Software and Systems Modeling
https://doi.org/10.1007/s10270-023-01114-4

REGULAR PAPER

Verifying consistency of software product line architectures with product architectures

Hector A. Duran-Limon[1] · Perla Velasco-Elizondo[2] · Manuel Mora[3] · Maria E. Meda-Campana[1] · Karina Aguilar[4] · Martha Hernandez-Ochoa[5] · Leonardo Soto Sumuano[1]

Received: 9 May 2022 / Revised: 27 April 2023 / Accepted: 24 May 2023

Abstract

There has been increasing interest in modeling software product lines (SPLs) using architecture description languages (ADLs). However, sometimes it is required to reverse engineer an SPL architecture from a set of product architectures. This procedure needs to be performed manually as currently does not exist tool support to automate this task. In this case, verifying consistency between the product architectures and the reverse engineered SPL architecture is still a challenge; particularly, verifying component interconnection aspects of product architectures with respect to the commonality and variability of an SPL architecture represented in an ADL. Current approaches are unable to detect whether the component interconnections in a product architecture have inconsistencies with the component interconnections defined by the SPL architecture. To tackle these shortcomings, we developed the *Ontology-based Product Architecture Verification (OntoPAV)* framework. OntoPAV relies on the ontology formalism to capture the commonality and variability of SPLs architectures. Reasoning engines are employed to automatically identify component interconnection inconsistencies among SPL and product architectures. Our evaluation results show that our verifier has a high accuracy for detecting consistency errors and that it scales linearly for architectures from 1000 to 5000 architecture elements.

Keywords Software product lines · Software architecture · SPL Verification · Architecture verification · Ontologies · Model-driven engineering

1 Introduction

Software product lines (SPLs) have proved to improve software quality and shorten costs and development time [1, 2]. An SPL is a family of software systems that share a set of common assets (commonly referred to as the SPL commonality) but also have some other characteristics that make them different from each other (commonly named as SPL variability). *Feature modeling* [3] is a technique that allows for specifying commonalities and variabilities in an SPL. A feature represents an increment in product functionality [4]. Features are commonly represented in a tree structure called a feature diagram. A product configuration involves a selec-

Communicated by Ina Schaefer.

✉ Hector A. Duran-Limon
hduran@cucea.udg.mx

Perla Velasco-Elizondo
pvelasco@uaz.edu.mx

Manuel Mora
mmora@correo.uaa.mx

Maria E. Meda-Campana
emeda@cucea.udg.mx

Karina Aguilar
kaguilar@edu.uag.mx

Martha Hernandez-Ochoa
martha.ochoa@cunorte.udg.mx

Leonardo Soto Sumuano
leonardo.soto@cucea.udg.mx

[1] University of Guadalajara, CUCEA, Guadalajara, Jalisco, Mexico

[2] Autonomous University of Zacatecas, Zacatecas, Mexico

[3] Autonomous University of Aguascalientes, Aguascalientes, Mexico

[4] Autonomous University of Guadalajara, Guadalajara, Jalisco, Mexico

[5] University of Guadalajara, CUNORTE, Guadalajara, Jalisco, Mexico

Published online: 08 July 2023

tion of features from an SPL. Such a configuration represents a member of the product family. There are different types of relationships between a parent feature and a child feature [4]. A *Mandatory* relationship states that the child is included in all products belonging to the SPL to which the parent element belongs as well. An *Optional* relationship means that the child may or may not be part of a product. An *Alternative* relationship indicates that only one feature of a set of child features can be part of a product, whereas an *OR* relationship means that one or more features of a set of child features can be part of a product. Regarding cross-tree relationships, a *requires* relationship indicates that a feature requires the presence of another feature, whereas an *excludes* relationship excludes the presence of another feature.

Product derivation concerns the process of building a product from an SPL at the implementation level, whereas *product architecture derivation* (also called *architecture customization process*) aims at producing a product-specific architecture from a generic product line architecture. The variability in the feature model is often expressed in terms of *variation points (VPs)*, i.e., an area affected by variability, and systems options known as *variants*.[1] This variability can be also described in the software architecture and represented through different mechanisms. There has been increasing interest in modeling SPLs using *Architecture description languages (ADLs)*, such as Koala [5] and PL-Xelha [6]. ADLs [7] represent formal notations for describing software architectures in terms of coarse-grained components and connectors.

Importantly, consistency errors can take place when an SPL architecture is reverse engineered [8–10] from different product architectures involving legacy applications. Although there have been some attempts to reverse engineer an SPL architecture [11–13], to the best of our knowledge, currently there are not approaches able to automatically extract ADL-based SPL architectures. Hence, the procedure to extract an SPL architecture from ADL-based product architectures needs to be carried out manually. Consistency errors (or inconsistencies) between an SPL architecture and a product architecture are those that make them incompatible so that the product architecture cannot be derived from the SPL architecture. For example, there is an inconsistency when a *Mandatory* connection defined in the SPL architecture is not present in a product architecture. Several efforts have focused on verifying consistency of feature models [14–18]. Other research has focused on verifying consistency of SPL architectures [19–24]. Only a few approaches have addressed the issue of checking consistency between a product architecture and an SPL architecture, but mainly focusing on behavioral aspects [22, 23]. Hence, to the best of our knowledge there are not approaches able to verify the consistency of component interconnection aspects between a product architecture and the SPL architecture represented in an ADL.

In this paper, we present the **Onto***logy-based* **P***roduct* **A***rchitecture* **V***erification (OntoPAV)* framework, which verifies consistency between the component interconnections of a product architecture and the component interconnections defined in the SPL architecture. Our solution uses a language-independent approach to perform the verification process. We rely on the ontology formalism to capture the commonality and variability of an SPL architecture. In particular, we employ Pellet [25], an ontology reasoner, to check consistency of product architectures. Although our approach involves the use of a formal method (i.e., ontology formalism), the user does not require any knowledge about ontologies nor special training in any other technique from formal methods. We make use of model-driven engineering (MDE) techniques [26] to automate the generation process of the populated ontology. We have built a prototype to test our approach. Also, we developed an SPL of a web scholar system as a case example.

We rely on the same meta-metamodel of the SPL Language defined in our previous work on product architecture derivation [6], which focused on automating the derivation of product architectures that are modeled with an ADL. Apart from the Reasoning Engine (a third party component), all the modules of the framework of the present work are different from those of the previous work.[2] In addition, this work is different from our previous work in two ways, mainly. First, in our previous work the generated populated ontology captures information about the SPL architecture and the feature tree, whereas in this work the produced populated ontologies include information about both the SPL architecture and a product architecture. Second, our previous work focused on generating a product architecture, whereas this work focuses on verifying the consistency of product architectures of legacy systems with regard to a reverse engineered SPL architecture.

The paper is structured as follows. Section 2 presents a motivating example. The OntoPAV framework is described in Sect. 3. Our ontology and the verification rules are defined in Sect. 4. Section 5 presents the Verification process to find the origin of errors. Evaluation results are shown in Sect. 6. Related work is included in Sect. 7. Finally, some concluding remarks are given in Sect. 8.

[1] We distinguish the term variant from the term product variant. The former is an option of a variation point, whereas the latter is a specific derived product.

[2] Even the Ontology Factory is different, as in the present work the populated ontology generated by this Factory states the valid *Mandatory* connections, *OR* connections, *Alternative* connections, and *Optional* connections as well as the *requires/excludes* restrictions. None of these restrictions are defined in the populated ontologies of our previous work.

2 Motivating example

We use a running example to illustrate the proposed OntoPAV framework throughout the paper. Our running example consists of a scholar system whose first products were not developed within an SPL. Although we do not have an SPL architecture of the system, there are a number of product architectures. Hence, the SPL architecture is reverse engineered. This is carried out manually given that currently there are not tool support for automating this task. We present two product architectures of the scholar system in Fig. 2. However, all the product architectures employed to reverse engineer the SPL architecture can be consulted in [27].

The scholar system allows for managing the information of students, teachers as well as the subjects and courses taught. The first configuration (i.e., product architecture 1), shown in Fig. 2, can run a single term, whereas the second configuration (i.e., product architecture 2) can run multiple terms concurrently. The area of specialization is an *Optional* feature, given that it is present in the product architecture 1, but it is not present in the product architecture 2. Upon finishing the courses included in a degree, the scholar system can offer different options for finishing such a degree, namely thesis, research article, and two or more years of professional practice. The product architecture 1 includes the former two, whereas the product architecture 2 includes the thesis and professional practice.

A correctly derived SPL architecture of the scholar system is depicted in Fig. 3 in PL-Xelha [6], which involves 14 product architectures (and an SPL architecture with errors is shown in Fig. 4). Throughout the paper, we use PL-Xelha [6] only as a way to illustrate our approach since OntoPAV is language-independent, as mentioned earlier. Therefore, any modeling language that complies with the meta-metamodel defined in our previous work [6] can be used. Our meta-metamodel was defined in the metaobject facility (MOF).[3] Figure 1 depicts the meta-metamodel [6],[4] which defines the architectural commonality and variability. The former is defined in terms of *Nodes* and *Connections*. Nodes represent fixed architectural elements, whereas a Connection is a binding between nodes. Also, *Nodes* may have *Ports*, which are interaction points. *ProvidedPorts* offer services, whereas *RequiredPorts* require them. Ports are inner elements of both Nodes and variant elements. On the other side, *Variability* includes *VariationPoints* and *Variants*. The former permit the selection among different Variants. In addition, *VariantConnections* are connections that bind Variants to VariationPoints.

In PL-Xelha, there are two kinds of interfaces: *provided* and *required*. The former is represented by *facets* and offer

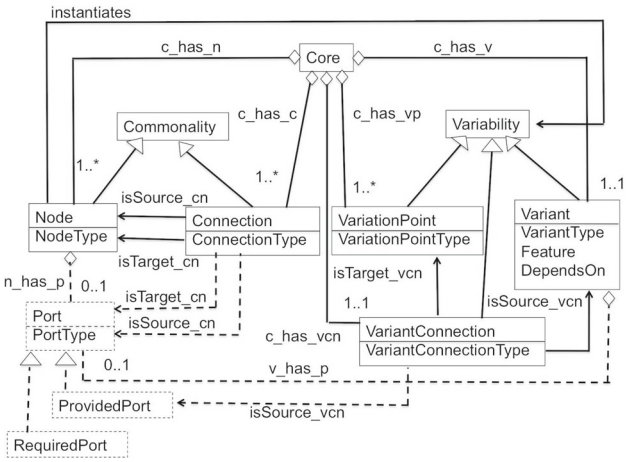


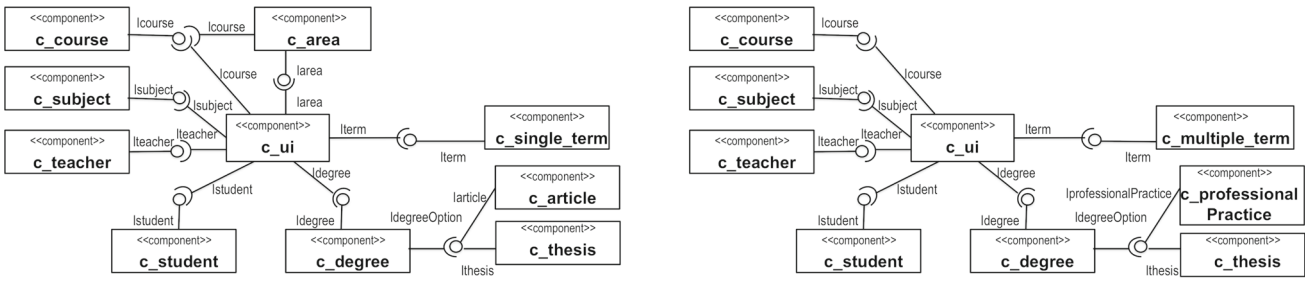
Fig. 1 Meta-metamodel

services, whereas the latter is denoted by *receptacles* and requires services. The SPL architecture includes a number of *Mandatory* components, such as `c_student` and `c_teacher`, which are represented with solid line boxes and are stereotyped as `<<component>>`, as shown in Fig. 3. Variation points are denoted with dotted line boxes and are stereotyped as `<<optionalComponent>>`, which have one associated or more features.[5] Variants are represented with solid line boxes and are stereotyped as `<<variant>>` and have defined the feature that represents. A feature can be mapped to one or more components and a component can also define dependencies with other features; however, none of these characteristics are employed in order to simplify our example. `vp_area` represents an *Optional* variation point, which is realized by the variant `c_area` when the feature `Area` is selected. An *Optional* variation point can have associated one or more *Optional connections*. An *Optional* connection is a connection (represented with a dotted line), which is optional. For instance, `c_ui` employs an *Optional* connection to connect to `vp_area`, and the latter also employs an *Optional* connection to connect to `c_course`. In case the feature `Area` is selected, these two connections are realized in the corresponding product architecture; otherwise, both connections are removed. `vp_term` is an *Alternative* variation point, which can be realized by the variant `c_single_term` in case the feature `Single` is selected, whereas `c_multiple_term` instantiates this variation point when the feature `Multiple` is chosen. An *OR* variation point is conformed by `vp_professionalPractice`, `vp_article`, and `vp_thesis`, and one or more of them can be instantiated by their associated variants. This depends on whether one or more of the following features are selected:

[3] http://www.omg.org/mof/.

[4] https://github.com/hduran-limon/copyRightMetaModel.

[5] PL-Xelha is also able to denote connectors and *Optional* connectors, which in our running example are not used.



(a) Product Architecture 1 (b) Product Architecture 2

Fig. 2 Product architectures of the scholar system

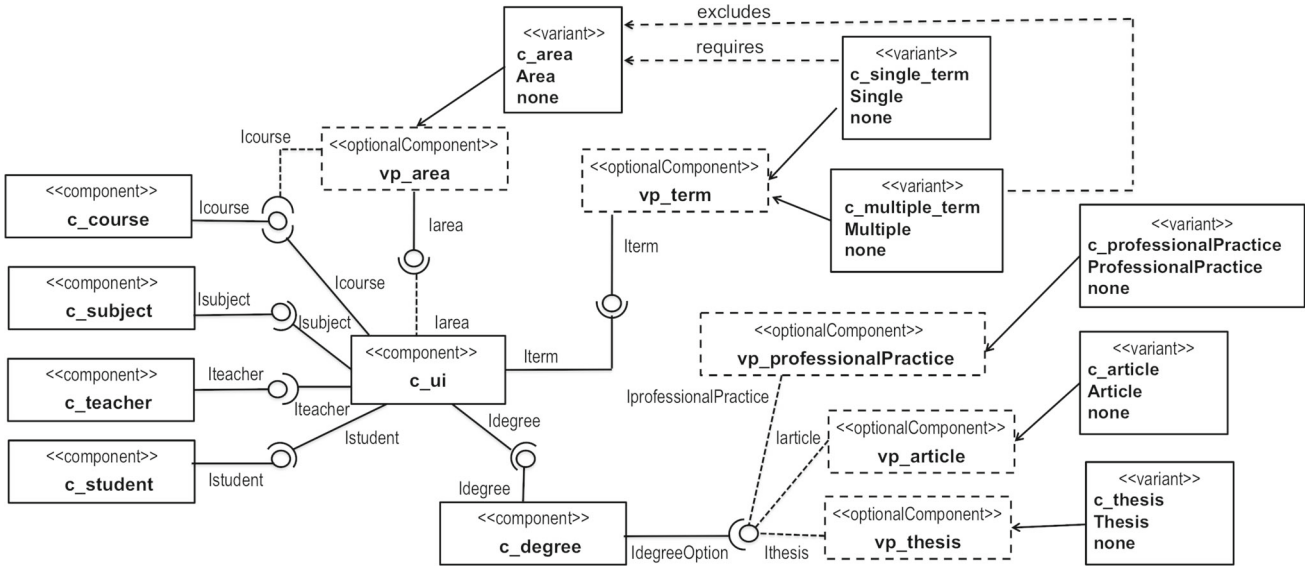


Fig. 3 SPL architecture of the scholar system without errors

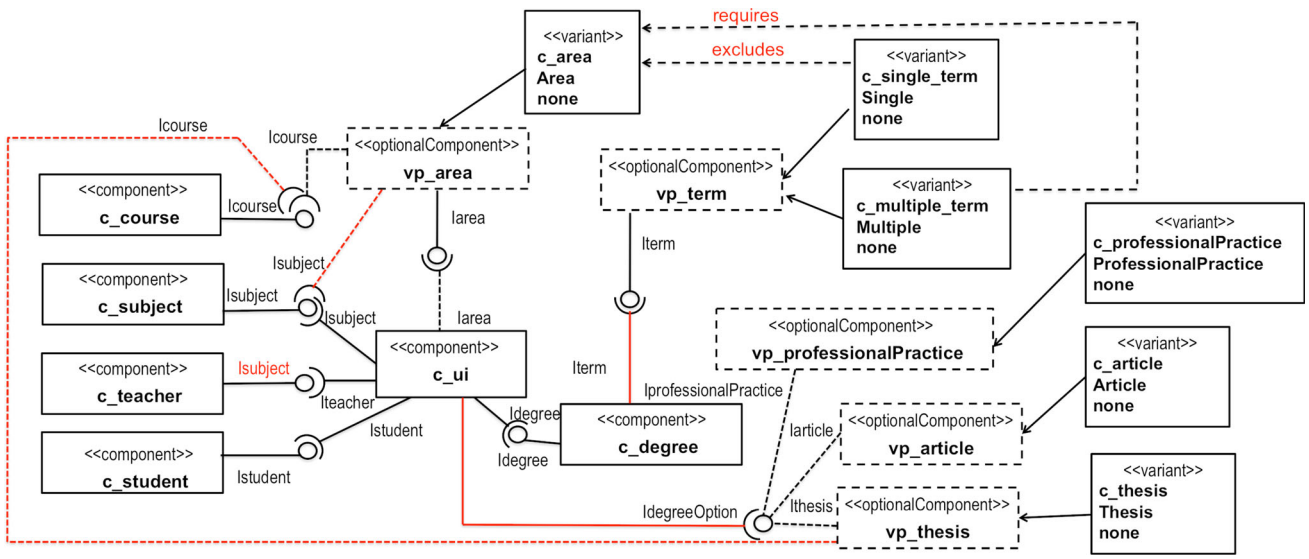


Fig. 4 SPL architecture of the scholar system with errors

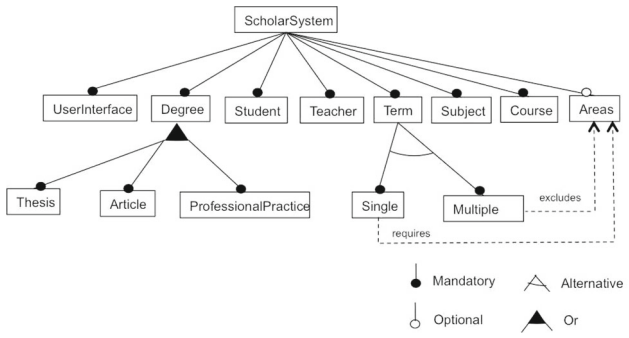


Fig. 5 Feature model tree of the scholar system

`ProfessionalPractice`, `Article`, and `Thesis`. Lastly, the variant `c_single_term` requires `c_area`, whereas the variant `c_multiple_term` excludes it. The variability of the product family of the scholar system is shown in the feature tree of Fig. 5.[6]

Now, consider the SPL architecture that is derived with errors (highlighted in red), as shown in Fig. 4. Such errors may not be easy to identify without performing an automated verification process. First of all, there are a number of *Mandatory* connections in the product architectures that do not appear in the SPL architecture, namely the connections between `c_ui` and `c_course`, and `c_ui` and `c_teacher`. Although in the latter case `c_ui` is connected to `c_teacher`, the facet interface name is wrong as it should be `Iteacher` instead of `Isubject`. Then, `c_ui` should be connected to `vp_term`, as shown in Fig. 3; however, this connection does not appear in Fig. 4. Also, `c_degree` should be connected to the following *Alternative* components: `vp_professionalPractice`, `vp_article`, and `vp_thesis`. Nevertheless, `c_degree` is not connected to any of them. On the other side, there are number of invalid connections in the SPL architecture, i.e., connections that are not defined in the correctly derived SPL architecture, which involve the connections between `c_degree` and `vp_term`, `c_ui` and `vp_article`, `c_ui` and `vp_thesis`, and `c_ui` and `vp_professionalPractice`, as shown in Figs. 3 and 4. Furthermore, `c_area` should be required by `c_single_term`; however, this component is required by `c_multiple_term` (see Figs. 3, 4).

Consequently, we illustrate in the following sections, via our motivating example, how we can verify consistency between the component interconnections of a product architecture and the component interconnections of the SPL architecture.

3 The OntoPAV framework

The main elements of the OntoPAV framework, and their relationships, are presented in the collaboration diagram depicted in Fig. 6. The ADL definitions of both the SPL architecture and the product architecture are the inputs of OntoPAV. These architectures can be defined in any ADL that complies with our meta-metamodel [6]. For instance, both the SPL architecture and the product architecture of the running example are defined in PL-Xelha, as shown in Figs. 3 and 2,[7] respectively. These ADL definitions are transformed to a populated ontology by the *Ontology Factory*. A populated ontology includes instances of concepts, defined in the Ontology, such as component, connector, interface, feature, and variant. The populated ontology is used to verify consistency between the component interconnections of the product architecture and the component interconnections of the SPL architecture. In order to achieve language independence, a specific factory is generated for a specific SPL language.

While the Ontology Factory is language-dependent (e.g., PL-Xelha requires a different implementation of this factory from that needed by Koala), the rest of the modules do not have any dependencies on the SPL language employed.

The Ontology Factory asks the *Verification Manager* to verify the populated ontology (step 1). The Verification Manager is in charge of coordinating the checks performed and also delivers the verification result. For this purpose, the Verification Manager first makes a query to identify invalid connections (step 2). Second, the Verification Manager checks consistency of the populated ontology (this reflecting consistency of the product architecture, step 3); both cases with the aid of the *Reasoning Engine*. In case one or more inconsistencies are detected, the Verification Manager asks the *Debugger* to find the origin of such inconsistencies (step 4). In order to find an inconsistency, the Debugger checks consistency on different subsets of the populated ontology to find the errors (step 5). The Debugger returns information with the origin of errors to the Verification Manager (step 6), which in turn informs the user about the verification result (step 7).

Verifying component interconnections involves checking that in the product architecture *Mandatory* connections are present, *Alternative* connections have one and only one connection, *OR* connections have at least one connection, and *Optional* connections are present in case the associated *Optional* variation point is instantiated. Our framework also checks that invalid connections are not present in the product architecture. That is, OntoPAV checks that only component connections defined in the SPL architecture are present in the product architecture whereby components and connec-

[6] Although the feature tree is not used by OntoPAV, we show it in order to improve the understandability of the motivating example.

[7] Only one architecture definition is taken as input of a product architecture of OntoPAV.

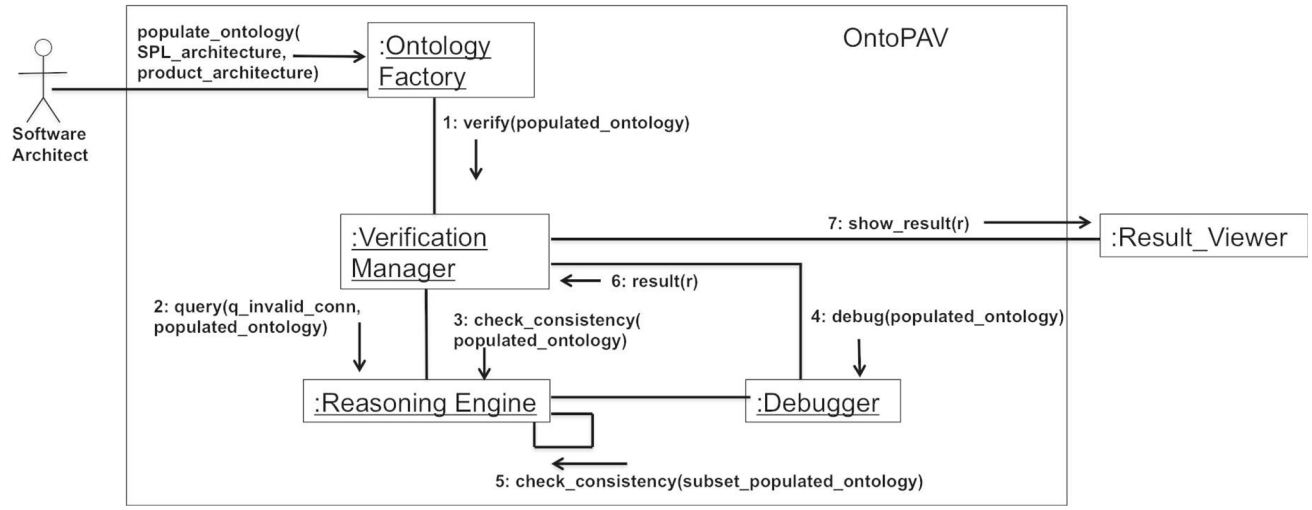


Fig. 6 Main elements of OntoPAV

tors in the product architecture are interconnected in the right sequential order and with the right interface types. Lastly, our framework checks that *requires/excludes* relationships are met.

OntoPAV can be used as follows. The software architect employs OntoPAV to check consistency between a reverse engineered SLP architecture and a product architecture. In case inconsistencies are detected, the software architect makes corrections to the SPL architecture and checks consistency again with the same product architecture. The architect makes corrections if inconsistencies are still detected, and so on. The same process is applied for each one of the product architectures from which the SPL architecture was reverse engineered. Future work considers verifying the SPL architecture with the product architectures altogether in order to facilitate the process of making corrections to the SPL architecture.

In the following section, we present details of our ontology and the verification rules we employ.

4 Verification rules and queries

An ontology is a formal and explicit specification of a shared conceptualization of a domain [28, 29]. We make use of OWL [30] to represent ontologies. OWL is based on *description logic (DL)* [31], which is a family of logic-based knowledge representation formalisms. We use the Manchester OWL Syntax [30] to illustrate the OWL descriptions used for the verification. This syntax is easy to read and write and is mainly employed by many ontology editing tools such as Protégé [30].

An ontology employs *individuals*, *classes*, and *object properties* to describe the domain. Individuals, which are the basic units in the domain, are instances of classes. Sets of individuals with similar characteristics conform to classes. Object properties represent binary relationships between individuals. Class constructors include the use of intersection, union, and complement operations as well as the *existential*[8] and *universal*[9] *quantifiers*, which in the Manchester OWL syntax are denoted by the keywords `some` and `any`, respectively. Class hierarchies can be defined by using the *subclass* property. A subclass is a subset of individuals of its parent class. A *necessary condition*[10] is represented as an anonymous superclass, whereas a *necessary and sufficient condition*[11] is represented as an equivalent class in the Manchester OWL Syntax format. Both conditions are also called *restrictions*. These and other features of OWL can be used to give a precise and unambiguous meaning to the descriptions of the domain.

The elements of our ontology are represented in the SPL architecture metamodel depicted in Fig. 7. Hence, our ontology includes components, connectors, interfaces, and connections, which are used to represent product architectures. The ontology additionally employs *Optional* components, *Optional* connectors, variants, and features to represent SPL architectures. The former two are also called variation points. The population process of our ontology in OWL is based on the work of Wang et al. [32] where ontologies are used to

[8] This means that if a relation exists, at least one relation should exist with the specified class at the right side of the relation.

[9] This means that if a relation exists, none relation or at most one relation could exist with the specified class at the right side of the relation.

[10] If an individual is a member of this class, then it is necessary to fulfill this condition, but we cannot say that if an individual fulfills this condition, then it must be a member of this class.

[11] If an individual is a member of this class, then it is necessary to fulfill this condition, and if an individual fulfills this condition, then it must be a member of this class.

model feature relationships. We populate an ontology as follows.

First, we make use of disjoint OWL classes to represent components, connectors, *Optional* components, *Optional* connectors, interfaces, variants, and features. Each of these classes keep a subclass relationship with the OWL class `Root`. For instance, the component `c_student` is represented as an OWL class named `c_student`. Variants keep a subclass relationship with their associated variation points (i.e., *Optional* components and *Optional* connectors).

Second, each class defined previously[12] has an equivalence statement, placed within its associated OWL class. Such an equivalence statement is a necessary and sufficient condition that binds the class to an existential restriction. For example, the existential restriction `hasC_student some c_student` is defined with an equivalence statement in the OWL class `c_student`.

Third, we represent component and connector connections with existential statements as follows. Consider that the receptacle interface $\texttt{I}_\texttt{i}$ of $\texttt{element}_\texttt{c}$ is connected to the facet interface $\texttt{I}_\texttt{j}$ of $\texttt{element}_\texttt{d}$. Such a connection is represented in the following form: $\texttt{element}_\texttt{c}$ − $\texttt{I}_\texttt{i}$`IsNextTo some` $\texttt{element}_\texttt{d}$ − $\texttt{I}_\texttt{j}$. For example, the following statement specifies that the receptacle interface `Istudent` of the component `c_ui` is connected to the facet interface also named `Istudent` of the component `c_student`.

```
(c_ui-IstudentIsNextTo some c_student
- Istudent)
```

Fourth, we included OWL restrictions in charge of associating features with components and connectors. For instance, the OWL class named `c_student` has the existential statement `hasStudent some Student`.

Fifth, we define a number of OWL restrictions for each class to specify *Mandatory*, *OR*, *Alternative*, *Optional*, and *requires/excludes* relationships (see below). In addition, we define a query mechanism, which is not part of a populated ontology, in charge of identifying invalid connections (see below).

4.1 Verification rules

The verification rules are defined in terms of OWL restrictions and are used to detect inconsistencies between a product architecture and the SPL architecture. Next, we define one rule for product architectures, followed by six rules for the SPL architecture.

Rule 1. *Define restrictions for elements of the Product architecture.* An element can be a connection, a component, or a connector. We define an OWL restriction that specifies the elements that are present and those that are not present in the product architecture.

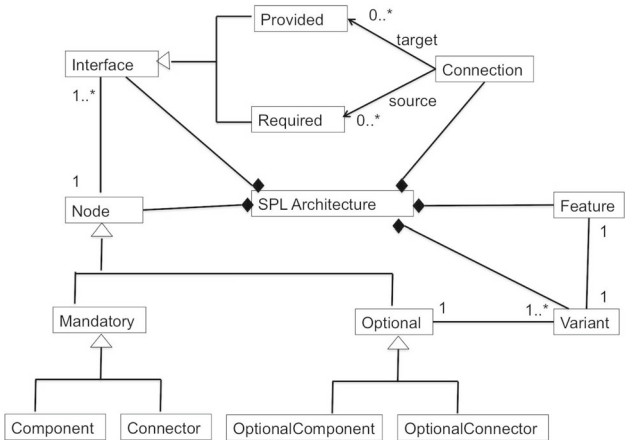


Fig. 7 SPL architecture metamodel

Definition 1 Let $\texttt{elem}_1$, $\texttt{elem}_2$,... $\texttt{elem}_\texttt{m}$ be the elements that are present in the SPL architecture but are not present in the product architecture and $\texttt{elem}_{\texttt{m+1}}$, $\texttt{elem}_{\texttt{m+2}}$,... $\texttt{elem}_{\texttt{m+x}}$ be the elements that are present in both the SPL architecture and the product architecture. Such elements are specified with a necessary and sufficient condition in the OWL class named `E_Product`[13] with the following form:

```
not (elem1) and not (elem2) and not....
and not (elemm) and (elemm+1) and (elemm+2)
and... and (elemm+x)
```

In the case of our running example, below we specify an excerpt of the connections in the SPL architecture with errors (Fig. 4) that are not present in the product architecture 1 (see Fig. 2a), which represent the connections of `c_degree` with `c_single_term` and `c_multiple_term`, respectively. It is also specified that the component `c_multiple_term` is not present:

```
not (c_degree-ItermIsNextTo some c
_single_term-Iterm)
not (c_degree-ItermIsNextTo some c_mu
ltiple_term-Iterm)
and not (hasC_multiple_term some c_mu
ltiple_term)
...
```

We present below an excerpt of the statement that specifies the connections that are present in the product architecture 1 (see Fig. 2a) involving the connections of `c_ui` with `c_teacher` and `c_subject`, respectively. It is also specified that the component `c_student` is present:

[12] This with the exception of the classes defined for features and interfaces.

[13] `E_Product` is placed at the same level of `Root` in the populated ontology hierarchy.

```
(c_ui-IteacherIsNextTo some c_teacher
-Iteacher)
and (c_ui-IsubjectIsNextTo some c_sub
ject-Isubject)
and (hasC_student some c_student)
...
```

Note that we have split the restriction statement into two parts for clarity only as both of them form a single restriction.

Rule 2. Define restrictions for Mandatory connections of the SPL architecture. A Mandatory connection in an SPL architecture takes place between two Mandatory elements and there is a one-to-one relationship.

Theorem 2 *Let $P = \{a_1,a_2,...,a_n\}$ be the set of Mandatory connections defined in an SPL architecture and let $Q = \{b_1,b_2,...,b_m\}$ be the set of Mandatory connections not defined in a product architecture, where $Q \subseteq P$. We associate the logical rule $rule_P$ with P and the logical rule $rule_Q$ with Q where:*

$rule_P \Rightarrow a_1$ and a_2 and... and a_n and $rule_Q \Rightarrow$ not (b_1) and not (b_2) and... and not (b_m)

If there exists at least one connection $a_i \in P$ and one connection $b_j \in Q$ such that $b_j = a_i$ and as a consequence $Q \neq \emptyset$, then a Mandatory condition is not met indicating that the product architecture is not consistent with the SPL architecture. In this case, $rule_P$ represents verification Rule 2 and $rule_Q$ represents a simplification of verification Rule 1 involving only the set of Mandatory connections not defined in the product architecture and omitting any other type of connections.

Proof If we assume that both $rule_P$ and $rule_Q$ are true and that there exists at least one connection $a_i \in P$ and one connection $b_j \in Q$ such that $b_j = a_i$ and as a consequence $Q \neq \emptyset$, then we have that $rule_P \Rightarrow Q = \emptyset$, which is a contradiction, and therefore, a *Mandatory* condition is not met.

Definition 2 Let $conn_i$ be a Mandatory connection of the SPL architecture. There can be multiple Mandatory connections, which we can represent as $conn_1$, $conn_2$, $conn_3$,..., $conn_n$. Such connections are specified with a necessary condition in the OWL class Root with the following form:

$(conn_1)$ and $(conn_2)$ and... and $(conn_n)$

Some of the Mandatory connections in our running example (see Fig. 3) are the connections of c_ui with c_student and c_teacher, which are represented as follows:

```
(c_ui-IstudentIsNextTo some c_student
-Istudent)
and (c_ui-IteacherIsNextTo some c_tea
cher-Iteacher)
```

Rule 3. Define restrictions for OR connections of the SPL architecture. *OR* connections involve variation points whose variant connections have an *OR* relationship.

Theorem 3 *Let $P = \{a_1,a_2,...,a_n\}$ be the set of OR connections defined in an SPL architecture and let $Q = \{b_1,b_2,...,b_m\}$ be the set of OR connections not defined in a product architecture, where $Q \subseteq P$. We associate the logical rule $rule_P$ with P and the logical rule $rule_Q$ with Q where:*

$rule_P \Rightarrow a_1$ or a_2 or.... or a_n

and $rule_Q \Rightarrow$ not (b_1) and not (b_2) and.... and not (b_m)

If there exists for each connection $a_i \in P$ one connection $b_j \in Q$ such that $b_j = a_i$ and as a consequence $Q = P$, then an OR condition is not met indicating that the product architecture is not consistent with the SPL architecture. Here, $rule_P$ represents verification Rule 3 and $rule_Q$ represents a simplification of verification Rule 1 involving only the set of OR connections not defined in the product architecture and omitting any other type of connections.

Proof If we assume that both $rule_P$ and $rule_Q$ are true and that for each connection $a_i \in P$ there exists one connection $b_j \in Q$ such that $b_j = a_i$ and as a consequence $Q = P$, then we have that $rule_P \Rightarrow Q \neq P$, which is a contradiction, and therefore, an *OR* condition is not met.

Definition 3 Let $variantConn_i$ be an *OR* connection of a variation point of the SPL architecture. There can be two or more *OR connections associated with a variation point*, which we can represent as $variantConn_1$, $variantConn_2$, $variantConn_3$,..., $variantConn_n$. Such *OR connections* are specified with a necessary condition in the OWL class Root with the following form:

$(variantConn_1)$ or $(variantConn_2)$ or.... or $(variantConn_n)$

For example, the following statement specifies the *OR* relationships of the connections of c_degree with the instantiation of the variation points vp_professionalPractice, vp_article, and vp_thesis (see Fig. 3).

```
(c_degree-IdegreeOptionIsNextTo some
c_professionalPractice-Iprofessional
Practice)
or (c_degree-IdegreeOptionIsNextTo some
c_article-Iarticle)s
or (c_degree-IdegreeOptionIsNextTo some
c_thesis-Ithesis)
```

Rule 4. Define restrictions for Alternative connections of the SPL architecture. Alternative connections involve variation points whose variant connections have an Alternative relationship.

Theorem 4 *Let $P = \{a_1,a_2,...,a_n\}$ be the set of Alternative connections defined in an SPL architecture and let $Q_1 = \{b_1,b_2,...,b_m\}$ be the set of Alternative connections not defined in a product architecture and $Q_2 = \{b_{m+1},b_{m+2},...,b_{m+x}\}$*

be the set of Alternative connections defined in a product architecture, where $P = Q_1 \cup Q_2$. We associate the logical rule $rule_P$ with P and the logical rule $rule_Q$ with Q where:

$rule_P \Rightarrow (not\ (a_1)\ and\ not\ (a_2)\ and\ not\ (a_3)\ and\ not...\ and\ not\ (a_{n-1})\ and\ a_n)$
$or\ (not\ (a_1)\ and\ not\ (a_2)\ and\ not\ (a_3)\ and\ not...\ and\ (a_{n-1})\ and\ not\ a_n)$
$or\ (not\ (a_1)\ and\ not\ (a_2)\ and\ not\ (a_3)\ and\ not...\ and\ (a_{n-2})\ and\ not\ (a_{n-1})\ and\ not\ a_n)$
$...$
$(or\ (a_1)\ and\ not\ (a_2)\ and\ not\ (a_3)\ and\ not$ ...and not $(a_{n-1})\ and\ not\ (a_n))$

and $rule_Q \Rightarrow not\ (b_1)\ and\ not\ (b_2)\ and\ not$ ....and not $(b_m)\ and\ (b_{m+1})\ and\ (b_{m+2})\ and...$ and (b_{m+x})

In this case, $rule_P$ represents verification Rule 4 and $rule_Q$ represents a simplification of verification Rule 1 involving only the set of Alternative connections not defined and those that are defined in the product architecture and omitting any other type of connections.

Theorem 4 a. *If we have $Q_2 = \emptyset$ and that for each $a_i \in P$ there is a $b_j \in Q_1$* such that $b_j = a_i$ and *as a consequence $Q_1 = P$, then we have that an Alternative condition is not met indicating that the product architecture is not consistent with the SPL architecture*.

Proof If we assume that both $rule_P$ and $rule_Q$ are true and $Q_2 = \emptyset$ and that for each $a_i \in P$ there is a $b_j \in Q_1$ such that $b_j = a_i$ and as a consequence $Q_1 = P$, then we have that $rule_P \Rightarrow Q_1 \neq P$, which is a contradiction, and therefore, an *Alternative* condition is not met.

Theorem 4 b. *If $Q_1 \neq P$ and there exists at least one pair of connections* a_i and $a_k \in P$ and one pair of connections b_j and $b_1 \in Q_2$ such that $b_j = a_i$ and $b_1 = a_k$ where i and $k \in \{1, 2, ..., n\}, i < k, j$ *and* $l \in \{m+1, m+2, ..., m+x\}$ *and* $j < l$ *and as a consequence* $| Q_2 | > 1$, *then we have that an Alternative condition is not met indicating that the product architecture is not consistent with the SPL architecture*.

Proof If we assume that both $rule_P$ and $rule_Q$ are true and $Q_1 \neq P$ and that there exists one pair of connections a_i and $a_k \in P$ and one pair of connections b_j and $b_1 \in Q_2$ such that $b_j = a_i$ and $b_1 = a_k$ and as a consequence $| Q_2 | > 1$, then we have that $rule_P \Rightarrow | Q_2 | = 1$, which is a contradiction, and therefore, an *Alternative* condition is not met.

Definition 4 Let $variantConn_i$ be an Alternative connection of a variation point of the SPL architecture. There can be two or more Alternative connections, which we can represent as $variantConn_1, variantConn_2, variantConn_3$,..., $variantConn_n$. Such Alternative connections are specified with a necessary condition in the OWL class of the variation point with the following form:

(not ($variantConn_1$) and not ($variantConn_2$) and not ($variantConn_3$) and not... and not ($variantConn_{n-1}$) and $variantConn_n$)
or (not ($variantConn_1$) and not ($variantConn_2$) and not ($variantConn_3$) and not ... and ($variantConn_{n-1}$) and not $variantConn_n$)
or (not ($variantConn_1$) and not ($variantConn_2$) and not ($variantConn_3$) and not... and ($variantConn_{n-2}$) and not ($variantConn_{n-1}$) and not $variantConn_n$)
...
or (($variantConn_1$) and not ($variantConn_2$) and not ($variantConn_3$) and not... and not ($variantConn_{n-1}$) and not ($variantConn_n$))

For example, the following statement specifies the *Alternative* relationships of the connections of the component c_ui with the instantiation of the variants c_single_term and c_multiple_term (see Fig. 3).

(not (c_ui-ItermIsNextTo some c_single_term-Iterm)
and (c_ui-ItermIsNextTo some c_multiple_term-Iterm))
or (not (c_ui-ItermIsNextTo some c_multiple_term-Iterm)
and (c_ui-ItermIsNextTo some c_single_term-Iterm))

Rule 5. *Define restrictions for Optional connections associated with an Optional variation point in the SPL architecture.* This rule involves two parts. Part 1 regards a restriction indicating the *Optional* variation point is instantiated. Part 2 defines the restrictions of the connections involved. There are four different cases for Part 2. This depending on whether the *Optional* connections are defined between (1) the *Optional* variation point and a *Mandatory* component, (2) two *Optional* variation points, (3) the *Optional* variation point and an *OR* variation point, and (4) the *Optional* variation point and an *Alternative* variation point. Part 2 involves applying Rule 2 for cases one and two; and Rule 3 and Rule 4 for case three and case four, respectively. Note that the clauses also include the case in which an *Optional* variation point is not instantiated prescribing the *Optional* connections must not be present.

Definition 5.1 *Let* $opConn_i$ *be an Optional connection of the SPL architecture between an Optional variation point* $opVarPoint_j$ *and a Mandatory component. The instantiation of the Optional variation point* $instantiation_j$ *(Part 1) is specified together with such a connection (Part 2)*

with a necessary condition in the OWL class Root *with the following form:*

((instantiation$_j$) and (opConn$_i$)) or
(not (instantiation$_j$)and not (opConn$_i$))

There are two *Optional* connections associated with the *Optional* variation point vp_area that is instantiated with c_area in our running example (see Fig. 3). These connections are between c_ui and vp_area and between vp_area and c_course. As an example, consider the constraint for the latter:

((hasC_area some c_area)and
(c_area-IcourseIsNextTo some c_course
−Icourse))
or
(not (hasC_area some c_area)and not
(c_area-IcourseIsNextTo some c_course
−Icourse))

Definition 5.2 *Let* opConn$_i$ *be an Optional connection of the SPL architecture between two Optional variation points*, namely opVarPoint$_j$ *and* opVarPoint$_{j+1}$. *The instantiation of the Optional variation points* instantiation$_j$, instantiation$_{j+1}$ (Part1) are specified together with such a connection (Part 2) with a necessary condition in the OWL class Root *with the following form:*

((instantiation$_j$) and (instantiation$_{j+1}$)
and (opConn$_i$)) or
(not (instantiation$_j$) or not (insta
ntiation$_{j+1}$) and
(not (opConn$_i$))

Definition 5.3 *Let* opConn$_i$ *be an Optional connection of the SPL architecture between an Optional variation point* opVarPoint$_j$ *and an OR variation point. There can be multiple Optional connections, which we can represent as* opConn$_1$, opConn$_2$, opConn$_3$,..., opConn$_n$. *The instantiation of the Optional variation point* instantiation$_j$ *(Part 1) is specified together with such connections (Part2) with a necessary condition in the OWL class* Root *with the following form:*

((instantiation$_j$) and (not (not
(opConn$_1$) and not (opConn$_2$) and not....
and not (opConn$_n$))))
or
(not (instantiation$_j$) and (not
(opConn$_1$) and not (opConn$_2$) and not....
and not (opConn$_n$)))

Definition 5.4 *Let* opConn$_i$ *be an Optional connection of the SPL architecture between an Optional variation point* opVarPoint$_j$ *and an Alternative variation point. There can be multiple Optional connections, which we can represent as* opConn$_1$, opConn$_2$, opConn$_3$,..., opConn$_n$. *The instantiation of the Optional variation point* instantiation$_j$ *(Part 1) is specified together with such connections (Part2) with a necessary condition in the OWL class* Root *with the following form:*

((instantiation$_j$) and ((not
(opConn$_1$) and not (opConn$_2$) and not
(opConn$_3$) and not... and not (opConn$_{n-1}$)
and opConn$_n$)
or (not (opConn$_1$) and not (opConn$_2$) and
not (opConn$_3$) and not... and (opConn$_{n-1}$)
and not opConn$_n$)
or (not (opConn$_1$) and not (opConn$_2$) and not

(opConn$_3$) and not... and (opConn$_{n-2}$) and not
(opConn$_{n-1}$) and not opConn$_n$)
...
or ((opConn$_1$) and not (opConn$_2$) and not
(opConn$_3$) and not... and not (opConn$_{n-1}$)
and not (opConn$_n$))))
or
(not (instantiation$_j$) and (not (opConn$_1$)
and not (opConn$_2$) and not.... and not
(opConn$_n$)))

Rule 6. *Define requires restrictions in the SPL architecture.* A *requires* restriction in an SPL architecture takes place between two or more elements whereby the former requires the presence of the latter elements. An element can be either a component or a connector.

Theorem 5 *Let* $P = \{a_1, a_2, ..., a_n\}$ *be the elements required by element e and let* $Q = \{b_1, b_2, ..., b_m\}$ *be the set of elements not defined in a product architecture. We associate the logical rule* $rule_P$ *with* P *and the logical rule* $rule_Q$ *with* Q *where:*

$rule_P \Rightarrow a_1$ and a_2 and... and a_n

and $rule_Q \Rightarrow$ not (b_1) and not (b_2) and... and not (b_m)

If there exists at least one element $a_i \in P$ *and one element* $b_j \in Q$ *such that* $b_j = a_i$ *and as a consequence* $P \cap Q \neq \emptyset$, *then a requires condition is not met indicating that the product architecture is not consistent with the SPL architecture. In this case,* $rule_P$ *represents verification Rule 6 and* $rule_Q$ *represents a simplification of verification Rule 1 involving only the set of elements not defined in the product architecture and omitting any other type of elements.*

Proof If we assume that both $rule_P$ and $rule_Q$ are true and that there exists one element $a_i \in P$ and one element $b_j \in Q$ such that $b_j = a_i$ and as a consequence $P \cap Q \neq \emptyset$, then we have that $rule_P \Rightarrow P \cap Q = \emptyset$, which is a contradiction, and therefore, a *requires* condition is not met.

Definition 5 Let element$_i$ be a required *element of the element* element$_j$ *of the SPL architecture*. There can be multiple required elements, which we can represent as

$element_1$, $element_2$, $element_3$,..., $element_n$. Such required elements are specified with a necessary condition in the OWL class of the element $element_j$ with the following form:

$(element_1)$ and $(element_2)$ and... and $(element_n)$

For example, the following statement specifies the *requires* constraint of the component c_single_term (see Fig. 3) that requires the component c_area as follows:

(hasC_area some c_area)

Rule 7. Define excludes restrictions in the SPL architecture. An *excludes* restriction in an SPL architecture takes place between two or more elements whereby the former excludes the presence of the latter elements. An element can be either a component or a connector.

Theorem 6 *Let $P = \{a_1,a_2,...,a_n\}$ be the elements excluded by an element e and let $Q = \{b_1,b_2,...,b_m\}$ be the set of elements defined in a product architecture. We associate the logical rule $rule_P$ with P and the logical rule $rule_Q$ with Q where:*

$rule_P \Rightarrow not\,(a_1)$ and not (a_2) and... and not (a_n)

and $rule_Q \Rightarrow (b_1)$ and (b_2) and.... and (b_m)

If there exists at least one element $a_i \in P$ and one element $b_j \in Q$ such that $b_j = a_i$ and as a consequence $P \cap Q \neq \emptyset$, then an excludes condition is not met indicating that the product architecture is not consistent with the SPL architecture. In this case, $rule_P$ represents verification Rule 7 and $rule_Q$ represents a simplification of verification Rule 1 involving only the set of elements that are defined in the product architecture.

Proof If we assume that both $rule_P$ and $rule_Q$ are true and that there exists at least one element $a_i \in P$ and one element $b_j \in Q$ such that $b_j = a_i$ and as a consequence $P \cap Q \neq \emptyset$, then we have that $rule_P \Rightarrow P \cap Q = \emptyset$, which is a contradiction, and therefore, an *excludes* condition is not met.

Definition 6 *Let $element_i$ be an excluded element of the element $element_j$ of the SPL architecture. There can be multiple excluded elements, which we can represent as $element_1$, $element_2$, $element_3$,..., $element_n$. Such excluded elements are specified with a necessary condition in the OWL class of the element $element_j$ with the following form:*

not $(element_1)$and not $(element_2)$and... and not $(element_n)$

For example, the following statement specifies the *excludes* constraint of the component c_multiple_term (see Fig. 3) that excludes the component c_area as follows:

not (hasC_area some c_area)

4.2 Query invalid connections

We employ a query that allows us to detect the presence of connections not defined in the SPL architecture (see step 2 of Fig. 6).

Query 1. *Query the connections present in the Product architecture that are not part of the SPL architecture.* Let spl_conn be the set of connections defined in the SPL architecture that represent the set of valid connections and p_conn be the set of the connections defined in the product architecture. This query retrieves the set of invalid connections $\{invalid\_conn_1, invalid\_conn_2, invalid\_conn_3,..., invalid\_conn_n\}$ where $invalid\_conn_i \in$ p_conn and $invalid\_conn_i \notin$ spl_conn.

An example of this query is presented in the next section.

5 The verification process

The verification process involves two phases. In the first phase, the Verification Manager employs Query 1 to identify invalid connections. In the second phase, the Verification Manager checks whether the populated ontology is inconsistent in order to detect any errors related to *Mandatory* connections, *OR* connections, *Alternative* connections, *Optional* connections, and *requires/excludes* relationships. In case the populated ontology is inconsistent, the debugging process takes place. The Reasoning Engine is able to detect when a populated ontology is inconsistent; however, such an engine is unable to indicate what is causing the inconsistency. For instance, the Reasoning Engine is able to detect that there is an error when a *Mandatory* connection is missing in the product architecture. However, the Reasoning Engine does not provide any clue about the type of error nor the elements that may be involved in this error. The same happens with *OR* connections, *Alternative* connections, *Optional* connections, and *requires/excludes* relationships. This issue makes it difficult to find the origin of errors in an inconsistent product architecture. In order to tackle this issue we make use of the Debugger, which, by employing multiple verification iterations of subsets of the populated ontology can find the origin of an error. Each verification phase is described further below.

The verification information presents the connections in the SPL architecture that have inconsistencies. As mentioned earlier, a connection is represented by $element_c$ − I_iIsNextTo some $element_d$ − I_j, where $element_c$ and $element_d$ are components, and I_i and I_j are their related interfaces. For example, the following verification result informs the users that there is an inconsistency with the connection between the receptacle interface Istudent of

the component c_ui and the facet interface also named Istudent of the component c_student.

```
(c_ui-IstudentIsNextTo some c_student
-Istudent)
```

5.1 Identify invalid connections

The Verification Manager employs Query 1 to identify the connections defined in the product architecture that are not specified in the SPL architecture. Regarding our running example, the Verification Manager checks consistency between the product architecture 1 (Fig. 2a) and the SPL architecture with errors (Fig. 4). Hence, Query 1 returns the following invalid connections (i.e., connections in the product architecture 1 that are not present in the SPL architecture with errors):

```
c_degree-IdegreeOptionIsNextTo some c
_thesis-Ithesis
c_degree-IdegreeOptionIsNextTo some c
_article-Iarticle
c_ui-ItermIsNextTo some c_single_term
-Iterm
c_ui-IcourseIsNextTo some c_course
-Icourse
c_ui-IteacherIsNextTo some c_teacher
-Iteacher
```

The former represents the connection between c_degree and c_thesis. In order to fix this error in the SPL architecture (Fig. 4), c_degree needs to be connected to VP_thesis as c_thesis is a variant. The second case represents the connection between c_degree and c_article. Similarly, this error can be corrected by connecting c_degree to VP_article. The third case is the connection between c_ui and c_single_term, which can be fixed by connecting c_ui to VP_term. The fourth case is the connection between c_ui and c_course, which can be easily repaired by connecting the former to the latter. The last case is a *Mandatory* connection whose facet interface is defined as Isubject by the SPL architecture, whereas the product has Iteacher instead. This is simply rectified by renaming the facet interface of the SPL architecture to Iteacher.

5.2 Identify errors with *Mandatory*, *OR*, *Alternative*, and *Optional* connections, and with *requires/excludes* relationships

In case the populated ontology is inconsistent, the following debugging process is carried out. First, the connection patterns of the SPL architecture are identified. In our running example, we have connection patterns such as IdegreeOptionIsNextTo, ItermIsNextTo, and IcourseIsNextTo, among others. Such patterns involve the receptacle interface that takes place in a connection. In this way the Debugger can group the set of connections errors that take place.[14]

For each connection pattern, *i* the Debugger constructs a node $\texttt{E\_Debug\_step\_by\_step}_i$. In case a node is inconsistent, the Debugger raises an error showing the connections involved in such a node. In our running example (see Figs. 2a and 4), the Debugger identifies the following *Mandatory* connection error:

```
1.....c_ui-IteacherIsNextTo some c_te
acher-Isubject
```

The Debugger identifies that a *Mandatory* connection is missing from c_ui to c_teacher (line 1). Although in the product architecture, there is a connection between c_ui and c_teacher, the facet interface is named Iteacher in the product architecture 1 while the name defined in the SPL architecture with errors is Isubject. This error is also detected by Query 1[15]. This error was already fixed above.

The Debugger identifies the following *OR* connection error:

```
2.....c_ui-IdegreeOptionIsNextTo some
c_article-Iarticle, c_ui-IdegreeOption
IsNextTo some c_professionalPractice
-IprofessionalPractice, c_ui-Idegree
OptionIsNextTo some c_thesis-Ithesis
```

There is an *OR* connection error between c_ui and the following three variation points (line 2): vp_profe ssionalPractice, vp_article, and vp_thesis. In the product architecture 1, c_ui is not connected to any of the three variants related to these variation points, namely c_professionalPractice, c_article, or c_thesis (see Fig. 2a), while at least one connection should be present according to the SPL architecture with errors (see Fig. 4). This error can be simply repaired by removing the *OR* connection from c_ui in the SPL architecture (Fig. 4).

In addition, the Debugger identifies the following *Alternative* connection error:

```
3.....c_degree-ItermIsNextTo some
c_multiple_term-Iterm,
c_degree-ItermIsNextTo some
c_single_term-Iterm
```

There is an *Alternative* connection error between the component c_degree and the variation point vp_term (line 3). The reason of this error is c_degree is not

[14] We assume that each connection between two elements has a different connection pattern. That is, a receptacle interface name must be a singleton in the SPL architecture, i.e., there cannot be two or more receptacle interfaces with the same name unless they are connecting other instances of the same two elements and in the same order.

[15] Query 1 only detects invalid connections, but in this case it happened that this invalid connection (one interface name is not consistent) is also a *Mandatory* connection between the two components.

connected to any variant of this variation point, namely `c_single_term` and `c_multiple_term` (as shown in Fig. 2a) while the SPL architecture with errors indicates that `c_degree` should be connected to one of them (as shown in Fig. 4). This error can be easily fixed by removing the *Alternative* connection from `c_degree` in the SPL architecture (Fig. 4).

The Debugger also identifies the following *Optional* connection error:

```
4.....c_area-IsubjectIsNextTo some
c_subject-Isubject
```

There is an *Optional* connection error between the *Optional* variation point `vp_area` and the component `c_subject` (line 4). Given that the *Optional* feature *Area* was selected, the *Optional* variation point should be instantiated with the component `c_area` as well as a connection between the components `c_area` and `c_subject`, `c_area` and `c_course`, and `c_ui` and `c_area` but the connection between `c_area` and `c_subject` is not present (as shown in Fig. 2a); however, the latter connection is included in the SPL architecture with errors (as shown in Fig. 4). This error can be simply corrected by removing the *Optional* connection between `vp_area` and `c_subject` in the SPL architecture (Fig. 4).

Lastly, the Debugger identifies the following *requires* constraint is not met:

```
5.....hasC_area some c_area
```

According to the SPL architecture with errors, when the variant `c_single_term` is selected, it excludes the variant `c_area` to be selected (as shown in Fig. 4). However, in the product architecture 1 `c_single_term` is present as well as `c_area` (as shown in Fig. 2a). This error can be fixed by first removing the *requires* constraint in the SPL architecture (Fig. 4). Then, we need to reverse engineer the correct requires constraint, if any, from the product architectures and place it in the SPL architecture.

6 Evaluation

In this section, we describe the evaluation of both the accuracy and scalability of our approach. We also present a discussion of the results and discuss the limitations and threats to validity of our work.

6.1 Prototype implementation

We used a tool [6] to visually edit both the SPL architecture and the product architecture in PL-Xelha. Regarding the implementation of the Ontology Factory, we used Acceleo [33] to transform the PL-Xelha models of the SPL architecture and product architecture to text in Turtle [34]. The Turtle syntax is supported by most ontology-based reasoning engines and its syntax is less verbose and more user-friendly than other formats. As mentioned earlier, we used Pellet [25] as the Reasoning Engine. The Verification Manager and the Debugger were implemented in Java. Lastly, Query 1 was implemented in Java as part of the Verification Manager. As mentioned earlier, the Ontology Factory is language-dependent (e.g., PL-Xelha requires a different implementation of this factory from that needed by Koala). However, the rest of the modules do not have any dependencies on the SPL language employed.

6.2 Accuracy evaluation

We used *mutation testing* [35] to evaluate the accuracy of our verification approach. Mutation testing has been employed for evaluating the adequacy of test suites [36–38] and also for guiding the generation of test cases [38–40]. Mutation testing has also been used to test feature models in a software product line [41, 42] as well as a means to generate test configurations for an SPL [43] and assess the ability of test suites to detect errors in feature models [44]. *Mutants* are software artifacts that have artificial errors injected. Such artificial errors are called *mutations*. *Mutation operators* are mutation rules used to inject mutations. In case the test suit is able to detect a mutation (i.e., a test case fails), it is said that a mutant is *killed*. *Equivalent mutants* are those whose functionality is equivalent to the original artifacts. The ratio of mutants detected by the test cases is called the *mutation score*. In this paper, the mutation score represents the accuracy of our verifier to detect errors. We calculated the mutation score as follows:

```
m_score = mutants_killed / (mutants
- e_mutants) * 100
```

where `m_score` is the mutation score, `mutants_killed` are the number of mutants killed, `mutants` are the number of selected mutants, and `e_mutants` are the number of equivalent mutants. Equivalent mutants are subtracted from the number of mutants since equivalent mutants are not able to produce errors.

We employed mutation testing to systematically derive test cases for our running example.[16] Based on the test cases generated, we evaluated the ability of our verification approach to detect inconsistencies between an SPL architecture and a product architecture. We achieved this by conducting the following steps:

[16] The number of product architectures of the SPL architecture employed by the evaluation is 15. This given that our running example was extended with an *Optional* connection between the *Optional* component `vp_term` and the *Optional* component `vp_area` (see mutation operators 24 and 28).

1. *Define mutation operators.* In order to define mutation operators, we identified what can be changed in an SPL architecture. These operators were defined based on the operations *add*, *remove*, and *change*, which are the basic operations needed to introduce changes. Then we applied these operations to the different elements of an SPL architecture such as *Mandatory* component, *Optional* component, *OR* group, and *Alternative* group. A partial list of the mutation operators we applied to the SPL architecture of our running example (see Fig. 3) is presented in Table 1. The full list of the mutation operators is given in "Appendix A" in Table 7.
2. *Generate the mutants.* We manually applied the mutation operators to the SPL architecture of our running example to obtain the mutants. The generated mutants are SPL architectures with changes introduced.
3. *Generate test cases.* We generated a test case for each mutant. Thus, a test case is a product architecture that is consistent with a mutant (i.e., a modified SPL architecture). There may be multiple test cases associated with a single mutant. Some of these test cases could be mutant killers and some others could be not. We randomly selected only one mutant killer.
4. *Evaluate the verifier with the test cases.* The steps to evaluate the verifier are as follows. First, we verify the test cases against the mutants where no verification errors should be raised. Contrarily, the test case is erroneous and must be corrected until no errors are detected by the verifier. Second, the test cases are verified with respect to the original SPL architecture of the running example. In this case, it is expected that the verifier detects one or more errors related to the changes made by the associated mutation operator.

The results of the mutation testing are shown in Table 2. The first column shows the total number of mutants that can be produced by applying each mutation operator. The second column presents the number of mutants that were actually generated. For example, in the case of the operator "12. Change the instantiation relationship of an Optional component," there are 30 possible mutants. However, we only generated 20% of them (i.e., 6) to reduce the effort of manually generating them. We also selected 20% or higher of the mutants in the case of the operators 17, 18, 19, 20, 31, and 33. It has been shown elsewhere [45] that randomly selecting 20% of the mutants results in a fault loss of only about 16%. The percentage of selected mutants is shown in the third column. Then, the fourth column shows the cases when a mutant is killed. This happens when a test case fails, as earlier mentioned. Lastly, the fifth column shows the equivalent mutants, which are those mutants that are equivalent to the original SPL. All the mutants generated by the operators 6, 7, 10, and 11 are equivalent mutants. In the case of operator 8, only one mutant out of five is mutant equivalent. In the case of operator 12, there are two mutants out of six that are mutant equivalent. We obtained 116 mutants killed, which is exactly the same number that results from subtracting the number of equivalent mutants (i.e., 14) to the number of selected mutants (i.e., 130). As a result, our verifier obtained a mutation score of 100%.

6.3 Scalability evaluation

We carried out a number of experiments to test the scalability of our approach. Each experiment was repeated 30 times and the average execution time of these runs was considered.

We carried out our experiments on a MacBook Pro Retina Intel Core i7 at 2.5 GHz with 6 MB of L3 cache memory and 16 GB of RAM running Mac OS X, version 10.13.2. The scalability of the prototype was evaluated for different sizes of SPL architectures. In our experiment, we tested the scalability of the Ontology Factory, Verification Manager, and Debugger. The Reasoning Engine was indirectly tested by the latter two as these modules perform queries over the ontologies.

In order to test the scalability of our framework, we implemented a Java program in charge of generating random SPL architectures. We chose using random architectures as no industrial SPL architectures are freely available and with random architectures we have more flexibility to decide the size of the SPL architecture to test. This program also generates a set of product architectures for each SPL architecture that corresponds to a number of random selections from the set of valid configurations. Errors were randomly introduced in the generated product architectures, which consisted of removing an arbitrary *Mandatory* connection and an arbitrary *Alternative* connection from the product architectures. In this way, the Debugger was forced to check the validity of *Mandatory, OR*, and *Alternative* connections.

In addition, the architecture generator program can produce SPL architectures of different sizes that follow a proportion similar to our running example and involves the same number of *Alternative* variation points as the number of *Mandatory* elements and *requires/excludes* relationships altogether. Such relationships are approximately the fifth part of *Mandatory* elements. There are twice as many *OR* variation points as there are *Alternative* variation points and there are two to seven variants associated with each variation point (the specific number of variants was randomly selected). *Alternative* variation points were connected consecutively in line (one to another) in order to test the worst case scenario (i.e., a case with higher computing complexity). This is because Rule 3 statements grow exponentially as the number of variants increases; hence, scalability is affected accordingly (see below).

Table 1 Partial list of mutation operators

Operator	Example
1. Remove a Mandatory component	Remove the Mandatory component `c_student` from the SPL architecture
2. Change a Mandatory component to an Optional component	Convert the Mandatory component `c_student` to an Optional component in the SPL architecture
3. Change the name of a Mandatory component	The name of the Mandatory component `c_student` is changed to `c_studentV2`
4. Add an Optional component to an OR group	The Optional component `vp_area` is disconnected from `c_ui` and connected to the receptacle `IdegreeOption` of the component `c_degree` as another degree option
5. Add an Optional component to an Alternative group	The Optional component `vp_area` is removed and its associated variant `c_area` is added as an alternative of the Optional component `vp_term`
6. Remove an Optional component from an OR group	Remove both the Optional component `vp_thesis` and its associated variant `c_thesis` from the OR group in the SPL architecture
7. Remove an Optional component from an Alternative group	The variant `c_single_term` is removed from the Alternative group in the SPL architecture
8. Remove an Optional component and its associated variants	Remove both the variation point `vp_area` and its associated variant `c_area` from the SPL architecture
9. Change the name of a variant	Change the name of the variant `c_thesis` by `c_thesisV2`
10. Change an Optional component to a Mandatory component	The Optional component `vp_area` and its associated variant `c_area` are transformed to a Mandatory component and the requires/excludes relationships are eliminated
11. Change the name of an Optional component	The name of the variant `c_area` is changed to `c_areaV2`. Another example is changing the variant `c_thesis` to `c_thesisV2`
12. Change the instantiation relationship of an Optional comp	The variants `c_thesis` and `c_article` are interchanged so that now they instantiate the Optional components `vp_article` and `vp_thesis`, respectively
13. Remove an OR group relationship	The Optional components `vp_professionalPractice`, `vp_article` and `vp_thesis` are removed along with their associated variants `c_professionalPractice` `c_article`, and `c_thesis`
14. Change OR group relationship to Alternative group relationship	The OR group relationship of our running example is transformed to an Alternative group relationship
15. Remove an Alternative group relationship	The Optional component `vp_term` is removed along with its variants `c_single_term` and `c_multiple_term`
16. Change Alternative group relationship to OR group relation	The Alternative group relationship of our running example is transformed to an OR group relationship
17. Add a connection between two Mandatory components	A receptacle of the component `c_student` is connected to a facet of the component `c_degree`
18. Add a connection between a Mandatory component and an Optional component	A receptacle of the component `c_course` is connected to a facet of the Optional component `vp_area`
19. Add a connection between an Optional component and a Mandatory component	Connect a receptacle of the Optional component `vp_area` to the facet of the component `c_course`
20. Add a connection between two Optional components	Connect a receptacle of the Optional component `vp_area` to a facet of the Optional component `vp_term`
21. Remove a connection between two Mandatory components	The connection between the components `c_ui` and `c_course` is removed
22. Remove a connection between a Mandatory comp and Op comp	The connection between the component `c_ui` and the Optional component `vp_area` is removed

Table 1 continued

Operator	Example
23. Remove connection between Op comp and a Mandatory comp	The Optional connection between the Optional component `vp_area` and the component `c_course` is removed
24. Remove a connection between two Optional components	Given that our running example does not count with such a type of connection we added an Optional connection between the Optional component `vp_term` and the Optional component `vp_area`, and we considered this as the original SPL architecture for testing the operator. Then, after applying this operator, this connection is removed to obtain the mutant
25. Change the connection of a Mandatory receptacle with another Mandatory receptacle	The receptacles `Isubject` and `Icourse` of the component `c_ui` are connected to the facets `Icourse` and `Isubject`, respectively

The experiments we carried out involved different scenarios, whose details are presented below. In scenario 1, we tested the scalability of our approach. In Table 3, we show the number of architecture elements we used. We can see that the experiments in this scenario involve SPL architectures starting with 57 architecture elements up to 285 elements. The scalability behavior of the Ontology Factory is almost linear as shown in Fig. 8. Figure 9 shows that the performance degradation of the Verification Manager increases in a linear way demanding up to 10 s for 285 elements. However, the performance of the Debugger degrades exponentially as more elements are included in the SPL architecture demanding up to 463 s for 285 elements, as shown in Fig. 10.

Given that we found that the Debugger degrades rapidly we adopted an strategy that allowed our approach to handle larger architectures. This strategy involves making sub-partitions of the SPL architectures together with their associated product architectures into smaller sub-architectures. We generated such sub-architectures manually. We considered two constraints for maintaining the verification of partitioned architectures consistent. First, it was considered that two adjacent sub-architectures cross-cut each other to avoid loosing information, where the second sub-architecture includes as its first element the last element of the first sub-architecture. Such an element can be a variation point, a component, or a connector, in the case of an SPL architecture. In the case of the product architecture, this element can be a component or a connector. Second, component and connector connections cannot cross-cut two sub-architectures. In addition, the maximum size of the sub-architectures was set by determining the average rate of change of the performance degradation. We calculate the average rate of change by dividing the change in the execution time by the change in the number of architecture elements. We defined a maximum rate of change of around one in order to keep a linear performance degradation when the size of the SPL architecture scales up. A rate of change greater than one means that the execution time grows more rapidly than the number of architecture elements. We found, in the case of our testbed platform, that SPL sub-architectures of up to 216 were suitable. This because as a base line we had 62 elements which took 13.70 s. Then we got the average rate of change of the base line with respect to different amounts of elements. We obtained an average rate of change of 0.697 for 216 elements. The rate of change was larger than one for larger sub-architectures.

Therefore, in scenario 2, we employed the partitioning strategy to test the scalability of our approach with larger architectures. In Table 4 we show the number of architecture elements we used for this scenario, which ranges from 1026 elements to 5016 elements; and we considered a varying number of *requires/excludes* relationships, *Mandatory* elements, variation points, and variants ranging from 9 to 44, 45 to 220, 162 to 792, and 810 to 3960, respectively. We can see that the Ontology Factory, the Verification Manager, and even the Debugger exhibit a linear scalability behavior in scenario 2, as depicted in Figs. 11, 12, and 13, respectively. In the case of the Debugger experiments (Fig. 13), we reduced the number of repetitions to five given that in some cases each execution took around one hour. Based on the central limit theorem and given that in this experiment scenario the sampling distribution was nearly normal, we considered the sample size (i.e., five runs) to be large enough.

In scenario 3, we tested how the performance of the Debugger degraded for two *Alternative* variation points (VPs) interconnected and a *Mandatory* component connected to one of the VPs when the amount of variants increases. We performed this test given that the computational complexity of Rule 3 statements, which define the restrictions for *Alternative* connections, grows exponentially as the number of variants increases, specially when two *Alternative* VPs are interconnected. We focused only on the degradation of the Debugger since it suffers a much higher degradation than both the Ontology Factory and the Verifi-

Table 2 Results of mutation testing

Operator	Mutants	Selected Mutants	% Selected Mutants	Mutants Killed	Equivalent Mutants
1. Remove a Mandatory component	6	6	100	6	0
2. Change a Mandatory comp to an Optional comp	6	6	100	6	0
3. Change the name of a Mandatory component	6	6	100	6	0
4. Add an Optional component to an OR group	3	3	100	3	0
5. Add an Optional component to an Alternative group	1	1	100	1	0
6. Remove an Optional component from an OR group	3	3	100	0	3
7. Remove an Optional comp from an Altern group	2	2	100	0	2
8. Remove an Optional comp and its assoc variants	5	5	100	4	1
9. Change an Optional comp to a Mandatory comp	6	6	100	6	0
10. Change an Optional comp to a Mandatory comp	1	1	100	0	1
11. Change the name of an Optional component	5	5	100	0	5
12. Change the instantiation relationship of an Optional component	30	6	20	4	2
13. Remove an OR group relationship	1	1	100	1	0
14. Change an OR group relationship to an Alternative group relationship	1	1	100	1	0
15. Remove an Alternative group relationship	1	1	100	1	0
16. Change an Alternative group relationship to an OR group relationship	1	1	100	1	0
17. Add a connection between two Mandatory comp	24	5	20.83	5	0
18. Add a connection between a Mandatory comp and an Optional component	27	6	22.22	6	0
19. Add a connection between an Optional comp and a Mandatory component	25	5	20	5	0
20. Add a connection between two Optional comp	20	4	20	4	0
21. Remove a connection between two Mandatory comp	6	6	100	6	0
22. Remove a connection between a Mandatory comp and an Optional component	1	1	100	1	0
23. Remove a connection between an Optional comp and a Mandatory component	1	1	100	1	0
24. Remove a connection between two Optional comp	1	1	100	1	0
25. Change the connection of a Mandatory receptacle with another Mandatory receptacle	5	5	100	5	0
26. Change the connection direction between two Mandatory components	6	6	100	6	0
27. Change the connection direction between a Mandatory component and an Optional component	5	5	100	5	0
28. Change the connection direction between two Optional components	1	1	100	1	0
29. Change the name of a receptacle interface	8	8	100	8	0
30. Change the name of a facet interface	8	8	100	8	0
31. Add a requires relationship	27	6	22.22	6	0
32. Remove a requires relationship	1	1	100	1	0
33. Add an excludes relationship	27	6	22.22	6	0
34. Remove an excludes relationship	1	1	100	1	0
Total	272	130	–	116	14

Table 3 Number of architecture elements in Scenario 1

Requires/excludes relationships	Mandatory	OR VPs	Alternat. VPs	Variants	Total
0	3	6	3	45	57
1	5	12	6	90	114
1	8	18	9	135	171
2	10	24	12	180	228
2	13	30	15	225	285

Table 4 Size of SPL architectures in Scenario 2

Requires/excludes relationships	Mandatory	OR VPs	Alternat. VPs	Variants	Total
9	45	108	54	810	1026
17	88	210	105	1575	1995
26	133	318	159	2385	3021
35	178	426	213	3195	4047
44	220	528	264	3960	5016

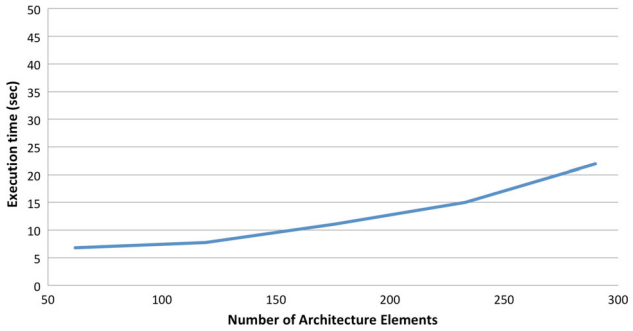


Fig. 8 Scenario 1, scalability of the Ontology Factory

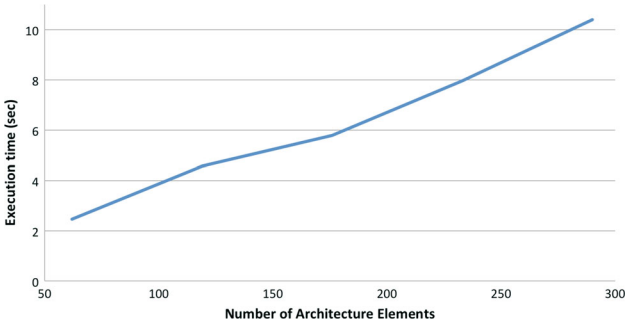


Fig. 9 Scenario 1, scalability of the Verification Manager

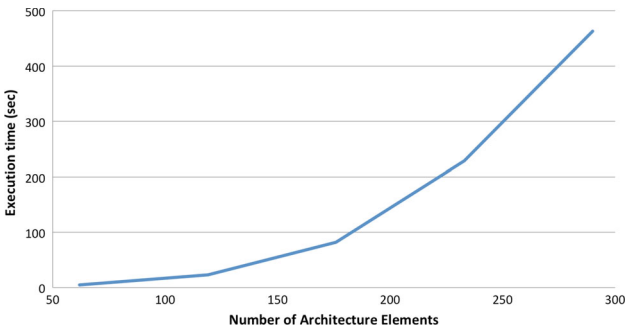


Fig. 10 Scenario 1, scalability of the Debugger

cation Manager, as shown in scenario 1. Table 5 depicts the performance of the Debugger degrades rapidly as the number of variants increases. In this case, we also reduced the number of repetitions to five given that in the case of 30 variants each execution took more than one hour. We obtained the average rate of change of the base line with respect to different amounts of variants. As a base line, we took the first row of the table. In addition, we tested the performance degradation of the Debugger for a *Mandatory* component connected to an *OR* variation point. Table 6 shows that in this case, a higher number of variants can be handled by the Debugger.

6.4 Discussion

The accuracy evaluation results have shown that our verifier has a high accuracy for detecting errors in product architectures (i.e., elements in a product architecture that are not consistent with an SPL architecture). We defined 34 mutant operators and evaluated 130 mutants (i.e., 130 modified SPLs) where 116 mutants were killed and 14 mutants were mutant equivalent. Our accuracy evaluation obtained a mutation score of 100%, which indicates that all errors in the test cases were correctly detected and the sources of errors were correctly identified by the Debugger. The verifier was unable to detect errors only in the cases the mutants were mutant equivalent. This is the expected behavior of the verifier since the products of a mutant equivalent are consistent with the original SPL. This was the case of removing a component from an *OR* group (operator 6), removing a component from an *Alternative* group (operator 7), or changing the name of an *Optional* component (operator 11), which only shrink the configuration space but do not generate test cases with errors.

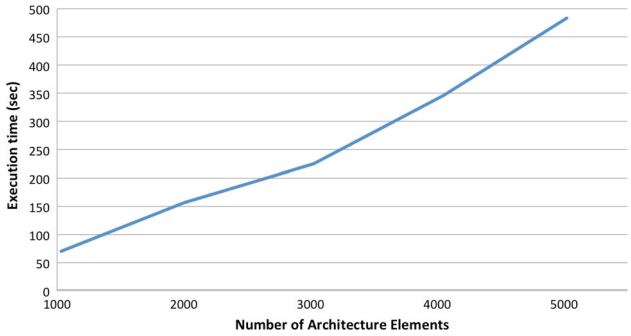


Fig. 11 Scenario 2, scalability of the Ontology Factory

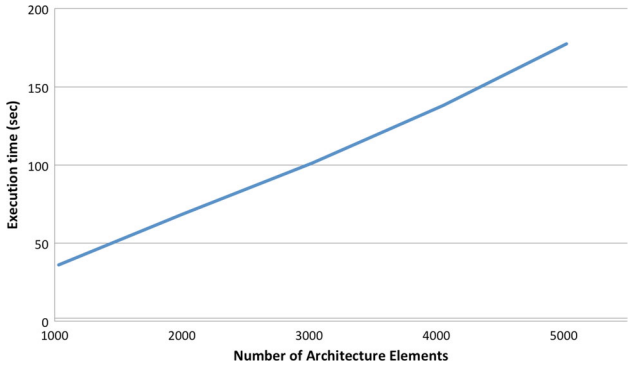


Fig. 12 Scenario 2, scalability of the Verification Manager

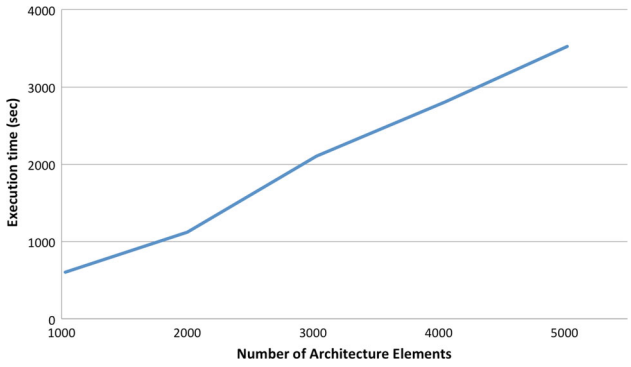


Fig. 13 Scenario 2, scalability of the Debugger

As an example, consider the mutant operator "7. Remove an Optional component from an Alternative group" that generates mutants that are mutant equivalent. In case we apply this operator and remove the variant `c_single_term` from the SPL, all products of the mutant will still be consistent with the original SPL. On the other hand, the selected variants of a product configuration that are not directly related to the change of a mutant do not affect the verification result. Therefore, we did not create all possible test cases, rather, we randomly selected a test case that is directly related to the change made to the mutant; this to make sure that this test case is a mutant killer. For instance, operator 2 regards changing a *Mandatory* component to an optional component such as converting the *Mandatory* component `c_student` to an optional component. In this case, we select a product that does not include `c_student` for it to be a mutant killer. In other cases, all products are mutant killers. For example operator 15 involves removing an *Alternative* group relationship. In this case, all products of the mutant are mutant killers. This is because in all products both connections `c_ui` to `c_single_term` and `c_ui` to `c_multiple_term` are missing. Nevertheless, the important point is finding a mutant killer and not every possible mutant killer. This is because one mutant killer is enough to test the same type of point of failure.

Table 5 Scenario 3a. Scalability of the Debugger for two Alternative VPs interconnected

Variants per Alternative VP	Execution Time (s)	Average Rate of Change
10	3.12	–
12	5.28	1.07
15	21.51	3.67
20	86.36	8.32
25	286.08	18.86
30	792.46	39.46

Table 6 Scenario 3b. Scalability of the Debugger for an OR VP connected to a Mandatory Component

Variants per OR VP	Execution time (s)	Average rate of change
100	1.92	–
200	5.51	0.03
300	10.42	0.04
400	16.78	0.04
500	26.32	0.06

Regarding the scalability evaluation, the first scenario is useful to find out the maximum number of architectural elements our approach is able to handle on a particular hardware platform without degrading by partitioning it into sub-architectures, where each sub-architecture does not surpass such a number. The second scenario shows that the partition approach is useful to keep the linear performance behavior. The third scenario helps us to find out the maximum number of variants an *Alternative* variation point is able to handle on a particular hardware platform. We can observe that in both scenario 1 and scenario 2 the Debugger takes much more time to execute than both the Ontology Factory and the Verification Manager. The experimental results in scenario 1 show that the Debugger does not scale well for SPL architectures that include more than 225 elements. The expressiveness obtained by description logics is achieved at the expense of higher computational complexity that is present in current ontology reasoning engines [46]. This issue

is exacerbated by the fact that the Debugger employs multiple queries and consistency checks in the debugging process. The Debugger employs a consistency check for each component connection pattern in the SPL architecture. We addressed this scalability issue by using a partitioning strategy, whereby we partition large SPL architectures and product architectures into small sub-architectures. We have shown that by using this partitioning strategy, the scalability of our framework is linear. For instance, in the case of an SPL architecture of 1026 elements, the Debugger takes 603 s, whereas in the larger case with an SPL of 5016 elements the Debugger takes 3525 s.

The third scenario shows that *Alternative* variation points face scalability problems. The restrictions of *Alternative* elements are defined according to Rule 3. The scalability problems are due to the fact that restrictions of *n Alternative* connections require *n* statements each one involving *n* terms to define an *Alternative* variation point restriction; hence, the number of terms needed is n^2. In the case of interconnecting an $Alternativevariationpointvp_1$ to a variation point vp_2, where vp_2 is either another *Alternative* variation point or an *OR* variation point whereby vp_1 and vp_2 have *n* and *m* variants, respectively, we have $n*m$ possible connections. Therefore, restrictions of *Alternative* connections require $n*m$ statements, where each statement involves *m* terms; hence, the total number of terms needed is $n*m^2$. In order to avoid scalability problems we suggest that those *Alternative* variation points that are connected to another variation point be partitioned into a sub-architecture when the number of variants of the *Alternative* variation point has an average rate of change considerably higher than 1, which in our platform happens from 15 variants (see Table 5). Then, we suggest limiting the maximum number of variants associated with an *Alternative* variation to the point where the Debugger exhibits a rate that is close to one (which in our platform happens to be around 12 variants as shown in Table 5). Although our approach is able to handle only a short number of variants of *Alternative* variation points, OntoPAV is able to mange a larger number of variants in the case of a *Mandatory* component connected to an *OR* variation point. Furthermore, our approach can still be useful for small- and some medium-scale SPL architectures involving a short number of variants per *Alternative* variation points. In addition, the validity of the verification rules is shown with the theorems presented in Sect. 4. Such theorems validate the verification rules carried out by our framework for *Mandatory*, *OR*, *Alternative*, and *Optional* connections as well as for *requires/excludes* relationships.

Finally, it should be noted that simply using propositional formulas is not enough to validate the component interconnections of a product architecture are consistent with the component interconnections of an SPL architecture. Component interconnections involve relationships among component interfaces. However, our ontology-based approach has more expressive power to define and validate such relationships.

6.5 Limitations and threats to validity

Our work has a number of limitations. The process of manually partition both an SPL architecture and a product architecture can be a complex and error-prone task. Automating such a process is not addressed by this paper and is regarded as future work. Our proposal is able to handle a large number of variants for *OR* variation points; however, this is not the case of *Alternative* variation points that can only support a short number of variants. Future work will look into improving the performance of OntoPAV for *Alternative* variation points involving a larger number of variants. Although the Debugger is able to find the origin of errors, it does not allow us to make fixes on the fly. Further work is required to have a fully automated debugging process. As we have only addressed the verification of component interconnections, further work is required to include support for quality of service (QoS) aspects. Although our framework supports *requires* and *excludes* constraints, it does not support *complex constraints* [47] involving arbitrary propositional formulas. Nevertheless, it is feasible to extend our framework to support complex constraints given that arbitrary propositional formulas can be easily handled by description logic (DL) [31], which is employed by our framework. This is because description logic is more expressive than propositional logic. Such an extension to our framework concerns future work. In addition, our proposal does not support constraints with numerical features that require arithmetical operations. Incorporating these kinds of constraints is not straightforward and, thus, requires further work.

Next, we discuss the validity threats that could affect the evaluation results. In the case of threats to internal validity, the results of the accuracy evaluation could be affected by a bias in the selection of the test cases assessed. We reduced this threat by using mutation testing, which allowed us to guide the generation of test cases to cover a wide range of points of failure. Regarding the results of the scalability evaluation, there are various factors that can have an impact on the execution time of our system prototype such as the load imposed by operating system processes and other processes running on the machine. We repeated 30 times each experiment in order to reduce this threat.[17]

Threats to external validity are related to the fact that the artifacts evaluated may not reflect real world SPL architectures. In the case of the results of the accuracy evaluation, we improved external validity by using a test case, i.e., an

[17] The repetitions were reduced to five times only when the Debugger degraded rapidly.

SPL architecture, that includes all the types of connections checked by our verifier, namely *Mandatory*, *OR*, *Alternative*, and *Optional* connections as well as *requires/excludes* relationships. This threat was also diminished by testing all the mutation operators we defined. For this purpose, it was necessary to extend the SPL architecture in order to test the mutation operators 24 and 28 that were not covered by the original SPL architecture (see description of operators 24 and 28 in Sect 6.2). Regarding the results of the scalability evaluation, both the SPL and product architectures were randomly generated in our experiments. This fact is a threat to external validity given that these architectures may not reflect real world SPL and product architectures. The factors that impact scalability are the number of *requires/excludes* relationships, *Mandatory* elements (i.e., components and connectors) as well as the number of variation points (i.e., optional components and optional connectors) and variants that are included in the SPL architecture. Hence, we reduced this threat by considering a varying number of *requires/excludes* relationships, *Mandatory* elements, variation points, and variants ranging from 9 to 44, 45 to 220, 162 to 792, and 810 to 3960, respectively (see Table 4). The total number of architecture elements that we considered in our experiments were 1026, 1995, 3021, 4047, and 5016 (see Table 4). We also diminished this threat by considering worst case scenarios. For example, both scenario 1 and scenario 2 consider SPL architectures, in which *Alternative* variation points were connected consecutively in line (one to another). This can cause that terms of Rule 3 grow exponentially, as previously discussed. Lastly, scenario 3 tested the scalability problems faced by *Alternative* variation points when associated with a larger number of variants.

7 Related work

Several works have been carried to address SPL reverse engineering, some of them focus on feature models such as [48]. However, the majority of them have focused at the source code level [49]. A more generic approach [12] presents a unified framework to help the adoption of SPLs from legacy systems. The authors' approach enables feature identification along with their constraints and the generation of new products. The proposed framework enables the possibility of adapting it to different artifact types and algorithms. Rubin et al. [13] propose a development framework that guides the process of reengineering an SPL architecture in terms of components and their connections. However, the SPL architecture extraction process has to be carried out manually. A few efforts have been done to reverse engineer UML models of software architecture product lines. For example, Assunção et al. [11] propose an approach to automatically extract UML class diagrams with features annotations. UML class diagrams are indeed useful for the design of SPL architectures. Nevertheless, ADL-based designs complement class diagrams with a higher level of abstraction making it easier to reason about the architecture and communicating the design choices among stakeholders. However, to the best of our knowledge, currently there are not approaches able to automatically extract ADL-based SPL architectures, hence making it necessary to do it manually.

A related problem to SPL extraction from legacy products involves the coevolution between SPLs and products where the SPL needs to be extracted or merged to make it consistent with changes introduced separately in the products. For instance, Schulze et al. [50, 51] propose a method to ensure consistency among artifacts whereby the initial version of a product, the modified version of a product within the SPL, and the modified version within the application domain are merged to a consistent version. This process can be useful for future efforts targeting software architecture artifacts.

Several efforts have been carried out to achieve verification of feature models [4, 14–18, 32, 52–60];these approaches commonly identify whether a feature model represents at least one product and whether the model contains any errors such as contradictory feature relationships. A number of works have used an ontology-based approach to verify feature models [14, 32, 53, 55]. In [14] a predicate-based ontology language is used for modeling and formalizing feature models, and consistency is checked with Prolog. Guo et al. [53] propose an ontology-based formalization of feature models. In this work, changes in the model can be verified for consistency by employing a dependency matrix. Other ontology-based efforts have used description logic to verify the consistency of feature models [32, 55]. In our work, we follow an ontology-based approach similar to Wang et al. [32]; however, we focus on verifying consistency of product architectures rather than feature models. Verifying consistency of SPL architectures has received some attention [19, 24]. Czarnecki et al. define an approach to verify the well-formedness of an SPL architecture (i.e., an annotated model) by using OCL constraints. The approach ensures that a well-formed SPL architecture will derive in well-formed product architectures (i.e., template instances) for any valid feature configuration. For this purpose, the user has to manually define OCL constraints. In this approach, the authors assume the product architecture can be easily derived by a Template Preprocessor by simply removing those elements whose presence conditions, in the SPL architecture, evaluate to false. Also, the solution of our previous work allows for automating the generation of the product architectures [6] by defining the SPL in and ADL and providing the feature tree selections. However, none of these works helps to ensure that a reverse engineered SPL architecture is consistent with a set of legacy product architectures. One challenging problem is verifying a product architecture maintains consistency when

it is derived from dependent product lines (aka. multi-product lines [61]); this given that an evolution of an artifact in one product line can introduce inconsistencies in the product architectures of the other product lines. In [24], multi-view models with variability are checked for consistency. Other work considers verifying the consistency of SPL implementations [20]. More specifically, this approach verifies the programs are type safe, i.e., absence of references to undefined elements such as classes and variables. Thüm et al. [62] and Kästner et al. [63] provide an approach to verify at the implementation level the composition of multiple product lines. Other efforts check the consistency between the SPL architecture and the feature model [21]. Janota et al. [64] and van der Storm et al. [65] present an approach that verifies the mappings between feature and component architecture models. Asadi et al. [16] propose an approach to detect inconsistencies between goal models and feature models. However, only a few approaches have addressed the issue of checking consistency between the product architecture and the SPL architecture, but mainly focusing on behavioral aspects [22, 23]. Therefore, approaches are still missing for verifying structural aspects of the commonality and variability of a manually extracted SPL architecture with respect to the product architectures from which it is derived.

8 Conclusion

In this paper, we have presented the OntoPAV framework, an approach to verify consistency of component interconnection aspects between a product architecture of a legacy system and a reverse engineered SPL architecture. We rely on the use of ontologies to perform such a verification. Importantly, the user does not need to have skills on ontologies to use our approach, rather, we make use of model-driven techniques to automate the populated ontology generation process. We illustrated our approach via a motivating scholar system example that shows the validity of our solution. Our evaluation results show that our verifier has a high accuracy in detecting consistency errors between product architectures and an SPL architecture. Lastly, we have shown that by employing a partitioning strategy our framework is able to achieve a linear performance scalability for small- and, in some cases, medium-scale approaches.

Future work regards automating the process of partitioning SPL and product architectures. We will also look into the issue of improving the performance of our approach when handling *Alternative* variation points. Future improvements of our work also include extending OntoPAV to support not only the verification of consistency at the architecture level, but also at finer granularity levels (e.g., a code block [66]). We believe the preliminary results of our accuracy evaluation are encouraging and it is regarded as a future work using a synthetic data set to strengthen the accuracy evaluation. Finally, we will explore using our approach to verify changes introduced at runtime to SPLs.

Acknowledgements This work has been partially supported by the Universidad de Guadalajara under the PROSNI program.

Appendix A

See Table 7.

Table 7 Full list of mutation operators

Operator	Example
1. Remove a Mandatory component	Remove the Mandatory component `c_student` from the SPL architecture
2. Change a Mandatory component to an Optional component	Convert the Mandatory component `c_student` to an Optional component in the SPL architecture
3. Change the name of a Mandatory component	The name of the Mandatory component `c_student` is changed to `c_studentV2`
4. Add an Optional component to an OR group	The Optional component `vp_area` is disconnected from `c_ui` and connected to the receptacle `IdegreeOption` of the component `c_degree` as another degree option
5. Add an Optional component to an Alternative group	The Optional component `vp_area` is removed and its associated variant `c_area` is added as an alternative of the Optional component `vp_term`
6. Remove an Optional component from an OR group	Remove both the Optional component `vp_thesis` and its associated variant `c_thesis` from the OR group in the SPL architecture
7. Remove an Optional component from an Alternative group	The variant `c_single_term` is removed from the Alternative group in the SPL architecture
8. Remove an Optional component and its associated variants	Remove both the variation point `vp_area` and its associated variant `c_area` from the SPL architecture
9. Change the name of a variant	Change the name of the variant `c_thesis` by `c_thesisV2`
10. Change an Optional component to a Mandatory component	The Optional component `vp_area` and its associated variant `c_area` are transformed to a Mandatory component and the requires/excludes relationships are eliminated
11. Change the name of an Optional component	The name of the variant `c_area` is changed to `c_areaV2`. Another example is changing the variant `c_thesis` to `c_thesisV2`
12. Change the instantiation relationship of an Optional comp	The variants `c_thesis` and `c_article` are interchanged so that now they instantiate the Optional components `vp_article` and `vp_thesis`, respectively
13. Remove an OR group relationship	The Optional components `vp_professionalPractice`, `vp_article`, and `vp_thesis` are removed along with their associated variants `c_professionalPractice`, `c_article`, and `c_thesis`
14. Change OR group relationship to Alternative group relationship	The OR group relationship of our running example is transformed to an Alternative group relationship
15. Remove an Alternative group relationship	The Optional component `vp_term` is removed along with its variants `c_single_term` and `c_multiple_term`
16. Change Alternative group relationship to OR group relation	The Alternative group relationship of our running example is transformed to an OR group relationship
17. Add a connection between two Mandatory components	A receptacle of the component `c_student` is connected to a facet of the component `c_degree`
18. Add a connection between a Mandatory component and an Optional component	A receptacle of the component `c_course` is connected to a facet of the Optional component `vp_area`
19. Add a connection between an Optional component and a Mandatory component	Connect a receptacle of the Optional component `vp_area` to the facet of the component `c_course`
20. Add a connection between two Optional components	Connect a receptacle of the Optional component `vp_area` to a facet of the Optional component `vp_term`
21. Remove a connection between two Mandatory components	The connection between the components `c_ui` and `c_course` is removed
22. Remove a connection between	The connection between the component `c_ui` and the Optional component `vp_area`

Table 7 continued

Operator	Example
a Mandatory comp and Op comp	is removed
23. Remove connection between Op comp and a Mandatory comp	The Optional connection between the Optional component `vp_area` and the component `c_course` is removed
24. Remove a connection between two Optional components	Given that our running example does not count with such a type of connection, we added an Optional connection between the Optional component `vp_term` and the Optional component `vp_area`, and we considered this as the original SPL architecture for testing the operator. Then, after applying this operator, this connection is removed to obtain the mutant.
25. Change the connection of a Mandatory receptacle with another Mandatory receptacle	The receptacles `Isubject` and `Icourse` of the component `c_ui` are connected to the facets `Icourse` and `Isubject`, respectively
26. Change the connection direction between two Mandatory comp	The connection between the components `c_ui` and `c_course` is changed so that a receptacle of the component `c_course` is now connected to a facet of the component `c_ui`
27. Change the connection direction between a Mandatory component and an Optional component	Change the connection between the component `c_ui` and the Optional component `vp_area` so that a receptacle of `vp_area` is connected to a facet of `c_ui`
28. Change the connection direction between two Optional components	Given that our running example does not count with such a type of connection, we added an Optional connection between the Optional component `vp_term` and the Optional component `vp_area` (see operator 24).
29. Change the name of a receptacle interface	The name of the receptacle `Icourse` is changed to `IcourseV2`
30. Change the name of a facet interface	The name of the facet `Isubject` is changed to `IsubjectV2`
31. Add a requires relationship	We add a requires relationship from variant `c_thesis` to variant `c_single_term`
32. Remove a requires relationship	The requires relationship from variant `c_single_term` to the variant `c_area` is removed
33. Add an excludes relationship	We add an excludes relationship from variant `c_thesis` to the variant `c_single_term`
34. Remove an excludes relationship	The excludes relationship from variant `c_multiple_term` to the variant `c_area` is removed

References

1. Krueger, C.W.: New methods in software product line practice. Commun. ACM **49**, 37–40 (2006)
2. Weiss, D.M., Clements, P.C., Kang, K., Krueger, C.: Software product line hall of fame. In: 10th International Software Product Line Conference (SPLC'06), 2006, pp. 237–237. https://doi.org/10.1109/SPLINE.2006.1691614
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
4. Batory, D.: Feature models, grammars, and propositional formulas. In: Proceedings of the 9th International Conference on Software Product Lines, SPLC'05, Springer-Verlag, Berlin, pp. 7–20 (2005)
5. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)
6. Duran-Limon, H.A., Garcia-Rios, C.A., Castillo-Barrera, F.E., Capilla, R.: An ontology-based product architecture derivation approach. IEEE Trans. Softw. Eng. **41**(12), 1153–1168 (2015)
7. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)
8. Li, Y., Schulze, S., Scherrebeck, H.H., Fogdal, T.S.: Automated extraction of domain knowledge in practice: The case of feature extraction from requirements at danfoss. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A, SPLC '20, Association for Computing Machinery, New York, NY (2020). https://doi.org/10.1145/3382025.3414968
9. Martinez, J., Wolfart, D., Assunção, W.K.G., Figueiredo, E.: Insights on software product line extraction processes: Argouml to argouml-spl revisited. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A, SPLC '20, Association for Computing Machinery, New York, NY (2020). https://doi.org/10.1145/3382025.3414971
10. Schlie, A., Knüppel, A., Seidl, C., Schaefer, I.: Incremental feature model synthesis for clone-and-own software systems in matlab/simulink, SPLC '20, Association for Computing Machinery, New York, NY. USA (2020). https://doi.org/10.1145/3382025.3414973
11. Assunção, W.K.G., Vergilio, S.R., Lopez-Herrejon, R.E.: Automatic extraction of product line architecture and feature models from uml class diagram variants. Information and Software Technology **117**, 106198 (2020). https://doi.org/10.1016/j.infsof.2019.106198
12. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Le Traon, Y.: Bottom-up adoption of software product lines: A generic and

extensible approach. In: Proceedings of the 19th International Conference on Software Product Line, SPLC '15, Association for Computing Machinery, New York, NY, pp. 101–110 (2015). https://doi.org/10.1145/2791060.2791086
13. Rubin, J., Czarnecki, K., Chechik, M.: Cloned product variants: From ad-hoc to managed software product lines. Int. J. Softw. Tools Technol. Transf. **17**(5), 627–646 (2015). https://doi.org/10.1007/s10009-014-0347-9
14. Bhushan, M., Goel, S., Kumar, A.: Improving quality of software product line by analysing inconsistencies in feature models using an ontological rule-based approach. Expert Syst. **35**(3), e12256 (2018). https://doi.org/10.1111/exsy.12256
15. Elfaki, A.O.: A rule-based approach to detect and prevent inconsistency in the domain-engineering process. Expert. Syst. **33**(1), 3–13 (2016). https://doi.org/10.1111/exsy.12116
16. Asadi, M., Gröner, G., Mohabbati, B., Gašević, D.: Goal-oriented modeling and verification of feature-oriented product lines. Softw. Syst. Model. **15**(1), 257–279 (2016). https://doi.org/10.1007/s10270-014-0402-8
17. Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., Saake, G.: Potential synergies of theorem proving and model checking for software product lines. In: Proceedings of the 18th International Software Product Line Conference - Volume 1, SPLC '14, ACM, New York, NY, pp. 177–186 (2014). https://doi.org/10.1145/2648511.2648530
18. Zhang, X., Møller-Pedersen, B.: Towards correct product derivation in model-driven product lines. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) System Analysis and Modeling: Theory and Practice, pp. 179–197. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013)
19. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (Eds.), GPCE, ACM, 2006, pp. 211–220
20. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE '07, ACM, New York, NY, pp. 95–104 (2007)
21. Satyananda, T.K., Lee, D., Kang, S.: A formal approach to verify mapping relation in a software product line. In: CIT, IEEE Computer Society, pp. 934–939 (2007)
22. Kishi, T., Noda, N.: Formal verification and software product lines. Commun. ACM **49**(12), 73–77 (2006)
23. Brito, P.H.S., Rubira, C.M.F., de Lemos, R.: Verifying architectural variabilities in software fault tolerance techniques. In: WICSA/ECSA, pp. 231–240 (2009)
24. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (Eds.), ECMFA, Vol. 6138 of Lecture Notes in Computer Science, Springer, pp. 217–232 (2010)
25. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner, Web Semantics: Science, Services and Agents on the World Wide Web vol 5, no. 2, pp. 51–53, (2007) software Engineering and the Semantic Web
26. Selic, B.: Model-driven development: Its essence and opportunities. In: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC '06, IEEE Computer Society, Washington, DC, pp. 313–319 (2006)
27. Duran-Limon, H.A., Velasco-Elizondo, P., Mora, M., Meda-Campana, M.E., Aguilar, K., Soto-Sumuano, L., Hernanddez-Ochoa, M.: Ontopav framework: Complementary documents (2022). https://github.com/hduran-limon/OntoPAV
28. Studer, R., Benjamins, V., Fensel, D.: Knowledge engineering: Principles and methods. Data Knowl. Eng. **25**(1), 161–197 (1998). https://doi.org/10.1016/S0169-023X(97)00056-6
29. Guarino, N., Oberle, D., Staab, S.: What Is an Ontology? Handbook on Ontologies, pp. 1–17. Springer, Berlin (2009)
30. Horridge, M., Drummond, N., Jupp, S., Moulton, G., Stevens, R.: A practical guide to building owl ontologies using the protege-owl plugin and co-ode tools edition 1.2, Tech. rep., Technical report, The University Of Manchester (2009)
31. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press, New York (2007)
32. Wang, H.H., Li, Y.F., Sun, J., Zhang, H., Pan, J.: Verifying feature models using owl. Web Semant. **5**, 117–129 (2007)
33. Foundation, T.E.: Acceleo, http://www.acceleo.org/. Accessed: 2019-06-06 (2019)
34. W3C, Turtle, http://www.w3.org/TeamSubmission/turtle/, accessed: 2021-06-06 (2019)
35. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six-mutation testing advances: An analysis and survey, Vol. 112 of Advances in Computers, Elsevier, pp. 275–378 (2019). https://doi.org/10.1016/bs.adcom.2018.03.015
36. Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. **32**(8), 608–624 (2006). https://doi.org/10.1109/TSE.2006.83
37. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M.A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, Association for Computing Machinery, New York, NY, pp. 302–313 (2013). https://doi.org/10.1145/2483760.2483769
38. Offutt, J.: A mutation carol: Past, present and future. Inf. Softw. Technol. 53(10), pp. 1098–1107 (2011) special Section on Mutation Testing. https://doi.org/10.1016/j.infsof.2011.03.007
39. Fraser, G., Arcuri, A.: Achieving scalable mutation-based generation of whole test suites. Empirical Softw. Eng. **20**(3), 783–812 (2015)
40. Papadakis, M., Malevris, N.: Automatic mutation test case generation via dynamic symbolic execution. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering, pp. 121–130 (2010). https://doi.org/10.1109/ISSRE.2010.38
41. Ferreira, J.M., Vergilio, S.R., Quináia, M.A.: Software product line testing based on feature model mutation. Int. J. Softw. Eng. Knowl. Eng. **27**, 817–840 (2017)
42. Devroey, X., Perrouin, G.. Papadakis, M., Legay, A., Schobbens, P.-Y., Heymans, P.: Featured model-based mutation analysis. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 655–666 (2016). https://doi.org/10.1145/2884781.2884821
43. Henard, C., Papadakis, M., Le Traon, Y.: Mutation-based generation of software product line test configurations. In: Le Goues, C., Yoo, S. (eds.) Search-Based Software Engineering, pp. 92–106. Springer International Publishing, Cham (2014)
44. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Assessing software product line testing via model-based mutation: An application to similarity testing. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 188–197 (2013). https://doi.org/10.1109/ICSTW.2013.30
45. Papadakis, M., Malevris, N.: An empirical evaluation of the first and second order mutation testing strategies. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, pp. 90–99 (2010). https://doi.org/10.1109/ICSTW.2010.50
46. Dentler, K., Cornet, R., ten Teije, A., de Keizer, N.: Comparison of reasoners for large ontologies in the owl 2 el profile. Semant. web **2**(2), 71–87 (2011). https://doi.org/10.3233/SW-2011-0034

47. Knüppel, A.: The role of complex constraints in feature modeling, Master's thesis, Institute of Software Engineering and Automotive Informatics, Technische Universität Carolo-Wilhelmina zu Braunschweig (2016)
48. Lopez-Herrejon, R.E., Linsbauer, L., Galindo, J.A., Parejo, J.A., Benavides, D., Segura, S., Egyed, A.: An assessment of search-based techniques for reverse engineering feature models. J. Syst. Softw. **103**, 353–369 (2015). https://doi.org/10.1016/j.jss.2014.10.037
49. Martinez, J., Assunção, W.K.G., Ziadi, T.: Espla: a catalog of extractive spl adoption case studies. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, SPLC '17, Association for Computing Machinery, New York, NY, pp. 38–41 (2017). https://doi.org/10.1145/3109729.3109748
50. Schulze, S., Schulze, M., Ryssel, U., Seidl, C.: Aligning coevolving artifacts between software product lines and products. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '16, Association for Computing Machinery, New York, NY, pp. 9–16 (2016). https://doi.org/10.1145/2866614.2866616
51. Kirchhof, J.C., Nieke, M., Schaefer, I., Schmalzing, D., Schulze, M.: Variant and Product Line Co-Evolution, pp. 333–351. Springer International Publishing, Cham (2021)
52. Yang, D., Dong, M.: Applying constraint satisfaction approach to solve product configuration problems with cardinality-based configuration rules. J. Intell. Manuf. **24**(1), 99–111 (2013). https://doi.org/10.1007/s10845-011-0544-2
53. Guo, J., Wang, Y., Trinidad, P., Benavides, D.: Consistency maintenance for evolving feature models. Expert Syst. Appl. **39**(5), 4987–4998 (2012). https://doi.org/10.1016/j.eswa.2011.10.014
54. Gheyi, R., Massoni, T., Borba, P.: Automatically checking feature model refactorings. J. Univ. Comput. Sci. **17**(5), 684–711 (2011)
55. Noorian, M., Ensan, A., Bagheri, E., Boley, H., Biletskiy, Y.: Feature model debugging based on description logic reasoning, pp. 158–164 (2011)
56. Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., Toro, M.: Automated error analysis for the agilization of feature modeling. J. Syst. Softw. **81**(6), 883–896 (2008)
57. Zhang, W., Zhao, H., Mei, H.: A Propositional Logic-Based Method for Verification of Feature Models, pp. 115–130. Springer, Berlin (2004)
58. Yan, H., Zhang, W., Zhao, H., Mei, H.: An Optimization Strategy to Feature Models' Verification by Eliminating Verification-Irrelevant Features and Constraints, pp. 65–75. Springer, Berlin (2009)
59. Mendonca, M., Wasowski, A., Czarnecki, K.: Sat-based analysis of feature models is easy. In: Proceedings of the 13th International Software Product Line Conference, SPLC '09, Carnegie Mellon University, Pittsburgh, PA, pp. 231–240 (2009)
60. Segura, S.: Automated analysis of feature models using atomic sets. In: Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops), pp. 201–207 (2008)
61. Holl, G., Grünbacher, P., Rabiser, R.: A systematic review and an expert survey on capabilities supporting multi product lines. Inf. Softw. Technol. **54**(8), 828–852 (2012). special Issue: Voice of the Editorial Board
62. Thüm, T., Winkelmann, T., Schröter, R., Hentschel, M., Krüger, S.: Variability hiding in contracts for dependent software product lines. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '16, ACM, New York, pp. 97–104 (2016)
63. Kästner, C., Ostermann, K., Erdweg, S.: A variability-aware module system. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, ACM, New York, pp. 773–792 (2012)
64. Janota, M., Botterweck, G.: Formal Approach to Integrating Feature and Architecture Models, pp. 31–45. Springer, Berlin (2008)
65. van der Storm, T.: Generic Feature-Based Software Composition, pp. 66–80. Springer, Berlin (2007)
66. Horcas, J.-M., Cortiñas, A., Fuentes, L., Luaces, M.R.: Combining multiple granularity variability in a software product line approach for web engineering. Inf. Softw. Technol. **148**, 106910 (2022). https://doi.org/10.1016/j.infsof.2022.106910

Hector A. Duran-Limon is currently a full Professor at the Information Systems Department, University of Guadalajara, Mexico. He completed a PhD at Lancaster University, England in 2002. Following this, he was a post-doctoral researcher until December 2003. He obtained an IBM Faculty award in 2008. His research interests include Cloud Computing and High Performance Computing (HPC). He is also interested in Software Architectures, Software Product Lines and Component-based Development. In 2006, He was invited to create a PhD program in Information Technologies for the University of Guadalajara, becoming a member of the Academic-Council. Contact him at the Information Systems Department, University of Guadalajara, Mexico; hduran@cucea.udg.mx.

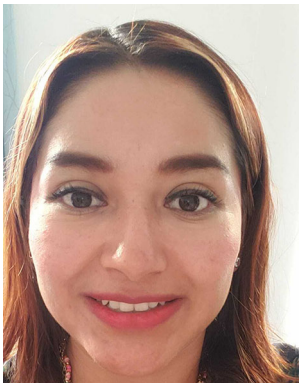
Perla Velasco-Elizondo is a researcher at the Autonomous University Zacatecas (Mexico). Her career includes positions such as Postdoctoral Researcher in the Architecture Based Languages and Environments group at the Institute for Software Research of the Carnegie Mellon University (USA), Associate Professor at the Centre for Mathematical Research (Mexico) and at the National Laboratory of Advanced Computer Science (Mexico). In all these institutions she has gained significant expertise in her main topics of interests: Software Architecture and Software Engineering. She teaches in under- and post-graduate programs in Software Engineering. She has also experience as a trainer for industry professionals; she has designed and executed customized training programs for industry professionals on topics such as: architectural requirements, design concepts, software architecture and software project management. Perla Velasco-Elizondo has worked with practicing software architects and software enginners helping them to

deploy project management, software architecture and software engineering practices and methods. She has some Software Architecture Certifications issued by Carnegie Mellon University - Software Engineering Institute. She also holds the Scrum Master certificate and the Kanban Team Practitioner certificate from the Scrum Alliance and the LeanKanban University, respectively. Perla Velasco-Elizondo co-authored one of the few books in Spanish on software architecture: Software Arquitectura: Conceptos y Ciclo de Desarrollo.

Manuel Mora has published over 100 research papers in international top conferences, research books, and refereed journals listed in JCRs and Scopus indexes. He has also co-edited five international research books on the topics of DMSS, IT Services and Data Centers, and Research Methods. He holds an M.Sc. in Artificial Intelligence (1989) from Monterrey Tech, and an Eng.D. in Engineering (2003) from the National Autonomous University of Mexico (UNAM), and currently is a full-time Professor at the Autonomous University of Aguascalientes (UAA), Mexico. His current research interests are agile development methodologies for: IT services, Big Data Analytics systems, ontology-based KMS, and SOA/MSA-based software systems. Prof. Mora is also an ACM Senior Member and a Mexican National Researcher at Level II.

Maria E. Meda-Campana received her Ph.D. in electronic engineering in 2002 at the Research Center and Advanced Studies of the National Polytechnic Institute, Mexico. Since 2003 she has worked at the University of Guadalajara as a full-time professor in the Information System Department. Her main research field deals with the modeling and applications of discrete event systems (DES) based on interpreted Petri nets (IPN). Current work deals with the analysis of the properties to characterize identifiable and diagnosable DES.

Karina Aguilar is the Postgraduate Coordinator at the Universidad Autónoma de Guadalajara. She is a professor and director of several postgraduate theses on software engineering topics. Dr. Aguilar has participated in several national and international forums with business and government organizations, maintains close relationships with the high-tech industry, and has received numerous teaching, research, service, and leadership awards.



Martha Hernandez-Ochoa is currently a full professor and researcher at the Knowledge Fundamentals Department, University of Guadalajara, since 2017. She received a B. S. degree in Computer Engineering from University of Guadalajara. Later, she received MSc. and Ph. D degree in Computer Science from CINVESTAV, IPN, Guadalajara. Her research interests include Wireless Network, Performance Models, Data communications and Networking, IoT and Data Acquisition Systems.

Leonardo Soto Sumuano PhD-Engineer in Computer Systems with Specialization in Telecommunications, Pierre et Marie Curie University, Paris VI (Paris, France 1986) Communications and Electronics Engineer from the Universidad Autonoma Metropolitana Unidad Iztapalapa (1980). He has projects as Engineer-Researcher at Telefónica Investigación y Desarrollo (TIDSA, Madrid) at the Centro Nacional de Estudios en Telecomunicaciones (CNET, Lannion France), and project leader at the Centro de Investigación y Desarrollo de Teléfonos de México (1986-1992). Member of the National System of Researchers of Mexico. Technology and Health research area. Environment and Environmental Pollution by non-ionizing electromagnetic radiation (RNI) and electronic waste.