information Science and Engineering, Management and Science University, University Drive, Off Persiaran Olahraga, Section 13, 40100 Shah Alam, Selangor Darul Ehsan, Malaysia
E-mail: abdelrahman_elfaki@msu.edu.my
E-mail: lfsim@msu.edu.my
E-mail: Kevin@msu.edu.my
E-mail: gapar@msu.edu.my
*Corresponding author

Abstract: Software product line (SPL) is an emerging methodology for developing software products. SPL consists of two processes: domain-engineering and application-engineering. Successful software product is highly dependent on the validity of a domain engineering process. Therefore, validation is a significant process within the domain-engineering. Anomalies such as dead feature, redundancy, and wrong-cardinality are well-known problems in SPL. In the literature, redundancy did not take the signs of attentions as a dead feature and wrong-cardinality. The maturity of the SPL can be enhanced by detecting and removing the redundancy from the domain engineering. This paper proposes first order logic (FOL) rules for detecting the redundancy in domain-engineering process. Detecting redundancy in the domain engineering direct is our contribution. Our methodology comprised of three steps:

1. variability is modelled in the form of predicates as a prerequisite
2. for each type of the redundancy, a general form is formulated to swathe all possible cases
3. FOL rules are illustrated to implement each possibility based on deducing the results from predefined cases.

As a result, all forms of redundancies in the domain-engineering process are amorphous. Finally, experiments are conducted to attest the scalability of our method.

Keywords: software product line; SPL; domain engineering; variability.


Biographical notes: Abdelrahman Osman Elfaki is currently a Senior Lecturer at Management and Science University in Malaysia. He has been involved in many projects and researches related to software engineering in different countries which refined his software engineering experience in both practical and academic fields. His current interests are modeling software product lines engineering and managing variability. He has published many papers related to his current interest.

Copyright © 2013 Inderscience Enterprises Ltd.
Sim Liew Fong graduated in 1999 with a double degree in English and Computer Science and a Master in IT in 2001. He has joined the Cosmopoint Sdn. Bhd. in 2001 and was appointed as the Program Manager. He has been awarded as Best Lecturer in 2002. He joined the Management and Science University (formerly known as KUTPM) in 2003 as a Lecturer. He then received the Best Researcher Award in 2005, 2007 and 2008. He was promoted to Senior Lecturer position in 2007. He received the Best Mentor Award in 2010 and has conducted research in Jawi and collaborated and received research grant from Ministry of Arts, Culture and Heritage Malaysia.

Kevin Loo Teow Aik graduated from Universiti Putra Malaysia with a Bachelor of Arts (English and Computer Science) in 1999 and Master of Science (Information Technology) in 2001. He has been working in the education line since 2002. Currently, he is actively involved in teaching, research and administrative activities in Management and Science University. Since being appointed as the Director of Industrial Linkages in 2010, he oversees and plans collaborations between the university and the industries. He is still involved in project supervision, research and publications with other researchers in the university. His area of interest includes software engineering and mobile application development.

Md Gapar Md Johar is the Vice President of the Academic of Management and Science University (MSU), Director of Information Technology and Innovation Center (MSU) and a Certified E-commerce Consultant, He has over 35 years of working experience in software and application development and has developed many applications such as housing loan accounting system, commercial vehicle licensing system, human resource management system, learning content management system and campus management system. His research interests include object-oriented analysis and design, software engineering, Java programming, digital image analysis, radio frequency identification (RFID) and knowledge management. Currently, he is involved in research and development of Jawi Word Processor and Converter in open system environment, image processing and character recognition, e-learning and digital content design. He holds a BSc (Hons) in Computer Science, an MSc in Data Engineering and a PhD in Computer Science.

1 Introduction and motivations

Software product line (SPL) has proven to be an effective strategy benefiting from software reuse (Bosch, 2002), allowing many organisations to reduce development costs and duration, and meanwhile increasing product quality (Clements and Northrop, 2001). It is an evolution from software reuse and commercial off-the-shelf (COTS) methodologies. Meyer and Lopez (1995) defined SPL as a set of products that share a common core technology and address a related set of market applications. SPL has two main processes: the first process is the domain-engineering process that represents domain repository and it is responsible for preparing domain artefacts including variability. The second process is the application engineering that aims to consume specific artefact, picking through variability, concerning to specification of the desired application. Feature model (FM) (Kang et al., 1990) and orthogonal variability model (OVM) (Pohl et al., 2005) are the useful techniques for representing variability in the SPL (Roos-Frantz, 2009). A particular product-line member is defined by a unique
Towards detecting redundancy in domain engineering process

combination of features (if variability modelled using FM) or a unique combination of variants (if variability modelled using OVM). The set of all legal features or variants combinations defines the set of product line members.

Variability is the ability of a system to be efficiently extended, changed, customised, or configured to be used in a particular context (Svahnberg et al., 2005). Variability is the main consideration in developing SPL, as it identifies the commonality and variability within a domain engineering process (van der Linden et al., 2007). A suitable variability model is a successful condition for a variability management. Variability management is the overall process of defining what is common and what is different across products in a SPL (Junior et al., 2005). Variability modelling techniques have been developed to assist engineers in dealing with the complications of a variability management (Sinnema and Deelstra, 2007). The success of SPL is largely dependent on effective variability management (Bachmann and Bass, 2001). According to Beuche et al. (2004) variability management proposals should be simple (easy to understand), universal, able to manage variability at all levels of abstraction, and the introduction of a new variability expression should be as easy as possible. These conditions for successful variability modelling methods are applicable in our proposed method. Moreover, the proposed method can be used for validating SPL.

FM is considered as one of the well-known methods for modelling SPLs. According to Czarnecki and Eisenecker (2002), the two most popular definitions of FM are:

1. an end user visible characteristic of a system
2. a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholders of the concept.

FM is a description of the commonalities and differences between the individual software systems in a SPL. In more detail, FM defines a set of valid feature combinations. Each of such valid feature combinations can be served as a specification of a software product (Asikainen et al., 2004). Czarnecki defined cardinality-based feature modelling by integrating a number of extensions to the original feature-oriented domain analysis (FODA) notation in Czarnecki et al. (2004). FM appeals to many SPL developers as features are essential abstractions that both customers and developers understand. Customers and developers usually speak of product characteristics in terms of those features the product has or delivers. Therefore, it is natural and intuitive to express any commonality or variability in terms of features (Kang et al., 2002).

OVM is one of the useful techniques to represent variability, which provides a cross-sectional view of the variability through all software development artefacts. Variation points, variants and constraint dependencies rules describe variability (in OVM). Constraint dependencies rules are: variation point requiring or excluding another variation point, variant requiring or excluding another variant, and variant requiring or excluding variation point.

The principal objective of SPL is to configure a successful software product from the domain engineering process by managing SPL artefacts using variability modelling technique. Recently, validation of SPL has been discussed as an important issue concentrating on the maturity of SPL (Benavides et al., 2008, 2009, 2010a). Validating SPL intends to ensure the correctness of artefacts in domain engineering and to produce error-free products including the possibility of providing explanations to the modeller so that errors can be detected and eliminated. Usually, a medium-size SPL contains
thousands of features. Therefore, validating SPL represents a challenge. The validation of SPL is a vital process and not feasible to be done manually. The lack of a formal semantics and reasoning support of FM has hindered the development of validation methods for FM (Wang et al., 2007). The automated validation of SPL was already identified as a critical task in Batory et al. (2006), Czarnecki (1998) and Massen and Litcher (2005).

In our previous work (Elfaki et al., 2009b), we explain how first order logic (FOL) can be used as a variability modelling technique, in which domain-engineering modelled as a knowledge-base. Also, in our previous works (Elfaki et al., 2008, 2009a, 2009b, 2009c), FOL rules are suggested to validate SPL. In addition, these FOL rules satisfied constraint dependency checking, optimisation and explanation. Moreover, implementation examples are discussed. In Elfaki et al. (2009a) the scalability of the proposed method is illustrated. In this paper, we complete our work by defining FOL rules for detecting redundancy in domain-engineering.

This paper is structured as follows. Related work is discussed in Section 2. The detection operations are discussed in Section 3. Scalability testing is illustrated in Section 4. Discussion and comparison with the previous work is mentioned in Section 5. Finally, conclusion and future work is presented in Section 6.

2 Related work

A knowledge-based product derivation process is suggested in Hotez and Krebs (2003a, 2003b). A knowledge-based product derivation process is a configuration model that includes three entities of knowledge base. The automatic selection provides a solution for complexity of product line variability. In contrast to the proposed method, the knowledge-based product derivation process does not provide explicit definition of variability notations and there is no mention for validating domain engineering.

Mannion (2002) was the first who connect propositional formulas to FM. Mannion’s model did not concern cross-tree constraints (require and exclude constraints) and has not been used for supporting validation operations. Zhang et al. (2004) defined a meta-model of FM using Unified Modeling Language (UML) core package and took Mannion’s proposal as foundation. They also suggested the use of an automated tool to support the FM. Batory (2005) proposed a coherent connection between FM, grammar, and propositional formulas. Batory’s study represented basic FM using context-free grammars plus propositional logic. Batory’s proposal allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfiable solvers to debug FM. Robak and Pieczynski (2003) described a system based on a feature diagram tree, annotated with weighted variant features in the basis of fuzzy logic for modelling variability. Robak did not describe how to validate the domain-engineering. Sun and Zhang (2005) proposed first-order logic to model the FM and used alloy analyser (the alloy analyser is a tool for analysing models written in alloy) to automated consistency checking in the configuration process. The proposal in Fan and Zhang (2006) does not deal with the validation operations. Fan and Zhang (2006) used description logic for reasoning in FM. The work in Sun and Zhang (2005), and Fan and Zhang (2006) did not mention the redundancy problem. Czarnecki and Antkiewicz (2005) proposed a general template-based approach for mapping FM. Czarnecki and Pietrowszek (2006) used object-constraint language (OCL) to validate constraint rules. Peng et al. (2006)
Towards detecting redundancy in domain engineering process

presented an approach for modelling and verifying FM using Web Ontology Language (OWL). Janota and Iniry (2007) used higher-order logic for reasoning about FM. Krueger (2006) suggested partitioning FM into smaller models. Krueger’s idea was to localise each feature within the smallest scope that it needs to influence. Asirelli et al. (2009) modelled variability with Deontic logics. The works that discussed above are just concerned with modelling variability and with consistency checking and did not deal with other validation operations of SPL, and did not consider anomalies detection in SPL as well. Broek and Galvão (2009) transformed a FM into generalised feature trees. Broek’s work satisfies only these validation operations (dead features, minimal set of conflicting constraints, explanation).

Feature model analyser (FAMA) is a well established tool for analysing FM. FAMA was introduced in Benavides et al. (2005). FAMA is based on translation of FM into a constraint satisfaction problem (CSP) to automate feature selection with a constraint solver. Trinidad et al. (2006) defined a method to detect dead features based on finding all products and search for unused features. Trinidad et al. (2008a, 2008b) extended this CSP technique to identify full mandatory features, void features, and dead features. The idea is to automate error detection (full mandatory features, void features, and dead feature) based on theory of diagnosis. This model mapped FM to diagnose-model and used CSP to analyse FM. van Deurse and Klint (2002) developed normalisation rules for eliminating duplicate features. Unfortunately, the work did not discuss the constraint dependencies, which is the main source of redundancy (Massen and Lichter, 2004). According to the study that has been done by Benavides et al. (2010b) there is no work dealing with redundancy in SPL.

3 Operations for detecting redundancy

This section illustrates how the proposed method can be used to define and to provide auto-support for detecting the redundancy. Prolog (Wielemaker, 2007) is used for implementing the proposed operations.

Generally, FM contains redundancies when some semantic information is modelled in a multiple ways. Redundancy problem is described as a light issue (Massen and Lichter, 2004). Nevertheless, it can decrease the maintainability of SPL by amplifying variants in domain engineering. Three terms are used to model variability: variation points, variants and constraint dependency rules (Pohl et al., 2005). Redundancy in information that contains only description of variation points or variants is just duplication, i.e., repeated predicates. Duplication is straightforward and can be detected easily. Incorrect usage of constraint dependency rules can lead to redundancy (Massen and Lichter, 2004). Constraint dependencies are defined as require or exclude constraints. They can be defined by three scenarios: variant requires/excludes variant, variant requires/excludes variation point, and variation point requires/excludes variation point (Pohl et al., 2005).

In our approach a feature has only two possibilities: variation point or variant.

Table 1  Main definition for the variant and variation point

| Definition: | Let f_i denotes a feature, VP_i denotes a variation point, and V_i denotes a variant, where i \in \mathbb{I}+.

Table 1 shows the main definition of the variant and variation point.
In the following subsection, FOL rules are used to deduce the redundancy that occurs from the use of *require* or *exclude* constraints.

### 3.1 Redundancy caused from using the require constraint

The general form of describing the redundancy that is caused by using the require constraint is:

\[(\text{Feature requires a previously required feature}).\]

There are two scenarios for a previously required feature: a previously required feature caused by transitive requiring, or a previously selected feature required by a common feature. In the following, each scenario is explained and detective rules are presented.

#### 3.1.1 First scenario (a previously required feature caused by transitive requiring)

In this scenario, there is a series of requiring relations. For instance, \(f_1\) requires \(f_2\), \(f_2\) requires \(f_3\), \(f_3\) requires \(f_4\), and \(f_1\) requires \(f_4\). In this relation, \(f_1\) requires \(f_4\) is a redundant requiring relation. This redundancy is caused by a transitive relation. Table 2 shows the rule that is used for detecting a redundancy caused by a transitive requirement.

**Table 2** Rule for detecting redundancy caused by transitive requiring

<table>
<thead>
<tr>
<th>Definitions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\forall V_1, V_2, V_3 : \text{requires}<em>{\cdot\cdot}(V_1, V_2) \land \text{requires}</em>{\cdot\cdot}(V_2, V_3) \land \text{requires}_{\cdot\cdot}(V_1, V_3) \Rightarrow \text{redundancy})</td>
</tr>
</tbody>
</table>

**Example:** Suppose the domain-engineering process has the pattern that is appear in Table 3. By using the programme from Table 4 the redundancy can be detected. In Table 5, the Prolog programme is asked to find the two variants that make the redundancy. The answer is \(f_1\) and \(f_8\).

**Table 3** Pattern of the relation ‘require’ in the domain engineering process

| requires_{\cdot\cdot}(f_1, f_2), |
| requires_{\cdot\cdot}(f_2, f_3), |
| requires_{\cdot\cdot}(f_3, f_4), |
| requires_{\cdot\cdot}(f_4, f_5), |
| requires_{\cdot\cdot}(f_5, f_6), |
| requires_{\cdot\cdot}(f_6, f_7), |
| requires_{\cdot\cdot}(f_7, f_8), |
| requires_{\cdot\cdot}(f_1, f_8). |
Towards detecting redundancy in domain engineering process

Table 4  A programme for implementing the rule in Table 2

redeun(X, Y):-
  requires_v_v(X, Y), requires_v_v(Y, X).

redeun(X, Z):-
  requires_v_v(X, Y), requires_v_v(Y, Z), requires_v_v(X, Z).

redeun(X, Z):-
  requires_v_v(X, Y), requires(Y, Z), requires_v_v(X, Z).

requires(X, Y) :-
  requires_v_v(X, Y).

requires(X, Z) :-
  requires_v_v(X, Y), requires_v_v(Y, Z).

requires(X, Z) :-
  requires_v_v(X, Y), requires(Y, Z).

Table 5  Result of detecting redundancy in Table 3

?- redeun(X, Y).
  X = f1,
  Y = f8;
false.

3.1.2 Second scenario (a previously selected feature required by a common feature)

A common feature is described as a feature that is included in any product. Therefore, we can define a common variation point as a previously selected feature. If a variation point is selected, this means that all its common variants must also be selected (Elfaki et al., 2008). Therefore, there are two possibilities for a previously selected feature, which are: common variation point, or common variant which belongs to a previously selected variation point. In general, any constraint relation related to a variation point also affects that variation point’s common variants. Thus, constraint relations related to variation points could be implemented in two forms: one form for the variation point itself and one form for its common variants. In the following, we will discuss how the redundancy can be caused for each possibility of the require constraint and develop FOL rules to deduce this redundancy.

3.1.2.1 Variant requires variation point

In this form, a variation point is required. As we explained before, in the case of variation points, the requirement is implemented in the variation point itself and in its common variants. Therefore, there are two possibilities of redundancy that could happen within this relation:
Variant requires a variation point and requires also a common variant which belongs to this variation point.

If a variant requires a variation point this means that all common variants belonging to this variation point are also required. Requirement of a common variant after requiring its variation point is a redundancy. Rule 2 detects this case of redundancy. In rule 2, the variant $V_1$ requires the variation point $VP_2$ and the variant $V_2$ is the common variant that belongs to $VP_2$. Therefore, $V_1$ requires $V_2$ is redundant. Table 6 and Figure 1 illustrate rule 2.

Table 6  Rule 2

| $\exists VP_2, V_1 : type(V_1, variant) \land type(VP_2, variationpoint)$ \\
| $\land type(V_2, variant) \land variant(VP_2, V_2) \land requires_v_vp(V_1, VP_2)$ \\
| $\land common(V_2, yes) \land requires_v_v(V_1, V_2) \Rightarrow redundancy$ |

Figure 1  Illustration of rule 2 (see online version for colours)

2 Variant requires a common variation point:

A common variation point is included by default in any product (Pohl et al., 2005). Therefore, requirement of a common variation point is a redundancy. Rule 3 detects this case of redundancy. In rule 3, the variant $V_1$ requires the common variation point $VP_2$. Since the variation point $VP_2$ is included in any product, $V_1$ requires $VP_2$ is redundant. Table 7 and Figure 2 illustrate rule 3.

Table 7  Rule 3

| $\exists VP_2, V_1 : type(V_1, variant) \land type(VP_2, variationpoint)$ \\
| $\land common(VP_2, yes) \land requires_v_vp(V_1, VP_2) \Rightarrow redundancy$ |

Figure 2  Illustration of rule 3 (see online version for colours)
Towards detecting redundancy in domain engineering process

3.1.2.2 Variation point requires variation point

There are three possibilities could be implemented within this relation:

1. Variation point requires a variation point and in the same time, a common variant belonging to the first variation point requires the required variation point.

   If a variation point requires another variation point, this means that if the first variation point is selected then the required variation point must be selected. On the other hand, if a variation point is selected, this means that all the common variants belonging to this variation point are also selected. From these two facts, we can conclude that if a variation point requires another variation point and, at the same time, a common variant belonging to the requiring variation point requires the required variation point, then this is a redundancy. Rule 4 illustrates this case of redundancy. In rule 4, the variant \( V_1 \) is a common variant belonging to the variation point \( VP_1 \) and \( VP_1 \) requires the variation point \( VP_2 \). Therefore, \( V_1 \) requires \( VP_2 \) is redundant. Table 8 and Figure 3 illustrate rule 4.

   Table 8  
   \[
   \begin{array}{|c|c|c|c|c|c|}
   \hline
   \text{VP}_1, \text{VP}_2, V_1 : \text{type(VP}_1, \text{variationpoint}) \wedge \text{type(V}_1, \text{variant}) \wedge \text{type(VP}_2, \text{variationpoint}) \\
   \wedge \text{variants(VP}_1, V_1) \wedge \text{common(V}_1, \text{yes}) \wedge \text{requires_v_vp(VP}_1, \text{VP}_2) \\
   \wedge \text{requires_v_vp(V}_1, \text{VP}_1) \Rightarrow \text{redundancy} \\
   \hline
   \end{array}
   \]

   Figure 3 Illustration of rule 4 (see online version for colours)

2. Variation point requires a variation point and in the same time, a common variant which belonging to the first variation point requires a common variant belonging to the required variation point.

   If a variation point requires another variation point, this means that if the first variation point is selected, then the required variation point must be selected. On the other hand, if a variation point is selected, this means that all common variants belonging to this variation point are also selected. From these two facts, we can conclude that if a variation point requires another variation point and, at the same time, a common variant that belongs to the first variation point requires the common variant that belongs to the required variation point, then this is a redundancy. Rule 5 illustrates this case of redundancy. In rule 5, the variation point \( VP_1 \) requires the variation point \( VP_2 \). Thus, this requires relation will be reflected in all the common variants belonging to the two variation points. The variant \( V_1 \) requiring the variant \( V_2 \) is redundant. Table 9 and Figure 4 illustrate rule 5.
Table 9  Rule 5

\[ \forall VP_1, VP_2, V_i, V_j : \text{type}(VP_1, \text{variationpoint}) \wedge \text{type}(V_i, \text{variant}) \]
\[ \wedge \text{type}(VP_2, \text{variationpoint}) \wedge \text{type}(V_j, \text{variant}) \wedge \text{variants}(VP_1, V_i) \]
\[ \wedge \text{variants}(VP_2, V_j) \wedge \text{common}(V_i, \text{yes}) \wedge \text{common}(V_j, \text{yes}) \]
\[ \wedge \text{requires}_\text{vp_vp}(VP_1, VP_2) \wedge \text{requires}_\text{v}_\text{v}(V_i, V_j) \Rightarrow \text{redundancy} \]  

Figure 4  Illustration of rule 5 (see online version for colours)

3  Variation point requires a common variation point.

A common variation point is included by default in any product. Therefore, requiring a common variation point is a redundancy. Rule 6 implements this case.

Table 10  Rule 6

\[ \forall VP_1, VP_2 : \text{type}(VP_1, \text{variationpoint}) \wedge \text{type}(VP_2, \text{variationpoint}) \]
\[ \wedge \text{common}(VP_2, \text{yes}) \wedge \text{requires}_\text{vp_vp}(VP_1, VP_2) \Rightarrow \text{redundancy} \]  

Figure 5  Illustration of rule 6 (see online version for colours)

The variation point VP_2 is common, which means that it will be included in any product. VP_1 requiring VP_2 is redundant. Table 10 and Figure 5 illustrate rule 6.

3.1.2.3 Variant requires variant

Only one possible scenario could be implemented in this case: variant requires a variant and the required variant is a common variant that belongs to a common variation point. A common variant belongs to a common variation point is included by default in any product. Rule 7 detects this case of redundancy. In rule 7, the variant V_2 is a common variant that belongs to the common variation point VP_2, which means V_2 is included in any product. Therefore, V_1 requiring V_2 is redundant. Table 11 and Figure 6 illustrate rule 7.
Towards detecting redundancy in domain engineering process

Table 11

<table>
<thead>
<tr>
<th>Rule 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀VP₂, V₁, V₂ : type(V₁, variant) ∧ type(V₂, variant) ∧ type(VP₂, variationpoint) ∧ variants(VP₂, V₁) ∧ common(VP₂, yes) ∧ common(V₂, yes) ∧ requires_v_v(V₁, V₂) ⇒ redundancy</td>
</tr>
</tbody>
</table>

Figure 6 Illustration of rule 7 (see online version for colours)

3.2 Redundancy caused from using the exclude constraint

The general pattern for describing the redundancy that results from using the exclude constraint is:

(Feature excludes a previously excluded feature).

There are two scenarios for a previously excluded feature: a previously excluded feature caused from the exclude relation or a previously excluded feature because this feature is a dead feature. In the following, we discuss each scenario and develop FOL rules to deduce this redundancy. A dead feature is a feature never included in any product (Trinidad et al., 2008a, 2008b). A dead feature can be a dead variation point, or a dead variant. In the following, we discuss each scenario and develop FOL rules to deduce this redundancy:

3.2.1 First scenario (a previously excluded feature caused from the exclude constraint)

The general concept behind this scenario is based on the following assumption. If a variation point is excluded this means that all its variants are also excluded (Pohl et al., 2005). As we know, there are three possibilities for the exclude constraint: variant excludes variation point, variation point excludes variation point, or variant excludes variant. The third exclude relation (variant excludes another variant) is not included in this general concept. Therefore, two of the exclude constraint relations can be applied under the first scenario:

3.2.1.1 Variant excludes variation point

A variant excludes a variation point and, at the same time, excludes a variant that belongs to this excluded variation point. Rule 8 deduces this type of redundancy. In rule 8, the
variant $V_1$ excludes the variation point $VP_2$ and the variant $V_2$ belongs to $VP_2$. If a variant excludes a variation point this means that all its variants are also excluded. Thus, $V_1$ excluding $V_2$ is a redundancy. Table 12 and Figure 7 illustrate rule 8.

**Table 12**  
Rule 8

| $\forall VP_1, V_i, V_j : \text{type}(V_i, \text{variant}) \land \text{type}(VP_1, \text{variation point})$ |
| $\land \text{type}(V_j, \text{variant}) \land \text{variants}(VP_2, V_j) \land \text{excludes}_{vp}(V_i, VP_2)$ |
| $\land \text{excludes}_{v, vp}(V_i, V_j) \Rightarrow \text{redundancy}$ |

**Figure 7**  
Illustration of rule 8 (see online version for colours)

### 3.2.1.2 Variation point excludes variation point

A variation point excludes another variation point and, at the same time, a variant belonging to the first variation point excludes a variant belonging to the excluded variation point. Rule 9 deduces this kind of redundancy. In rule 9, the variant $V_1$ belongs to the variation point $VP_1$, the variant $V_2$ belongs to the variation point $VP_2$ and the variation point $VP_1$ excludes the variation point $VP_2$. In fact, if a variation point excludes another variation point, this means that all the variants belonging to the first variation point exclude by default all the variants belonging to the excluded variation point. According to the previous fact, $V_1$ excluding $V_2$ is a redundancy. Table 13 and Figure 8 illustrate rule 9.

**Table 13**  
Rule 9

| $\forall VP_1, VP_2, V_i, V_j : \text{type}(VP_1, \text{variation point}) \land \text{type}(V_i, \text{variant})$ |
| $\land \text{type}(VP_2, \text{variation point}) \land \text{type}(V_j, \text{variant}) \land \text{variants}(VP_2, V_j)$ |
| $\land \text{variants}(VP_1, V_j) \land \text{excludes}_{vp}(VP_1, VP_2) \land \text{excludes}_{v, vp}(V_i, V_j)$ |
| $\Rightarrow \text{redundancy}$ |

**Figure 8**  
Illustration of rule 9 (see online version for colours)
3.3 Second scenario (a previously excluded feature because this feature is a dead feature)

As mentioned earlier, a dead feature is a feature never appears in any product (Benavides et al., 2005). Therefore, excluding a dead feature is a redundant. A dead variation point is a variation point where all its variants are dead. In Elfaki et al. (2008), we defined three rules for dead features deducing. These rules are:

\begin{align*}
\forall x, y, z, n &: \text{type}(x, \text{variant}) \land \text{type}(y, \text{variationpoint}) \land \text{variants}(y, x) \\
\land &\text{type}(z, \text{variant}) \land \text{type}(n, \text{variationpoint}) \land \text{variants}(n, z) \\
\land &\text{common}(n, \text{yes}) \land \text{common}(z, \text{yes}) \land \text{excludes_v vp}(z, y) \\
\Rightarrow &\text{dead_variant}(x). \\
\end{align*}

(i)

If variant $x$ belongs to the variation point $y$, and the common variant $z$ belongs to the common variation point $n$, which means the variant $z$ included in any product, the variant $z$ excludes the variation point $y$ that means none of its variants must be selected at all.

\begin{align*}
\forall x, y, z &: \text{type}(x, \text{variant}) \land \text{type}(y, \text{variationpoint}) \land \text{variants}(y, x) \\
\land &\text{type}(z, \text{variationpoint}) \land \text{common}(z, \text{yes}) \land \text{excludes_vp vp}(z, y) \\
\Rightarrow &\text{dead_variant}(x). \\
\end{align*}

(ii)

If the variant $x$ belongs to the variation point $y$, and the common variation point $z$ excludes the variation point $y$ that means $y$ must be excluded. The variation point $y$ is a dead variation point. The following rule explains a dead variation point, which is a subset of the second rule:

\begin{align*}
\forall y, z &: \text{type}(y, \text{variationpoint}) \land \text{type}(z, \text{variationpoint}) \land \text{common}(z, \text{yes}) \\
\land &\text{excludes_vp vp}(z, y) \Rightarrow \text{dead_variationpoint}(y). \\
\end{align*}

(iii)

Exclusion of the variation point $y$ prevents selection of its variants.

\begin{align*}
\forall x, y, n &: \text{type}(n, \text{variant}) \land \text{type}(y, \text{variationpoint}) \land \text{variants}(y, n) \\
\land &\text{common}(y, \text{yes}) \land \text{common}(n, \text{yes}) \land \text{type}(x, \text{variant}) \\
\land &\text{excludes_v_v}(n, x) \Rightarrow \text{dead_variant}(x) \\
\end{align*}

(iii)

If the common variant $n$ belongs to the common variation point $y$ that means $y$ must be selected in any product. When $y$ is selected then $n$ must be selected, which means $n$ must be selected in any product. The variant $n$ excludes the variant $x$ that means $x$ must not be selected in any product.

After we explain the dead feature we comeback for discussing the redundancy that is caused from excluded a dead feature. A general form for this kind of redundancy is:

- (variant excludes a dead variant),
- (variant excludes a dead variation point), or
- (variation point excludes a dead variation point).

The following rules describe the implementation of this general form. The dead_varinat() and dead_variationpoint() predicates describe a dead variant and dead variation point respectively.
3.3.1 Variant excludes a dead variant

Rule 10 implements this redundancy. In rule 10, the variant $V_1$ excludes the dead variant $V_2$ and this is a redundancy.

3.3.2 Variant excludes a dead variation point

In this redundancy, there are two possibilities: variant excludes a dead variation point, or variant excludes another variant and the excluded variant belongs to a dead variation point. Rule 11 implements the first possibility and rule 12 implements the second possibility. In rule 11, the variant $V_1$ excludes the dead variation point $VP_2$ and this is a redundancy. In rule 12, the variant $V_1$ excludes the variant $V_2$. The variant $V_2$ belongs to the dead variation point $VP_2$ and this is a redundancy.

3.3.3 Variation point excludes a dead variation point

Rule 13 implements this inconsistency. In rule 13, the variation point $VP_1$ excludes the dead variation point $VP_2$ and this is a redundancy. Table 14 shows the rules for detecting a previously excluded feature because this feature is a dead feature.

Table 14  Rules for detecting a previously excluded feature because this feature is a dead feature

<table>
<thead>
<tr>
<th>Definitions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>type($VP_1$, variationpoint), type($VP_2$, variationpoint), type($V_1$, variant), type($V_2$, variant), variants($VP_1$, $V_1$), and variants($VP_2$, $V_2$).</td>
</tr>
<tr>
<td>$\forall V_1, V_2: \text{dead}<em>\text{variant}(V_2) \land \text{excludes}</em>\text{v}_\text{v}(V_1, V_2) \Rightarrow \text{redundancy}$ (10)</td>
</tr>
<tr>
<td>$\forall VP_2, V_1: \text{dead}<em>\text{variationpoint}(VP_2) \land \text{excludes}</em>\text{v}_\text{vp}(V_1, VP_2) \Rightarrow \text{redundancy}$ (11)</td>
</tr>
<tr>
<td>$\forall VP_2, V_1, V_2: \text{dead}<em>\text{variationpoint}(VP_2) \land \text{excludes}</em>\text{v}_\text{v}(V_1, V_2) \Rightarrow \text{redundancy}$ (12)</td>
</tr>
<tr>
<td>$\forall VP_2, VP_3: \text{dead}<em>\text{variationpoint}(VP_2) \land \text{excludes}</em>\text{vp}_\text{vp}(VP_2, VP_3) \Rightarrow \text{redundancy}$ (13)</td>
</tr>
</tbody>
</table>

4 Scalability testing

Scalability is a key factor in measuring the applicability of the techniques dealing with variability modelling in domain engineering (Segura, 2009). In this section, we discuss the experiments related to redundancy detection. Testing the output-time is our objective from these experiments. In the following, we describe the method of our experiment:

- **Generate the domain engineering as a dataset:** Domain engineering is generated in terms of predicates (variation points and variants). We generated four sets containing 1,000, 5,000, 15,000 and 20,000 variants. Variants are defined as a set of numbers represented in sequential order, for example, in the first set (1,000 variants) the variants are: 1, 2, 3,...,1,000 and in the last set (20,000 variants) the variants are: 1, 2, 3,...,20,000. The number of variation points in each set is equal to the number of variants divided by five, which means each variation point has five variants. As an example, in the first set (5,000 variants), the number of variation points equals 1,000.
Each variation point is defined as a sequence number having the term vp as a prefix, e.g., vp12.

- **Define the assumptions:** we have two assumptions:
  1. each variation point and variant has a unique name
  2. all variation points have the same number of variants.

- **Set the parameters:** The main parameters that are used in our experiments are the number of variants and the number of variation points. The remaining parameters are common variants, common variation points, ‘variant requires variant’, ‘variant excludes variant’, ‘variation point requires variation point’, ‘variation point excludes variation points’, ‘variant requires variation point’, and ‘variant excludes variation point’. These eight parameters are defined randomly as a percentage of the number of variants or variation points. Three ratios are defined: 10%, 25%, and 50%. The number of parameters related to the variant, such as common variant, ‘variant requires variant’, ‘variant excludes variant’, ‘variant requires variation point’, and ‘variant excludes variation point’, is defined as a percentage of the number of variants. The number of parameters related to the variation point, such as ‘variation point requires variation point’, is defined as a percentage of the number of variation points. As an example, in the dataset that contains 1,000 variants (note that in the dataset that contains 1,000 variants there are 200 variation points) and in the ratio 10%, the number of each parameter related to the variant is 100 and number of each parameter related to the variation point is 20, i.e., ‘100 variants require 100 variants’, ‘100 variants exclude 100 variants’, ‘20 variants require 20 variation points’, ‘20 variants exclude 20 variation points’, ‘20 variation points require 20 variation points’, ‘20 variation points exclude 20 variation points’, 100 common variants, and 20 common variation points. The variants and variation points that are included in these constraint relations are selected randomly. The variants are selected randomly from 1 to 1,000 and the variation points are selected randomly from 1 to 200.

Table 15 represents snapshots of the experimental dataset, i.e., the domain engineering used in our experiments.

- **Calculate output:** for each set, we made thirty experiments, and calculated the execution time as average. The experiments were done with the range (1,000–20,000) variant, and percentage range of 10%, 25%, and 50%.

<table>
<thead>
<tr>
<th>Table 15</th>
<th>Snapshot from experiment’s dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>type(vp1, variationpoint).type(1, variant).</td>
<td></td>
</tr>
<tr>
<td>variants(vp1, 1).</td>
<td></td>
</tr>
<tr>
<td>common(570, yes).</td>
<td></td>
</tr>
<tr>
<td>common(vp123, yes).</td>
<td></td>
</tr>
<tr>
<td>requires_v_v(7552, 2517).</td>
<td></td>
</tr>
<tr>
<td>requires_vp_vf(vp1572, vp1011).</td>
<td></td>
</tr>
<tr>
<td>excludes_vp_vf(vp759, vp134).</td>
<td></td>
</tr>
<tr>
<td>excludes_v_v(219, 2740).</td>
<td></td>
</tr>
<tr>
<td>requires_v_vp(3067, vp46).</td>
<td></td>
</tr>
<tr>
<td>excludes_v_vf(5654, vp1673).</td>
<td></td>
</tr>
</tbody>
</table>
The results show the execution time compared with number of variants, number of variation points, and the eight parameters (the six dependency constraint rules, common variants, and common variation points).

Figure 9 shows the result of scalability test for redundancy detection. White et al. (2008) defined 5,000 features as a suitable number to mimic industrial SPL. White’s experiments developed based on random numbers. Our results show that the proposed operations can deal with large number of features (20,000) in reasonable time. In addition to scalability testing, the experiments can be used also to prove the correctness of the proposed method. As an example, in the case of 1,000 variants and the percentage of parameters is 10% the result is ‘Redundancy between....441...and....817’. This result shows that the redundancy happened between the variants 441, and 817. Table 16 describes the relation between variants 441 and 817 (these relations extracted by searching for the two variants in the repository). These relations show that the variant 441 requires a common variant (817) which belongs to the common variation point (vp164).

**Figure 9** Results of redundancy-detection scalability test (see online version for colours)

```
Table 16  Facts that describing the relation between variants 411 and 817

<table>
<thead>
<tr>
<th>Relation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type(817, variant)</td>
<td></td>
</tr>
<tr>
<td>type(441, variants)</td>
<td></td>
</tr>
<tr>
<td>type(vp164, variationpoint)</td>
<td></td>
</tr>
<tr>
<td>variants(vp164, 817)</td>
<td></td>
</tr>
<tr>
<td>common(817, yes)</td>
<td></td>
</tr>
<tr>
<td>common(vp164, yes)</td>
<td></td>
</tr>
<tr>
<td>requires_v_v(441, 817)</td>
<td></td>
</tr>
</tbody>
</table>
```

5 Discussions and comparison with previous work

Although the use of FOL is not a new decision in validation of SPL, redundancy has not been solved yet (Benavides et al., 2010b). In literature, methods that are used for validating SPL are implemented validation in the time of the configuration process, i.e., in application engineering. Therefore, the proposed rules which detect redundancy in the domain-engineering process is a novel approach. Detecting redundancy in domain...
Towards detecting redundancy in domain engineering process

Generally, researchers used solvers for validating SPL by generating all products and detect error in each product. The process of generating all products is very toughest process and almost impossible with large-size SPL (Benavides et al., 2010b). Our approach is based on validating SPL by detecting errors in domain engineering (Elfaki et al., 2008, 2009a, 2009c). We proposed a methodology based on definition of a general pattern for each error. Later, all the implementations of this general pattern are defined. By using this general pattern all types of redundancy could be detected. This methodology may be used to solve other problems in SPL.

The proposed approach enhances the searching process (in SPL) by predefining specific cases and search only for them. Therefore, our scalability experiments show good results in the output-time and size of SPL (number of variants).

6 Conclusions and future work

In this paper, deducing rules are presented to deduce redundancy. The proposed method is based on modelling variability using predicates, then defining a general form for each type of redundancy. These definitions formulate the problems and allow the FOL rules to deduce the results from predefined cases.

Many methods are applying empirical results to test scalability by generating random FMs (Trinidad et al., 2006, 2008a, 2008b; White et al., 2008; Yan et al., 2007). Comparing with literature, our test range (1,000–20,000 variants) is sufficient to test scalability. The proposed method is limited to work only in certain environment, i.e., where constraint dependency rules are well known in all cases.

At present, we are developing now a software tool that allows users to model their SPL using the proposed method. The tool provides direct link between the two layers and implements our proposed validation operation. Moreover, it provides scalability tests.

References


Towards detecting redundancy in domain engineering process


