The Journal of Systems and Software 139 (2018) 211–237

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

Change impact analysis for evolving configuration decisions in product line use case models

Ines Hajri[a], Arda Goknil[a,*], Lionel C. Briand[a], Thierry Stephany[b]

[a] SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
[b] International Electronics & Engineering (IEE), Contern, Luxembourg

ARTICLE INFO

Article history:
Received 19 June 2017
Revised 22 December 2017
Accepted 5 February 2018
Available online 15 February 2018

Keywords:
Change impact analysis
Product line engineering
Use case driven development
Use case configurator
Evolving decisions
Incremental reconfiguration

ABSTRACT

Product Line Engineering is becoming a key practice in many software development environments where complex systems are developed for multiple customers with varying needs. In many business contexts, use cases are the main artifacts for communicating requirements among stakeholders. In such contexts, Product Line (PL) use cases capture variable and common requirements while use case-driven configuration generates Product Specific (PS) use cases for each new customer in a product family. In this paper, we propose, apply, and assess a change impact analysis approach for evolving configuration decisions in PL use case models. Our approach includes: (1) automated support to identify the impact of decision changes on prior and subsequent decisions in PL use case diagrams and (2) automated incremental regeneration of PS use case models from PL use case models and evolving configuration decisions. Our tool support is integrated with IBM Doors. Our approach has been evaluated in an industrial case study, which provides evidence that it is practical and beneficial to analyze the impact of decision changes and to incrementally regenerate PS use case models in industrial settings.

1. Introduction

Product Line Engineering (PLE) is becoming crucial in many domains such as automotive and avionics where software systems are getting more complex and developed for multiple customers with varying needs. In such domains, many development contexts are use case-driven and this strongly influences their requirements engineering and system testing practices (Nebut et al., 2006a, 2006b; Wang et al., 2015a, 2015b).

For example, IEE S.A. (in the following "IEE") (IEE, International Electronics & Engineering), a leading supplier of embedded software and hardware systems in the automotive domain, follows a use case-driven development process to develop automotive sensing systems for multiple major car manufacturers worldwide. To develop a new product in a new project, IEE analysts elicit requirements as use case models from the initial customer. For each new customer of the product, IEE analysts clone the current models and identify differences to produce new use cases. With such practice, analysts loose track of commonalities and variabilities across products and they, together with the customer, need to evaluate the entire use cases. This practice is fully manual, error-prone and time-consuming, which leads to ad-hoc change management for requirements artifacts, e.g., use case diagrams and specifications, in the context of product lines. Therefore, product line modeling and configuration techniques are needed to automate the reuse of use case models in a product family.

The need for supporting PLE in the context of use case-driven development has already been acknowledged and many product line use case modeling and configuration approaches have been proposed in the literature (e.g., Eriksson et al., 2005; Eriksson et al., 2004; Fantechi et al., 2004a; Fantechi et al., 2004b; Czarnecki and Antkiewicz, 2005; Alférez et al., 2009). Most of the existing approaches rely on feature modeling, including establishing and maintaining traces between features and use case models (Sepulveda et al., 2016; Santos et al., 2015). The analysts should capture variability information as features, and establish traces between feature and use case models to model variability in use cases. For each new product in a product family, features should be selected to make configuration decisions and automatically generate use case models. In practice, many software development companies find such additional traceability and modeling effort to be impractical. In addition, requirements evolution results in changes

* Corresponding author.
E-mail addresses: ines.hajri@uni.lu (I. Hajri), arda.goknil@uni.lu, goknil@svv.lu (A. Goknil), lionel.briand@uni.lu (L.C. Briand), thierry.stephany@iee.lu (T. Stephany).

https://doi.org/10.1016/j.jss.2018.02.021

in configuration decisions and variability information, e.g., a selected variant use case being unselected for a product. It is critical for the analysts to identify in advance the impact of such evolution for better decision-making during the configuration process. For instance, impacted decisions, i.e., subsequent decisions to be made and prior decisions cancelled or contradicting when a decision changes, need to be identified to reconfigure the generated use case models.

To the best of our knowledge, there is no existing approach that explicitly supports automated change management of product line use cases for evolving configuration decisions. There are approaches (Thüm et al., 2009; Bürdek et al., 2015; Pleuss et al., 2012; Heider et al., 2012b; Paskevicius et al., 2012) that study the evolution of feature models in terms of identifying the impact of feature changes on other features, but they do not address the change impact on configuration decisions or on generated use cases.

In addition, existing configurators (e.g., Eriksson et al., 2005; Fantechi et al., 2004b; Czarnecki and Antkiewicz, 2005) do not support incremental reconfiguration of use case models, a capability that is essential in practice. For a variety of reasons, analysts manually assign traces from the configured use case models to other software and hardware specifications as well as to the customers' requirements documents for external systems (Ramesh and Jarke, 2001). Evolving configuration decisions result in the reconfiguration of Product Specific (PS) use case models. When the use case models are reconfigured for all decisions, including unimpacted decisions, manually assigned traces are lost. The analysts need to reassign all the traces after each reconfiguration. It is therefore vital to enable the incremental reconfiguration of use case models focusing only on changed decisions and their side-effects. With such support, the analysts could then reassign traces only for the parts of the reconfigured models impacted by decision changes.

In our previous work (Hajri et al., 2015), we proposed and assessed the Product Line Use case modeling Method (PUM) to support variability modeling in Product Line (PL) use case diagrams and specifications, intentionally avoiding any reliance on feature models and thus avoiding unnecessary modeling and traceability overhead. PUM adopts the existing PL extensions of use case diagrams in the work of Halmans and Pohl (Halmans and Pohl, 2003). In order to model variability in use case specifications, we introduced new product line extensions for the Restricted Use Case Modeling method (RUCM) (Yue et al., 2013). We developed a use case-driven configuration approach (Hajri et al., 2016a, 2016b) based on PUM. Our configuration approach supports guiding stakeholders in making configuration decisions (e.g., checking consistency of a decision with prior decisions) and automatically generating PS use case models from the PL models and configuration decisions. It is supported by a tool, *PUMConf (Product line Use case Model Configurator)* (Hajri et al., 2016b).

In this paper, we propose, apply and assess a change impact analysis approach, based on our use case-driven modeling and configuration techniques, to support the evolution of configuration decisions. We do not address here evolving PL use case models, which need to be treated in a separate approach. Change impact analysis provides a sound basis to decide whether a change is adequate, and to identify which decisions should be changed as a consequence (Passos et al., 2013). In our context, we aim to automate the identification of decisions impacted by changes in configuration decisions on PL use case models. Our approach supports three activities. First, the analyst proposes a change but does not apply it to the corresponding configuration decision. Second, the impact of the proposed change on other configuration decisions for the PL use case diagram are automatically identified. In the PL use case diagram, variant use cases and variation points are connected to each other with some dependencies, i.e., *require, conflict* and *include*. In the case of a changed diagram decision contradicting prior and/or subsequent diagram decisions, such as a subsequent decision resulting in selecting variant use cases violating some dependency constraints because of the new/changed decision, we automatically detect and report them. To this end, we improved our consistency checking algorithm (Hajri et al., 2016a), which enables reasoning on subsequent decisions as part of our impact analysis approach. The analyst is informed about the change impact on decisions for the PL use case diagram. One crucial and innovative aspect is that our approach identifies not only the impacted decisions but also the cause of the impact, e.g., violation of dependency constraints, changing decision restrictions, and contradicting decision restrictions. In practice, the reason of the impact is important to help the analyst identify what further changes to make on impacted decisions. Using the output of our impact analysis, the analyst should decide whether the proposed change is to be applied to the corresponding decision. Third, the PS use case models are incrementally regenerated only for the impacted decisions after the analyst makes all the required changes. To do so, we implemented a model differencing pipeline which identifies decision changes to be used in the reconfiguration of PS models. There are two sets of decisions: (i) the set of previously made decisions used to initially generate the PS use case models and (ii) the set of decisions including decisions changed after the initial generation of the PS models. Our approach compares the two sets to determine for which decisions we need to incrementally regenerate the PS models. To support these activities, we extended PUMConf.

This paper is an extension of our work published in REFSQ 2017 (Hajri et al., 2017b). The published work reported on the incremental reconfiguration of PS use case models. In the current paper, we introduce the automated impact analysis of decision changes on other decisions and we provide the details of the proposed tool support, which is made publicly available. We also improve the evaluation of our entire approach with a questionnaire study and some structured interviews with experienced engineers at IEE. To summarize, the contributions of this paper are:

- A change impact analysis approach that informs the analysts about the causes of change impacts on configuration decisions in order to improve the decision-making process and to incrementally reconfigure the generated PS use case models for the impacted decisions only;
- A publicly available tool integrated as a plug-in in IBM DOORS, which automatically identifies the impact of configuration decision changes and incrementally regenerates the PS use case models;
- An industrial case study demonstrating the applicability and benefits of our change impact analysis approach.

This paper is structured as follows. Section 2 provides the background on PUM and PUMConf on which this paper builds the proposed change impact analysis approach. Section 3 introduces the industrial context of our case study to illustrate the practical motivations for our approach. In Section 4, we provide an overview of the approach. Sections 5 and 6 provide the details of its core technical parts. In Section 7, we present our tool while Section 8 reports on an industrial case study, involving an embedded system called Smart Trunk Opener (STO). Section 9 discusses the related work. In Section 10, we conclude the paper.

2. Background

In this section we present the background regarding the elicitation of PL use case models (see Section 2.1), and our configuration approach (see Section 2.2).

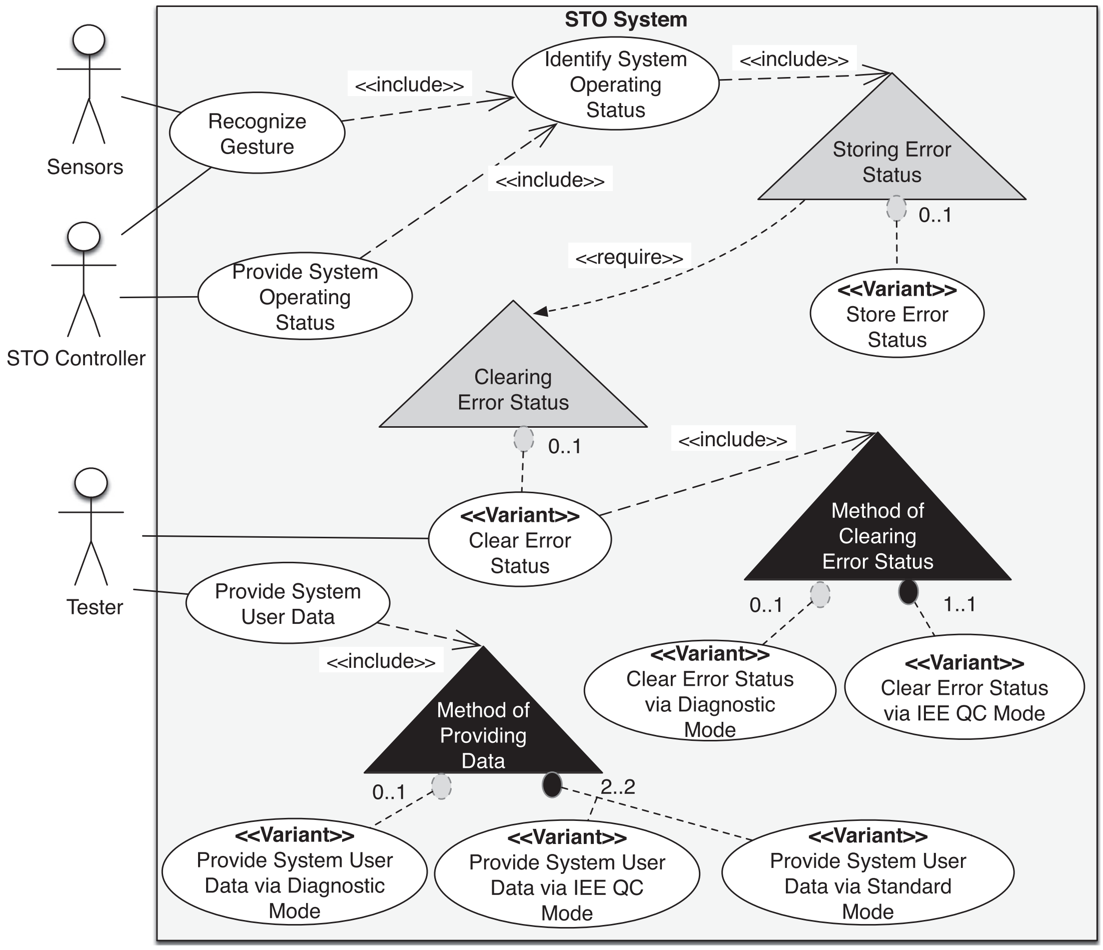

STO System
Sensors
STO Controller
Tester
Recognize Gesture
Identify System Operating Status
<<include>>
<<include>>
<<include>>
Storing Error Status
0..1
<<Variant>> Store Error Status
<<require>>
Provide System Operating Status
Clearing Error Status
0..1
<<include>>
<<Variant>> Clear Error Status
Method of Clearing Error Status
0..1
1..1
<<Variant>> Clear Error Status via Diagnostic Mode
<<Variant>> Clear Error Status via IEE QC Mode
Provide System User Data
<<include>>
Method of Providing Data
0..1
2..2
<<Variant>> Provide System User Data via Diagnostic Mode
<<Variant>> Provide System User Data via IEE QC Mode
<<Variant>> Provide System User Data via Standard Mode

Fig. 1. Part of the Product Line use case diagram for STO.

In the rest of the paper, we use Smart Trunk Opener (STO) as a case study, to motivate, illustrate and assess our approach. STO is a real-time automotive embedded system developed by IEE. It provides automatic, hands-free access to a vehicle's trunk, in combination with a keyless entry system. In possession of the vehicle's electronic remote control, the user moves her leg in a forward and backward direction at the vehicle's rear bumper. STO recognizes the movement and transmits a signal to the keyless entry system, which confirms that the user has the remote. This allows the trunk controller to open the trunk automatically.

2.1. Elicitation of variability in PL use cases

Elicitation of PL use case models is based on the Product line Use case modeling Method (PUM) (Hajri et al., 2015). In this section, we give a brief description of the PUM artifacts.

2.1.1. Use case diagram with PL extensions

For use case diagrams, we employ the PL extensions proposed by Halmans and Pohl (2003) and Buhne et al. (2003) since they support explicit representation of variants, variation points, and their dependencies (see Fig. 1). We do not introduce any further extensions.

A use case is either *Essential* or *Variant*. Variant use cases are distinguished from essential (mandatory) use cases, i.e., mandatory for all the products in a product family, by using the 'Variant' stereotype. A variation point given as a triangle is associated to one, or more than one use case using the 'include' relation. The mandatory variation points indicate where the customer has to make a selection for a product (the black triangles in Fig. 1). A 'tree-like' relation, containing a cardinality constraint, is used to express relations between variants and variation points, which are called *variability relations*. The relation uses a [min.max] notation in which *min* and *max* define the minimum and maximum numbers of variants that can be selected for the variation point. A variability relation is optional where $(min = 0)$ or $(min > 0$ and $max < n)$; n is the number of variants in a variation point. A variability relation is mandatory where $(min = \max = n)$. Optional and mandatory relations are depicted with light-grey and black filled circles, respectively (see Fig. 1). For instance, the 'Provide System User Data' essential use case has to support multiple methods of providing data where the methods of providing data via IEE QC mode and Standard mode are mandatory. In addition, the customer can select the method of providing data via diagnostic mode. In STO, the customer may decide the system does not store the errors determined while the operating status is being identified (see the 'Storing Error Status' optional variation point in Fig. 1). The extensions support the dependencies *require* and *conflict* among variation points and variant use cases (Buhne et al., 2003). With *require*, the selection of the variant use case in 'Storing Error Status' implies the selection of the variant use case in 'Clearing Error Status'.

Some further variability information is given in PL use case specifications. For instance, only PL use case specifications indicate in which flows of events a variation point is included.

2.1.2. Restricted Use Case Modeling (RUCM) with PL extensions

This section introduces the RUCM template and its PL extensions which we proposed. RUCM provides restriction rules and

Table 1
Some STO use cases in the extended RUCM.

1	**USE CASE** Recognize Gesture
2	**1.1 Basic Flow**
3	1. INCLUDE USE CASE Identify System Operating Status.
4	2. The system VALIDATES THAT the operating status is valid.
5	3. The system REQUESTS the move capacitance FROM the sensors.
6	4. The system VALIDATES THAT the movement is a valid kick.
7	5. The system SENDS the valid kick status TO the STO Controller.
8	**1.2 <OPTIONAL>Bounded Alternative Flow**
9	RFS 1–4
10	1. IF voltage fluctuation is detected THEN
11	2. RESUME STEP 1.
12	3. ENDIF
13	**1.3 Specific Alternative Flow**
14	RFS 2
15	1. ABORT.
16	**1.4 Specific Alternative Flow**
17	RFS 4
18	1. The system increments the OveruseCounter by the increment step.
19	2. ABORT.
20	
21	**USE CASE** Identify System Operating Status
22	**1.1 Basic Flow**
23	1. The system VALIDATES THAT the watchdog reset is valid.
24	2. The system VALIDATES THAT the RAM is valid.
25	3. The system VALIDATES THAT the sensors are valid.
26	4. The system VALIDATES THAT there is no error detected.
27	**1.4 Specific Alternative Flow**
28	RFS 4
29	1. INCLUDE <VARIATION POINT: Storing Error Status>.
30	2. ABORT.
31	
32	**USE CASE** Provide System User Data
33	**1.1 Basic Flow**
34	1. The tester SENDS the system user data request TO the system.
35	2. INCLUDE <VARIATION POINT : Method of Providing Data>.
36	
37	**<VARIANT>USE CASE** Provide System User Data via Standard Mode
38	**1.1 Basic Flow**
39	V1. <OPTIONAL>The system SENDS calibration TO the tester.
40	V2. <OPTIONAL>The system SENDS sensor data TO the tester.
41	V3. <OPTIONAL>The system SENDS trace data TO the tester.
42	V4. <OPTIONAL>The system SENDS error data TO the tester.
43	V5. <OPTIONAL>The system SENDS error trace data TO the tester.

keywords constraining the use of natural language (Yue et al., 2013). Since RUCM was not designed for PL modeling, we introduced some PL extensions (see Table 1). In RUCM, use cases have basic and alternative flows (Lines 2, 8, 13, 16, 22, 27, 33 and 38). In Table 1, we omit some alternative flows and basic information such as actors and pre/post conditions.

A basic flow describes a main successful path that satisfies stakeholder interests (Lines 3–7, 23–26 and 39–43). It contains use case steps and a postcondition. A step can be a system-actor interaction: an actor sends a request or data to the system (Lines 34); the system replies to an actor with a result (Line 7). In addition, the system validates a request or data (Line 4), or it alters its internal state (Line 18). The use case inclusion is given in a step with the keyword '*INCLUDE USE CASE*' (Line 3). The keywords are in capital letters. '*VALIDATES THAT*' (Line 4) indicates a condition that must be true to take the next step, otherwise an alternative flow is taken.

An alternative flow describes other scenarios, both success and failure. It always depends on a condition in a specific step of the basic flow. RUCM has *specific, bounded* and *global* alternative flows. A specific alternative flow refers to a step in the basic flow (Lines 13, 16, and 27). A bounded alternative flow refers to more than one step in the basic flow (Line 8), while a global one refers to any step in the basic flow. '*RFS*' is used to refer to reference flow steps (Lines 9, 14, 17, and 28). Bounded and global alternative flows begin with '*IF .. THEN*' for the conditions under which they are taken (Line 10). Specific alternative flows do not necessarily begin with '*IF .. THEN*' since a guard condition is already indicated in their reference flow steps (Line 4).

Our extensions are (i) new keywords for modeling interactions in embedded systems and (ii) new keywords for modeling variability. The keywords '*SENDS .. TO*' and '*REQUESTS .. FROM*' are to distinguish system-actor interactions (Lines 5, 7, 34, and 39–43). We introduce the notion of variation point and variant use case, complementary to the extensions in Section 2.1.1, into RUCM. Variation points can be included in basic or alternative flows with the keyword '*INCLUDE <VARIATION POINT : ... >*' (Lines 29 and 35). Variant use cases are given with the keyword '*<VARIANT >*' (Line 37).

Some variability cannot be captured in PL diagrams due to the required level of granularity for product configuration. To model such variability, as part of our extensions, we introduce optional steps, optional alternative flows and a variant order of steps. Optional steps and alternative flows begin with '*<OPTIONAL>*' (Lines 8 and 39–43). 'V' is used before any step number to express variant step order (Lines 39–43).

2.1.3. Discussion

Considerable research has already been devoted to documenting variability in use cases. Many approaches propose using variability models, e.g., feature models, that are traced to use case specifications and diagrams (Alferez et al., 2008; Eriksson et al., 2005, 2009; Buhne et al., 2006; Braganca and Machado, 2007; Griss et al., 1998). With the PL extensions, our method enables analysts to document variability directly in use case diagrams and specifications. This is a departure from the most common approach of having separate variability and use case models together with their trace links.

Our decision was motivated by our discussions with IEE analysts and engineers. In the current practice at IEE, like in many other environments, there is no systematic way to model variability information in use case diagrams and specifications. The IEE analysts write brief and informal notes attached to use case models to indicate what may vary in the use cases. IEE is reluctant to use feature models traced to use case models because of two main issues: (i) having feature models requires considerable additional modeling artifacts of a very different nature and additional tools, with manual assignment of traces at a very low level of granularity, e.g., sequences of use case steps; and (ii) they find it difficult to switch from feature models to use cases and vice versa during the decision-making process. By documenting variability directly in use case models, the analysts could focus on one artifact at a time to make configuration decisions. In our meetings at IEE, the analysts stated that the effort required to apply our PL extensions for modeling variability information was reasonable (Hajri et al., 2016a). They considered the extensions to be simple enough to enable communication between analysts and customers, but they also mentioned that training customers is necessary. Thus, the costs and benefits of the approach should be made clear to customers.

The separation of variability and development models was originally motivated by the need to provide representations targeting different stakeholders with distinct expertise and interests (Pohl et al., 2005). However, based on our observations in practice, it is often not the case that people who need to read variability models don't need to read development models, or vice-versa. These two groups are not mutually exclusive. Results in the case study and questionnaire study from our previous work (Hajri et al., 2015, 2016a) suggest that the PL use case model extensions we proposed were easy to read and used by automotive system engineers.

We do not claim that our modeling method is generally superior to feature modeling, or that feature modeling should be discarded. We only provide an alternative way to model variability

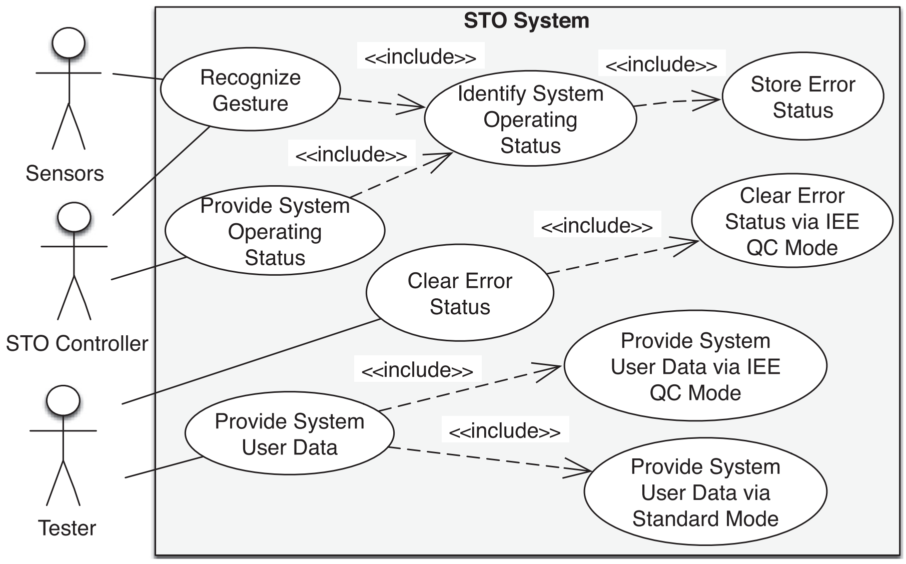

STO System
Sensors
STO Controller
Tester
Recognize Gesture
Identify System Operating Status
Store Error Status
Provide System Operating Status
Clear Error Status via IEE QC Mode
Clear Error Status
Provide System User Data via IEE QC Mode
Provide System User Data
Provide System User Data via Standard Mode
<<include>>

Fig. 2. Generated product specific use case diagram.

in the context of use case-driven development, which may be a preferred solution in certain contexts. Companies who already adopted feature modeling in their practice most probably have a different perception.

2.2. Configuration of PS use case models

PUMConf relies on variability information given in the PL use case models. The user selects (1) variant use cases in the PL diagram and (2) optional use case elements in the PL specifications, to generate the PS use case models.

The user makes decisions for the variation points in Fig. 1. A decision is about selecting, for the product, variant use cases in the variation point. The user selects *Store Error Status* and *Clear Error Status* in the variation points *Storing Error Status* and *Clearing Error Status*, respectively. She unselects *Clear Error Status via Diagnostic Mode* in the variation point *Method of Clearing Error Status*, while *Clear Error Status via IEE QC Mode* is automatically selected because of the mandatory variability relation. The user unselects *Provide System User Data via Diagnostic Mode* in the variation point *Method of Providing Data*. The PS diagram is automatically generated from the PL diagram and the diagram decisions (see Fig. 2 generated from Fig. 1).

The decision-making is an iterative process. We devised an algorithm to check the consistency of a decision with prior decisions (Hajri et al., 2016a). In the case of contradicting configuration decisions, such as two decisions resulting in selecting variant use cases violating some dependency constraints, the algorithm automatically detects and reports them. The user must then backtrack and revise the decisions to resolve the contradictions. Assume that the user first makes a decision in *Clearing Error Status*, which is unselecting *Clear Error Status*. No contradiction is identified since there is no prior decision. The user proceeds with *Storing Error Status* and selects *Store Error Status*. Our algorithm identifies a contradiction with the decision in *Clearing Error Status* since the selection of *Store Error Status* implies the selection of *Clear Error Status* via the *requires* dependency (see Fig. 1). The user is asked to resolve the contradiction by updating one of the decisions for *Storing Error Status* and *Clearing Error Status*. The user selects *Clear Error Status* to resolve the contradiction.

Next, the user makes decisions for the PL specifications. In Table 1, there are two variation points (Lines 29 and 35), one variant use case (Lines 37–43), five optional steps (Lines 39–43), one optional alternative flow (Lines 8–12), and one variant order group (Lines 39–43). The decisions for the variation points are already made in the PL diagram. The user selects only three optional steps with the order *V3, V1*, and *V5*. The optional alternative flow is unselected.

The PS use case specifications are automatically generated from the PL specifications, the diagram decisions and the specification decisions (see Table 2 generated from Table 1). For instance, based on the diagram decision for *Method of Providing Data* in Fig. 1, PUMConf creates two include statements for *Provide System User Data via Standard Mode* and *via IEE QC Mode* (Lines 31 and 34 in Table 2), a validation step (Line 30), and a specific alternative flow where *Provide System User Data via IEE QC Mode* is included (Lines 32–35). The validation step checks if the precondition of *Provide System User Data via Standard Mode* holds. If it holds, *Provide System User Data via Standard Mode* is executed in the basic flow (Line 31). If not, *Provide System User Data via IEE QC Mode* is executed in the alternative flow (Lines 32–35). The selected optional steps are generated with the decided order in the PS specifications (Lines 39–41).

3. Motivation and context

Our change impact analysis approach is developed as an extension of our configurator, PUMConf, in the context of software systems configured for multiple customers with varying needs, and developed according to a use case-driven process. In such a context, configuration decisions frequently change due to technological developments and evolving business needs. A change impact analysis approach is therefore needed for identifying other impacted decisions for the reconfiguration of PS models.

Changes on configuration decisions may have impact on other decisions in various ways. For instance, in the PL diagram in Fig. 1, the analyst changes the decision for the variation point *Clear Error Status* in order to resolve the contradiction with the prior decision for the variation point *Store Error Status* (see Section 2.2). This is done by selecting the variant use case *Clear Error Status*, which was previously unselected. This change has the following consequences: (i) the variation point *Method of Clearing Error Status* should now be considered in subsequent decisions; (ii) the variant use case *Clear Error Status via IEE QC Mode* is automatically selected because of the mandatory variability relation; (iii) the newly selected use cases should be added to the PS use case diagram while

Table 2
Some of the generated product specific specifications.

1	**USE CASE** Recognize Gesture
2	**1.1 Basic Flow**
3	1. INCLUDE USE CASE Identify System Operating Status.
4	2. The system VALIDATES THAT the operating status is valid.
5	3. The system REQUESTS the move capacitance FROM the sensors.
6	4. The system VALIDATES THAT the movement is a valid kick.
7	5. The system SENDS the valid kick status TO the STO Controller.
8	**1.2 Specific Alternative Flow**
9	RFS 2
10	1. ABORT.
11	**1.3 Specific Alternative Flow**
12	RFS 4
13	1. The system increments the OveruseCounter by the increment step.
14	2. ABORT.
15	
16	**USE CASE** Identify System Operating Status
17	**1.1 Basic Flow**
18	1. The system VALIDATES THAT the watchdog reset is valid.
19	2. The system VALIDATES THAT the RAM is valid.
20	3. The system VALIDATES THAT the sensors are valid.
21	4. The system VALIDATES THAT there is no error detected.
22	**1.4 Specific Alternative Flow**
23	RFS 4
24	1. INCLUDE USE CASE Store Error Status.
25	2. ABORT.
26	
27	**USE CASE** Provide System User Data
28	**1.1 Basic Flow**
29	1. The tester SENDS the system user data request TO the system.
30	2. The system VALIDATES THAT 'Precondition of Provide System User Data via Standard Mode'.
31	3. INCLUDE USE CASE Provide System User Data via Standard Mode.
32	**1.2 Specific Alternative Flow**
33	RFS 2
34	1. INCLUDE USE CASE Provide System User Data via IEE QC Mode.
35	2. ABORT.
36	
37	**USE CASE** Provide System User Data via Standard Mode
38	**1.1 Basic Flow**
39	1. The system SENDS the trace data TO the tester.
40	2. The system SENDS the calibration data TO the tester.
41	3. The system SENDS the error trace data TO the tester.

the corresponding use case specifications should be added to the PS specifications; and (iv) new optional steps and alternative flows are introduced for consideration if there is any in the added specifications. In some cases, subsequent decisions are also impacted because of decision restrictions. Assume that the variant use case *Store Error Status* in the variation point *Storing Error Status* is unselected, and no decision has been made yet for the variation point *Clearing Error Status*. When the analyst changes the decision by selecting *Store Error Status*, the subsequent decision for the variation point *Clearing Error Status* is restricted because *Clear Error Status* should be selected in the subsequent decision to avoid further decision contradictions.

In practice, from a more general standpoint, the analysts should be aware of the impacts of decision changes to possibly reconsider some of them. After changing a decision, impact analysis support is needed to guide subsequent decisions or to change prior decisions. Within our context, we identify two challenges that need to be considered in identifying the impact of decision changes and supporting the reconfiguration of PS use case models:

***Challenge 1*: Identifying the cause of the impact of changing decisions for PL use case diagrams.** Changes to configuration decisions driven by the PL use case diagram have an impact on prior decisions as well as on subsequent decisions to be made. Change impacts can have a variety of causes, which the analyst needs to take into account to decide whether the proposed change is adequate and to identify what further changes are needed on impacted decisions. For instance, a prior decision might be impacted because of the violation of some dependency constraints (i.e., *requires* and *conflicts*). A subsequent decision for a variation point might be impacted because it has a new restriction to satisfy the cardinality constraint of the variation point. Therefore, impact analysis should provide not only impacted decisions but also detailed information about their causes.

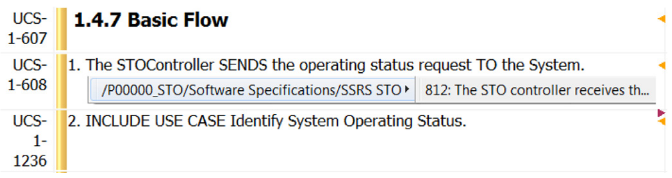

UCS-1-607 **1.4.7 Basic Flow**
UCS-1-608 1. The STOController SENDS the operating status request TO the System.
/P00000_STO/Software Specifications/SSRS STO ▸ 812: The STO controller receives th...
UCS-1-1236 2. INCLUDE USE CASE Identify System Operating Status.

Fig. 3. Part of an STO use case specification with trace links.

***Challenge 2*: Incremental regeneration of PS use case models.** In practice, for a variety of reasons, the analysts manually assign traces from the PS use case models to other software and hardware specifications as well as to the customers' requirements documents for external systems (Ramesh and Jarke, 2001). For instance, in order to verify the interaction between the system and the external systems, IEE's customers require that traces be assigned from the PS use case specifications to the related, external system requirements. Fig. 3 gives part of the basic flow of a PS use case specification in IBM DOORS with a trace to a customer's requirements specification.

Let us consider the trace in Fig. 3, which is from the first step of the basic flow to an external system requirement in the customer's software requirements specification. This use case step describes the operating status request sent by the STO controller, i.e., an external system implemented by the customer, while the traced external system requirement describes the condition in which the STO controller sends this request to the system. When the PS use case models are reconfigured for all the decisions, including unimpacted decisions, manually assigned traces such as the one in Fig. 3 are lost. The analysts need to reassign all the traces after each reconfiguration. It is therefore vital to enable the incremental regeneration of PS models by focusing only on impacted decisions. As a result, the analysts would reassign traces only for the parts of the PS use case models impacted by decision changes.

In the remainder of this paper, we focus on how to best address these challenges in a practical manner, in the context of use case-driven development, while relying on PUM for modeling PL use case models, and on PUMConf for the configuration of PS use case models.

4. Overview of the approach

The process in Fig. 4 presents an overview of our approach. In Step 1, *Propose a change for a decision*, the analyst is asked to propose a change for a configuration decision made previously for the PL use case diagram.

The configuration decision change proposed by the analyst is not actually applied to the corresponding decision yet. In Step 2, *Identify the change impact on other decisions*, our approach automatically identifies the impact of the proposed change on other configuration decisions for the PL use case diagram. The analyst is informed about the impact of the decision change on prior and subsequent decisions, e.g., contradicting decisions and restricted subsequent decisions (*Challenge 1*).

The analyst evaluates the impacted decisions to decide whether the proposed change is to be applied. In Step 3, *Apply the proposed change*, the analyst applies the proposed change to the corresponding decision. Steps 1, 2, and 3 are iterative: the analyst proposes and applies changes until all the required changes are considered. We discuss these three steps in Section 5.

Table 3
Change types for diagram decisions.

Change types
. Add a decision
. Delete a decision
. Update a decision
- Select some unselected variant use case(s)
- Unselect some selected variant use case(s)
- Unselect some selected variant use case(s) and select some unselected variant use case(s)

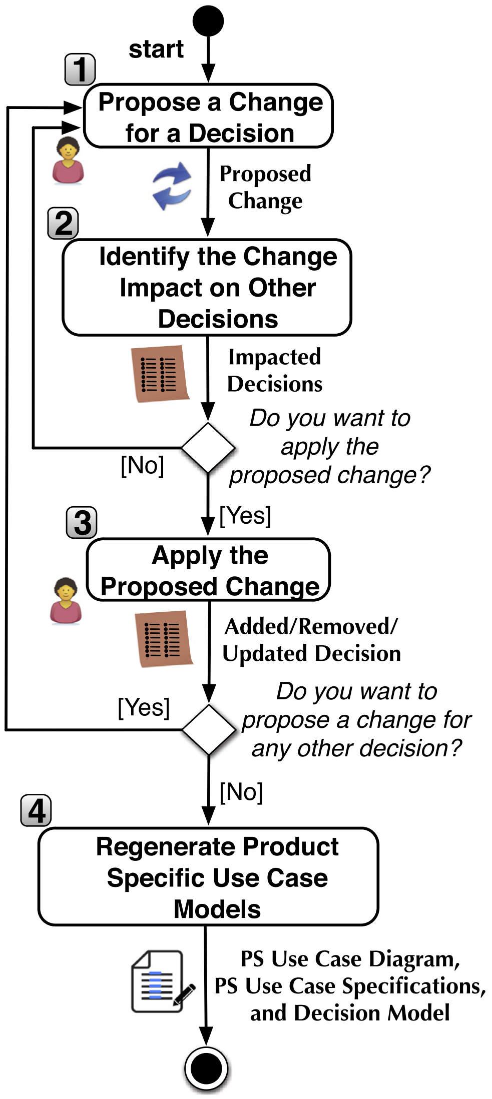

start
1
Propose a Change for a Decision
Proposed Change
2
Identify the Change Impact on Other Decisions
Impacted Decisions
Do you want to apply the proposed change?
[No]
[Yes]
3
Apply the Proposed Change
Added/Removed/ Updated Decision
[Yes]
Do you want to propose a change for any other decision?
[No]
4
Regenerate Product Specific Use Case Models
PS Use Case Diagram, PS Use Case Specifications, and Decision Model

Fig. 4. Overview of the approach.

After the analyst applies all the required changes to the configuration decisions, in Step 4, *Regenerate product specific use case models*, the PS use case diagram and specifications are incrementally and automatically regenerated for only the changed decisions (*Challenge 2*). The details of the step are described in Section 6.

5. Identification of change impact on decisions for PL use case diagrams

Decision-making during product configuration is iterative. The analyst may update or delete some of the prior decisions while new decisions are being made for undecided variants. A diagram decision is about selecting, for the product, variant use cases in

Table 4
Example decisions for the PL use case diagram in Fig. 5.

Decision ID	**Explanation of the decision**
d1	Selecting UC1 and UC2 in VP1
d2	Selecting UC9 and unselecting UC10 in VP4
d3	Unselecting UC15 in VP6
d2’	Selecting UC9 and UC10 in VP4

the variation point. Table 3 lists the change types for diagram decisions.

The first two change types in Table 3 are obvious manipulations over the diagram decisions. The subtypes of ‘Update a Decision’ match the (un)selection of variant use cases in a variation point.

When a change is introduced to a diagram decision, the analyst needs to identify not only the impacted decisions but also the reason of the impact, e.g., violation of dependency constraints, new restrictions for subsequent decisions, and contradicting decision restrictions (*Challenge 1*).

Automated analysis for configuration support often relies on translating models to propositional logic and using satisfiability (SAT) solvers (Benavides et al., 2010; Mendonca et al., 2009). As we discuss in Section 9, employing SAT solvers can help identify impacted decisions but does not provide further explanations regarding the reason of the impact. However, this is critical for the analysts to make further decisions based on the change impact. To this end, we devised a custom change impact analysis algorithm that identifies the impact of diagram decision changes on other diagram decisions and provide an explanation regarding the cause of the impact. In the following, we explain the steps of the algorithm with an illustrative example depicted in Fig. 5. The example is a slight adaptation of a piece of our industrial case study since we needed some additional modeling elements to illustrate the complete set of features of the algorithm. Fig. 5 depicts an example PL use case diagram including seven variation points, fourteen variant use cases, and one essential use case.

As an example, let us assume the analyst makes the decision *d1* for *VP1*, which is selecting *UC1* and *UC2* for the product. Further, the decisions *d2* and *d3* are made for *VP4* and *VP6*, which are selecting *UC9* and unselecting *UC10* in *VP4* and unselecting *UC15* in *VP6*, respectively. Further, let us assume that the analyst proposes to change *d2* with *d2′* by selecting unselected *UC10* in *VP4* (see Table 4).

Fig. 6 describes the change impact analysis algorithm for diagram decisions. The algorithm takes a set of prior decisions, a PL use case diagram, and a decision change as input. It reports added and deleted contradicting prior decisions, added and deleted restrictions for subsequent decisions, and sets of added and deleted contradicting restrictions as output.

The decision *d*, which precedes the decision change *c*, is a quadruple of the variation point *vp*, the use case *uc* including *vp*, the set of selected variant use cases *SUC* in *vp*, and the set of unselected variant use cases *NSUC* in *vp* (Line 6). The decision *d′*, which results from the change *c*, is given as a similar

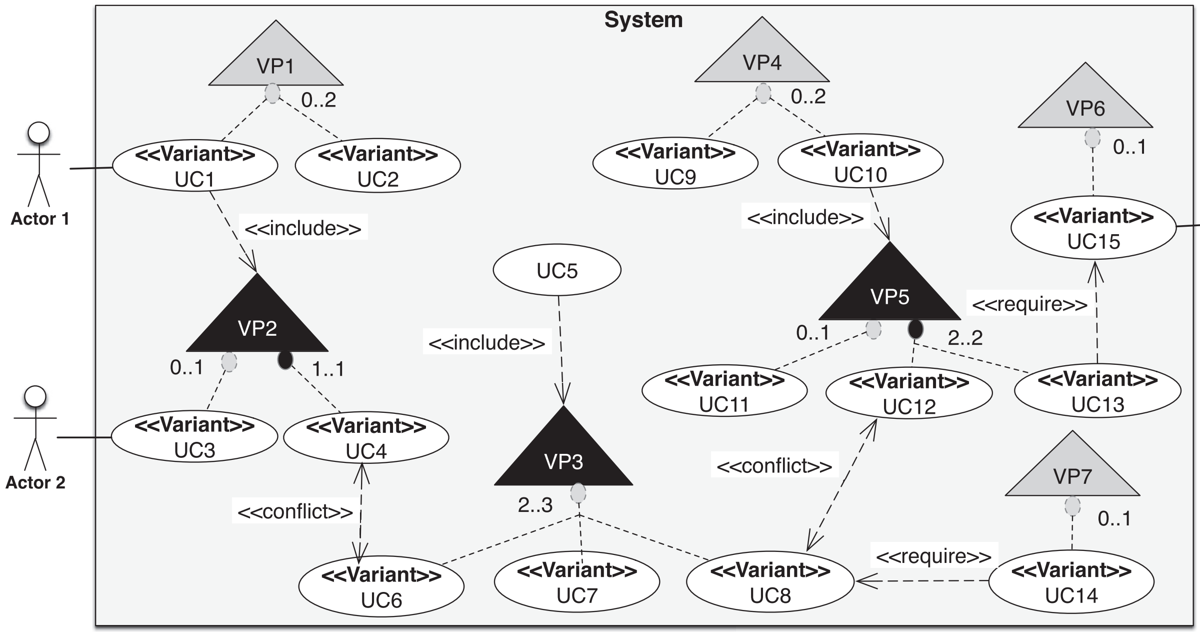


Fig. 5. An example product line use case diagram.

```
Input: Set of prior decisions ($DC$), PL use case diagram ($PLD$),
       Decision change ($c$)
Output: Sets of added and deleted contradicting prior
        decisions, added and deleted restrictions for subsequent
        decisions, and added and deleted sets of contradicting restrictions
1.  Let $p$ be a pair $(vp, uc)$ such that $vp$ is a variation point either
    included by a use case $uc$, or $vp$ is not included by any use case
2.  Let $SUC$ be the set of variant use cases selected in $p.vp$
    before the change $c$
3.  Let $NSUC$ be the set of variant use cases unselected in $p.vp$
    before the change $c$
4.  Let $SUC'$ be the set of variant use cases selected in $p.vp$
    after the change $c$
5.  Let $NSUC'$ be the set of variant use cases unselected in $p.vp$
    after the change $c$
6.  Let $d$ be the quadruple $(dp.vp, dp.uc, SUC, NSUC)$
7.  Let $d'$ be the quadruple $(dp.vp, dp.uc, SUC', NSUC')$
8.  Let $CD$ and $CD'$ be the empty sets for contradicting decisions
9.  Let $R$ and $R'$ be the empty sets for restrictions on further decisions
10. Let $CR$ and $CR'$ be the empty sets for sets of contradicting
    restrictions
11. $CD \leftarrow$ checkPriorDecisionConsistency($DC$, $d$, $PLD$)
12. $CD' \leftarrow$ checkPriorDecisionConsistency($DC$, $d'$, $PLD$)
13. $R \leftarrow$ inferDecisionRestrictions($DC \cup \{d\}$, $PLD$)
14. $R' \leftarrow$ inferDecisionRestrictions($DC \cup \{d'\}$, $PLD$)
15. $CR \leftarrow$ checkDecisionRestrictions($R$, $PLD$)
16. $CR' \leftarrow$ checkDecisionRestrictions($R'$, $PLD$)
17. return $(CD' \backslash CD,\ CD \backslash CD',\ R' \backslash R,\ R \backslash R',\ CR' \backslash CR,\ CR \backslash CR')$
```

Fig. 6. Change impact analysis algorithm for diagram decisions.

quadruple (Line 7). For instance, in our example, *d2* and *d2′* are (*VP*4, *null*, {*UC*9}, {*UC*10}) and (*VP*4, *null*, {*UC*9, *UC*10}, ∅), respectively.

We call *check* and *infer* functions with *d* and *d′* to identify the impact of *c* (Lines 11–16).

- ***checkPriorDecisionConsistency*** determines contradicting prior decisions for variation points. Two or more diagram decisions may contradict each other if they result in violating some variation point and variant dependency constraints (i.e., *require* and *conflict*);
- ***inferDecisionRestrictions*** determines restrictions on the selection of variant use cases in undecided variation points. The existing decisions may entail (un)selection of some variant use cases in subsequent decisions through the variation point and variant dependencies;
- ***checkDecisionRestrictions*** determines contradicting restrictions for subsequent decisions. Two or more decision restrictions may contradict each other if they result in violating some cardinality constraints or result in selecting and unselecting the same variant use case.

The algorithm of *checkPriorDecisionConsistency* was developed as part of our configurator, *PUMConf*, described in our previous work (Hajri et al., 2016a, 2016b). The algorithm is based on mapping variation points, use cases and variant dependencies to propositional logic formulas. For a given decision regarding a variation point, it only checks the satisfaction of the propositional formulas derived from the dependencies of the variation point (Hajri et al., 2016a). For example, assume there are two conflicting variant use cases *Ua* and *Ub* (i.e., *Ua* conflicts with *Ub*). *Ua* and *Ub* are selected in decisions *Da* and *Db*, respectively. *Da* and *Db* are contradicting because *Ua* and *Ub* cannot exist for the same product (i.e., ¬(*Ua*∧*Ub*)).

For changing *d2* with *d2′* in Fig. 5, we call *checkPriorDecisionConsistency* first with *d2* (*DC* = {*d*1, *d*3} and *d* = *d*2 in Line 11), and then with *d2′* (*DC* = {*d*1, *d*3} and *d* = *d*2′ in Line 12). For *d2*, the function returns no contradicting prior decision. When *UC10* is unselected in *d2, UC11, UC12* and *UC13* in *VP5* are automatically unselected because there is no other use case including *VP5. UC13* which is unselected in *d2* requires *UC15* which is unselected in *d3* (i.e., *U13* → *U15*). Therefore, *d2* and *d3* are not contradicting. *UC12* and *UC13* are automatically selected in *d2′* because of selected *UC10* and the mandatory variability relation in *VP5. UC13* which is selected in *d2′* requires *UC15* which is unselected in *d3*. Therefore, for *d2′*, *checkPriorDecisionConsistency* returns *d3* contradicting *d2′*. The decision change introduces a new contradiction with *d3* ($CD' \backslash CD = \{d3\}$ in Line 17). No existing contradiction is removed by the change ($CD \backslash CD' = \emptyset$ in Line 17). *d3* is impacted since it contradicts *d2′* after the change.

As part of our impact analysis approach, the algorithm of *inferDecisionRestrictions* also relies on propositional logic mappings for variation points, use cases and variant dependencies (see

Table 5
Restrictions inferred from the example decisions in Table 4.

Restriction ID	Explanation of the restriction
r1	UC6 in VP3 should not be selected
r2	UC8 in VP3 should not be selected
r3	UC14 in VP7 should not be selected

Section 5.1 for the details of the algorithm). For a given decision regarding a variation point, *inferDecisionRestrictions* infers restrictions for subsequent decisions by only checking the satisfaction of the propositional logic formulas derived from the dependencies of the variation point. Assume two variant use cases *Ua* and *Ub* in variation points *Va* and *Vb* with a *requires* relation (i.e., *Ua* requires *Ub*). *Ua* is selected in decision *Da* for *Va* while there is no decision yet for *Vb*. The subsequent decision for *Vb* is restricted as *Ub* needs to be selected to avoid a contradiction with *Da* because of the *requires* relation (i.e., $Ua \rightarrow Ub$).

For the input decisions *d1, d2* and *d3, inferDecisionRestrictions* returns restriction *r1* for *UC6* in *VP3* (Line 13). When *UC1* is selected in *d1, UC4* is automatically selected because of the mandatory variability relation in *VP2. UC4* conflicts with *UC6*, and there is no decision made for *UC6*. The selection of *UC4* restricts the subsequent decision for *VP3* so that *UC6* should not be selected to avoid the contradiction with *d1* (i.e., restriction *r1* in Table 5). For *d1, d2′* and *d3*, the function returns restrictions *r1* for *UC6* in *VP3, r2* for *UC8* in *VP3*, and *r3* for *UC14* in *VP7* (Line 14). *UC12* in *VP5* is automatically selected in *d2′*, and it conflicts with *UC8* in *VP3* for which there is no decision made yet. The selection of *UC12* restricts the subsequent decision for *VP3* through the conflicts relation that *UC8* should not be selected (i.e., *r2* in Table 5). The restriction for the subsequent decision for *UC8* restricts the subsequent decision for *VP7* through the *requires* relation (i.e., *r3* in Table 5). If *UC8* should not be selected, *UC14* should also not be selected since it requires *UC8*. The subsequent decisions for *VP3* and *VP7* are impacted by the change because of the new restrictions ($R' \backslash R = \{r2, r3\}$ and $R \backslash R' = \emptyset$ in Line 17).

We devise the algorithm of *checkDecisionRestrictions* as part of our change impact analysis approach (see Section 5.2 for the details of the algorithm). For a given set of decision restrictions, *checkDecisionRestrictions* identifies contradicting restrictions for subsequent decisions in terms of violating cardinality constraints and restricting the same variant use cases for being selected and unselected. For example, assume there are two restrictions *r1* and *r2* which state the variant use cases *Ua* and *Ub* in the variation point *V* need to be selected, respectively. *V* has the [0..1] cardinality constraint. *r1* and *r2* do not comply with this cardinality constraint.

For the restrictions before the decision change in our example (i.e., *r1*), *checkDecisionRestrictions* does not return any contradicting restriction (Line 15). *r1* restricts the subsequent decision for *VP3* so that *UC6* should not be selected. There is no other restriction, and *r1* complies with the cardinality constraint of *VP3* (i.e., [2..3]). For the restrictions after the change (i.e., *r1, r2* and *r3*), the function returns {{*r1, r2*}}, i.e., the set of sets of contradicting restrictions. *UC6* and *UC8* in *VP3* should not be selected according to *r1* and *r2*, respectively. The cardinality constraint in *VP3* requires at least two of three variant use cases in *VP3* to be selected. Therefore, *r1* and *r2* cannot exist together because of the cardinality constraint. A new contradiction is introduced after the decision change ($CR' \backslash CR = \{\{r1, r2\}\}$ and $CR \backslash CR' = \emptyset$ in Line 17). To resolve it, the decisions causing it need to be updated. *r2* is inferred from *d2′* through *UC12*, while *d1* results in *r1* through *UC4*. Therefore, *d1* is identified as impacted.

Changing *d2* with *d2′* impacts *d1* for *VP1, d3* for *VP6* and the subsequent decisions for *VP3* and *VP7*.

5.1. Identification of subsequent decision restrictions

Decision restrictions are inferred by mapping variation points, use cases and variant dependencies to propositional logic formulas. We assume that a PL use case diagram *PLD* is defined as a set, where each use case is a member of the set. The PL diagram consists of *n* use cases $PLD = \{u_1, \ldots, u_n\}$; each use case u_i in *PLD* is represented by a boolean variable with the same name. Boolean variable u_i evaluates to *true* if use case u_i is selected and *false* otherwise. If there is no decision made yet for use case u_i, variable u_i is not valued (*unknown*). Please note that all essential use cases are automatically selected.

Fig. 7 provides the corresponding propositional formulas for each pattern involving dependencies, variation points, and variant use cases, where propositions capture logical relationships among variant use cases. For instance, according to the corresponding propositional formula in Fig. 7(a), if use case UCA_m is selected for a product then this logically implies that use case UCB_n is also selected. Fig. 7(c) depicts the mapping when there is a *require* dependency between two variation points *A* and *B*. In such a case, if one of the variant use cases in variation point *A* ($UCA_1 \vee \ldots \vee UCA_m$) is selected, then at least one of the variant use cases in variation point *B* ($\rightarrow UCB_1 \vee \ldots \vee UCB_n$) should also be selected.

Fig. 8 describes the algorithm for *inferDecisionRestrictions*. To illustrate the algorithm, we rely on the example with the input decisions *d1, d2′* and *d3* in Fig. 5. For each decision *d* in the set of decisions *D*, the algorithm calls some *infer* functions to identify the decision restrictions for subsequent decisions in which the propositional logic formulas, derived from the dependencies to/from the diagram elements decided in *d*, are satisfied (Lines 11, 12, 15, 18, 19 and 21). Each *infer* function in Fig. 8 infers restrictions for subsequent decisions using the propositional formulas in one or more mappings in Fig. 7.

- ***inferConflictingVP*** uses the formulas in Fig. 7(d) and (g) to infer decision restrictions for variation points and use cases conflicting with selected variation point *vp* in decision *d*;
- ***inferConflictingUC*** uses the formulas in Fig. 7(b) and (g) to infer decision restrictions for variation points and variant use cases conflicting with selected variant use case *u* in *d*;
- ***inferRequiringVP*** uses the formulas in Fig. 7(c) and (e) to infer decision restrictions for variation points and variant use cases requiring unselected variation point *vp* in *d*;
- ***inferRequiredByVP*** uses the formulas in Fig. 7(c) and (f) to infer decision restrictions for variation points and variant use cases required by selected variation point *vp* in *d*;
- ***inferRequiringUC*** uses the formulas in Fig. 7(a) and (f) to infer decision restrictions for variation points and variant use cases requiring unselected variant use case *u* in *d*;
- ***inferRequiredByUC*** uses the formulas in Fig. 7(a) and (e) to infer decision restrictions for variation points and variant use cases required by selected variant use case *u* in *d*.

In Fig. 5, *inferConflictingUC* infers *r1* and *r2* from *UC4*, automatically selected in *d1*, and from *UC12*, automatically selected in *d2′*, respectively. The algorithm of *inferConflictingUC* is given in Fig. 9. For the rest of the *infer* functions, the reader is referred to Supplementary Material[1].

The algorithm of *inferConflictingUC* in Fig. 9 uses the formulas in Fig. 7(b) and (g) to restrict the subsequent decisions for variant use cases and variation points that conflict with selected use case

[1] http://people.svv.lu/hajri/change_impact/SupplementaryMaterial.pdf.

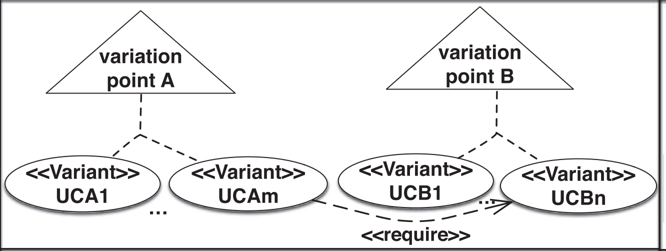
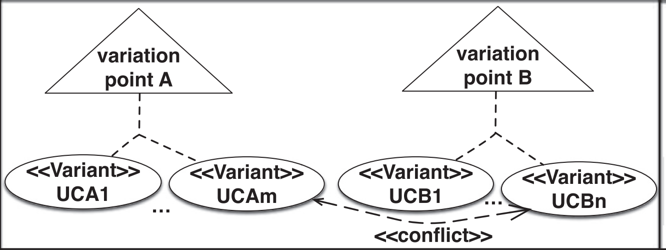
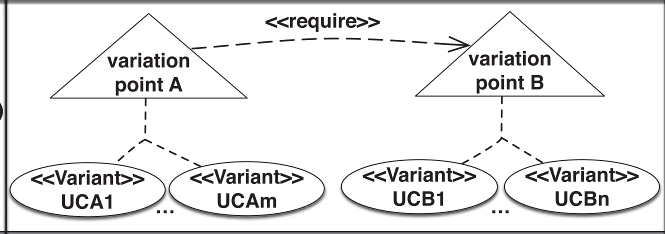
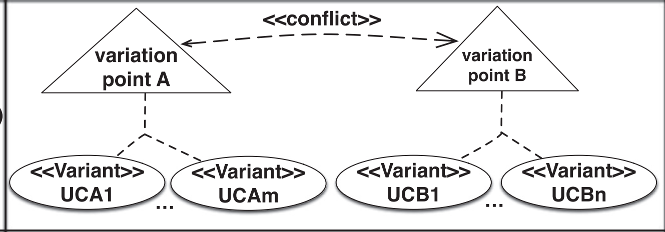
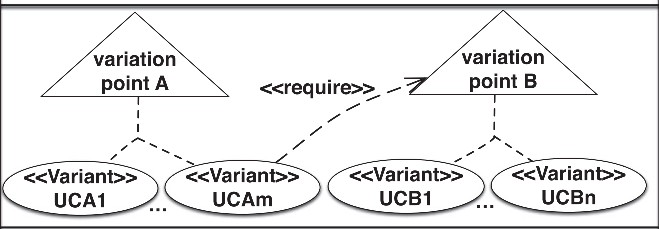
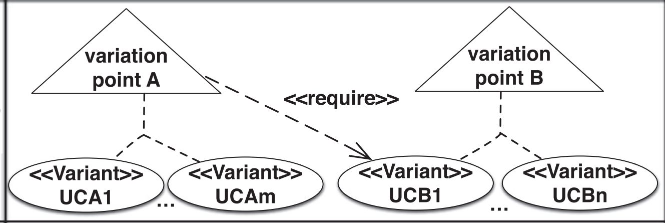
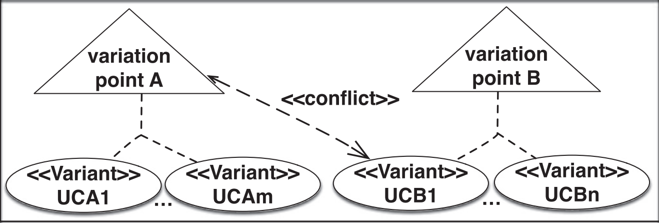
	Dependency, Variation Point, and Variant Use Case	Propositional Logic Mapping
(a)	 variation point A variation point B <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn <<require>>	$UCAm \longrightarrow UCBn \quad (m \geqq 1 \text{ and } n \geqq 1)$
(b)	 variation point A variation point B <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn <<conflict>>	$\neg (UCAm \wedge UCBn) \quad (m \geqq 1 \text{ and } n \geqq 1)$
(c)	 <<require>> variation point A variation point B <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn	$(UCA1 \vee \ldots \vee UCAm) \longrightarrow (UCB1 \vee \ldots \vee UCBn)$ $(m \geqq 1 \text{ and } n \geqq 1)$
(d)	 <<conflict>> variation point A variation point B <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn	$\neg ((UCA1 \vee \ldots \vee UCAm) \wedge (UCB1 \vee \ldots \vee UCBn))$ $(m \geqq 1 \text{ and } n \geqq 1)$
(e)	 variation point A variation point B <<require>> <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn	$UCAm \longrightarrow (UCB1 \vee \ldots \vee UCBn)$ $(m \geqq 1 \text{ and } n \geqq 1)$
(f)	 variation point A variation point B <<require>> <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn	$(UCA1 \vee \ldots \vee UCAm) \longrightarrow UCB1$ $(m \geqq 1 \text{ and } n \geqq 1)$
(g)	 variation point A variation point B <<conflict>> <<Variant>> UCA1 <<Variant>> UCAm <<Variant>> UCB1 <<Variant>> UCBn	$\neg ((UCA1 \vee \ldots \vee UCAm) \wedge UCB1)$ $(m \geqq 1 \text{ and } n \geqq 1)$

Fig. 7. Mapping from PL use case diagram to propositional logic.

```
Input: Set of diagram decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)
1.  $DC \leftarrow D$
2.  Let $IR$ be the empty set for inferred restrictions
3.  while ($DC \neq \emptyset$) do
4.      $d \in DC$
5.      Let $vp$ be the variation point in decision $d$
6.      Let $SUC$ be the set of selected variant use cases in decision $d$
7.      Let $NSUC$ be the set of unselected variant use cases in decision $d$.
8.      Let $SE$ be the set of variant use cases automatically selected
        when the variant use cases in $SUC$ are selected in decision $d$
9.      Let $NSE$ be the set of variant use cases automatically unselected
        when the variant use cases in $NSUC$ are unselected in $d$
10.     foreach ($u \in (SUC \cup SE)$) do
11.         $IR \leftarrow IR \cup$ inferRequiredByUC($u, D, PLD$)
12.         $IR \leftarrow IR \cup$ inferConflictingUC($u, D, PLD$)
13.     end foreach
14.     foreach ($u \in (NSUC \cup NSE)$) do
15.         $IR \leftarrow IR \cup$ inferRequiringUC($u, D, PLD$)
16.     end foreach
17.     if ($SUC \neq \emptyset$) then
18.         $IR \leftarrow IR \cup$ inferRequiredByVP($SUC, vp, D, PLD$)
19.         $IR \leftarrow IR \cup$ inferConflictingVP($SUC, vp, D, PLD$)
20.     else
21.         $IR \leftarrow IR \cup$ inferRequiringVP($NSUC, vp, D, PLD$)
22.     end if
23.     $DC \leftarrow DC \backslash \{d\}$
24. end while
25. return $IR$
```

Fig. 8. Algorithm for *inferDecisionRestrictions*.

u. For instance, in Fig. 7(b), when *UCAm* is selected, it checks if there is any decision made for *UCBn*. If there is no decision for *UCBn*, the subsequent decision is restricted that *UCBn* should not be selected.

inferConflictingUC takes as input selected variant use case *u*, set of decisions *D*, and PL use case diagram *PLD*, while it returns the set of decision restrictions *IR*. A decision restriction is given as a triple (*uc, vpo, b*) where *uc* is a variant use case, *vpo* is the variation point of *uc* and *b* is a boolean variable (Line 1 in Fig. 9). If the restriction is about the whole variation point, not about a single variant use case in the variation point, *uc* becomes *null*. *b* indicates whether the variant use case(s) should be selected or not. For instance, (*null, Va, false*) states that none of the variant use cases in variation point *Va* should be selected, while (*UCA1, Va, true*) states variant use case *UCA1* in *Va* should be selected.

The algorithm starts with identifying the variant use cases conflicting with the input selected variant use case *u* (see Fig. 7(b)). The conflicting variant use cases which have not been decided yet should be unselected in subsequent decisions (Line 8). The subsequent decisions should also be restricted for other undecided variant use cases and variation points which require those conflicting use cases (Lines 9 and 10). When the conflicting variant use cases are unselected because of the restriction, some variant use cases in the variation points included by those conflicting use cases might also be automatically unselected, and therefore the corresponding subsequent decisions need to be restricted (Lines 11–17). In our example, *UC4* is selected in *d1* (i.e., *u* = *UC4*), and only *UC6* conflicts with *UC4* (i.e., *CUC* = {*UC6*} in Line 3). There is no decision made for *UC6* which should not be selected (i.e., *r1* = (*UC6, VP3, false*) in Line 8). *UC4* does not include any variation point where variant use cases might be automatically unselected (i.e., *AUC* = ∅ in Line 12). As another input use case, *UC12* is selected in *d2′* (i.e., *u* = *UC12*), and only *UC8* conflicts with *UC12* (i.e., *CUC* = {*UC8*} in Line 3). There is no decision made for *UC8*. Therefore, it should not be selected in subsequent decisions (i.e., *r2* = (*UC8, VP3, false*) in Line 8). *UC8* is required by *UC14* in *VP7* which has not been decided yet

```
Input: Use case ($u$), Set of decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)
1.  Let a triple $(uc, vpo, b)$ denote a decision restriction
    where $uc$ is a variant use case, $vpo$ is the variation point of $uc$,
    and $b$ is a boolean variable
2.  Let $IR$ be the empty set for inferred restrictions
3.  Let $CUC$ be the set of variant use cases conflicting with $u$
4.  Let $CVP$ be the set of variation points conflicting with $u$
5.  foreach ($c \in CUC$) do
6.      if ((there is a subsequent decision to be made for $c$) and
            ($c$ has not been selected in prior decisions in $D$)) then
7.          Let $vp$ be the variation point of $c$
8.          $IR \leftarrow IR \cup \{(c, vp, false)\}$
9.          $IR \leftarrow IR \cup$ inferRequiringUC($c, D, PLD$)
10.         $IR \leftarrow IR \cup$ inferRequiringVP($\{c\}, vp, D, PLD$)
11.         Let $AUC$ be the set of variant use cases automatically
            unselected when $c$ is unselected
12.         foreach ($a \in AUC$) do
13.             Let $p$ be the variation point of $a$
14.             $IR \leftarrow IR \cup \{(a, p, false)\}$
15.             $IR \leftarrow IR \cup$ inferRequiringUC($a, D, PLD$)
16.             $IR \leftarrow IR \cup$ inferRequiringVP($\{a\}, p, D, PLD$)
17.         end foreach
18.     end if
19. end foreach
20. foreach ($p \in CVP$) do
21.     if ((there is a subsequent decision to be made for $p$) and
            (none of the variant use cases in $p$ has been selected in
            prior decisions in $D$)) then
22.         Let $UC$ be the set of variant use cases in $p$
23.         $IR \leftarrow IR \cup \{(null, p, false)\}$
24.         $IR \leftarrow IR \cup$ inferRequiringVP($UC, p, D, PLD$)
25.         foreach ($vc \in UC$) do
26.             $IR \leftarrow IR \cup$ inferRequiringUC($vc, D, PLD$)
27.         end foreach
28.         Let $AU$ be the set of variant use cases automatically unselected
            when the variant use cases in $p$ are unselected
29.         foreach ($vuc \in AU$) do
30.             Let $vp$ be the variation point of $vuc$
31.             $IR \leftarrow IR \cup \{(vuc, vp, false)\}$
32.             $IR \leftarrow IR \cup$ inferRequiringUC($vuc, D, PLD$)
33.             $IR \leftarrow IR \cup$ inferRequiringVP($\{vuc\}, vp, D, PLD$)
34.         end foreach
35.     end if
36. end foreach
37. return $IR$
```

Fig. 9. Algorithm for *inferConflictingUC*.

(see *inferRequiringUC* in Line 9). Another decision restriction *r3* is inferred for *VP7* (i.e., (*UC14, VP7, false*)).

The algorithm also identifies the variation points conflicting with the input selected variant use case *u* (see Fig. 7(g)). The variant use cases in the undecided conflicting variation points should be unselected in the subsequent decisions (Line 23). The variant use cases and variation points requiring those conflicting variation points or their variant use cases should also be unselected in the subsequent decisions (Lines 24–27). The subsequent decisions are restricted for variant use cases which are automatically unselected when the variant use cases in the undecided conflicting variation points are unselected (Line 28–34). For the example in Fig. 5, there is no variation point conflicting with the input use cases. The algorithm returns all the inferred restrictions (Line 37).

5.2. Identification of contradicting decision restrictions

For a given set of decision restrictions, our approach identifies (i) restrictions violating cardinality constraints in variation points and (ii) contradicting restrictions regarding the selection and unselection of the same variant use case. Assume we have two re-

```
Input: Set of decision restrictions ($R$), PL use case diagram ($PLD$)
Output: Set of sets of contradicting decisions ($CR$)
1.  Let a triple $(uc, vpo, b)$ denote a decision restriction
    where $uc$ is a variant use case, $vpo$ is the variation point of $uc$,
    and $b$ is a boolean variable
2.  Let $CR$ be the empty set for sets of contradicting restrictions
3.  Let $VP$ be the set of variation points in $PLD$
4.  foreach $(p \in VP)$ do
5.    Let $DR$ be the set of decision restrictions in $R$ for $p$
6.    $IR \leftarrow DR$
7.    foreach $(r \in DR)$ do
8.      $IR \leftarrow IR \setminus \{r\}$
9.      foreach $(e \in IR)$ do
10.       if $(e.uc = r.uc$ and $e.b \neq r.b)$ then
11.         $CR \leftarrow CR \cup \{\{e, r\}\}$
12.       end if
13.       if $(r.uc = null)$ then
14.         if $(r.b = false)$ then
15.           if $(e.b = true)$ then
16.             $CR \leftarrow CR \cup \{\{e, r\}\}$
17.           end if
18.         end if
19.       end if
20.     end foreach
21.     if $(r.uc = null$ and $r.b = true)$ then
22.       $CR \leftarrow CR \cup$ checkSeveralRestrictions$(p, DR, PLD)$
23.     end if
24.   end foreach
25.   $CR \leftarrow CR \cup$ checkCardinality$(p, DR, PLD)$
26. end foreach
27. return $CR$
```

Fig. 10. Algorithm for *checkDecisionRestrictions*.

strictions *rt1* and *rt2* where *rt1* = (*null, Va, false*) and *rt2* = (*Ua, Va, true*). *rt1* and *rt2* contradict each other because *Ua* in *Va* should be selected according to *rt2* while *rt1* states all the variant use cases in *Va* should be unselected.

Fig. 10 describes the algorithm of *checkDecisionRestrictions* that identifies contradicting restrictions. A contradiction is described as a set of contradicting decisions. For each variation point *p* in the PL diagram (Lines 3 and 4), the algorithm first checks if there are multiple restrictions (Lines 10–12). A contradiction is identified for two restrictions requiring the selection and unselection of the same variant use case (Lines 11 and 16). More than two restrictions result in a contradiction where a restriction requires at least one variant use case in a variation point to be selected while each variant use case in the same variation point is required to be unselected by yet another restriction (Line 22). Restrictions which do not comply with cardinality constraints also contradict each other (Line 25). We call two functions in Fig. 10 (Lines 22 and 25).

- ***checkSeveralRestrictions*** returns a set of contradictions for restrictions in *DR* in which more than two restrictions for variation point *p* contradict each other;
- ***checkCardinality*** returns a set of contradictions for restrictions in *DR* which do not comply with the cardinality constraints in variation point *p*.

For example, *checkDecisionRestrictions* checks the example restrictions for each variation point in Fig. 5 where *R* = {*r1, r2, r3*} and *PLD* is Fig. 5. Restrictions *r1* and *r2* apply to the subsequent decision in *VP3* while *r3* restricts another subsequent decision in *VP7*. *r1* and *r2* restrict the decision for different variant use cases in *VP3* (i.e., *r1.uc* ≠ *r2.uc* in Line 10, *r1.uc* ≠ *null* in Line 13, and *r2.uc* ≠ *null* in Line 13). *UC6* and *UC8* in *VP3* should be unselected according to *r1* and *r2* while the cardinality constraint requires at least two of three variant use cases in *VP3* to be selected (i.e., *checkCardinality* returns {{*r1, r2*}} in Line 25). *r3* complies with the cardinality constraint in *VP7*. *checkDecisionRestrictions* returns {{*r1, r2*}} for the contradicting restrictions in Fig. 5 (Line 27).

To summarize, here are the main building blocks of our change impact analysis algorithm:

- We automatically select variant use cases via the *include* relations and the mandatory variability relations. For instance, *UC4* is automatically selected via the include relation (i.e., *UC1* includes *VP2*) and the mandatory variability relation (i.e., the relation with the cardinality constraint ‘1..1’) when the user selects *UC1*.
- We use the *requires* and *conflicts* relations to infer restrictions on subsequent decisions. For instance, when *UC4* is automatically selected, the subsequent decision for *VP3* is restricted to *UC6* being unselected because of the *conflicts* relation between *UC4* and *UC6*. If the user selects *UC14*, the subsequent decision for *VP3* is restricted to *UC8* being selected because of the *requires* relation between *UC14* and *UC8*.
- Our approach does not have any limitation on the number of navigation steps in the PL use case diagram since we have recursion in the *infer* functions which traverse the graph of dependencies we derive from the PL use case diagram. For instance, the function *inferConflictingUC* in Fig. 9 has function calls to infer further restrictions (Lines 9, 10, 15, 16, 24, 26, 32 and 33). We do not have any upper bound in our reasoning. Assume there are variant use cases *A, B, C* and *D* where *A* requires B conflicting with *C* required by *D*. When the user selects *A*, we infer the restrictions that *B* should be selected, and *C* and *D* should not be selected.

6. Incremental reconfiguration of PS use case models

After all the decision changes are made, the PS use case models need to be incrementally reconfigured (*Challenge 2*). The reconfiguration of PS models is implemented as a pipeline (see Fig. 11). Configuration decisions are captured in a decision model during the decision-making process. The decision model conforms to a decision metamodel, described in our prior work (Hajri et al., 2015). PUMConf keeps two decision models, i.e., the decision model before changes (*M1* in Fig. 11) and the decision model after changes (*M2* in Fig. 11). Fig. 12 provides the decision metamodel and the two input decision models for the PL use case models in Fig. 1 and Table 1.

The pipeline takes the decision models, and the PS diagram and specifications as input. The PS models are reconfigured, as output, together with an impact report, i.e., list of reconfigured parts of the PS models. The pipeline has three steps (Fig. 11).

In Step 1, *Matching decision model elements*, the structural differencing of *M1* and *M2* is done by looking for the correspondences in *M1* and *M2*. To that end, we devise an algorithm that identifies the matching model elements in *M1* and *M2*. The output of Step 1 is the corresponding elements, representing decisions for the same variations, in *M1* and *M2* (Section 6.1).

The decision metamodel in Fig. 12(a) includes the main use case elements for which the user makes decisions (i.e., variation point, optional step, optional alternative flow, and variant order). In a variation point, the user selects variant use cases to be included for the product. For PL use case specifications, the user selects optional steps and alternative flows to be included and determines the order of steps (variant order). Therefore, the matching elements in Step 1 are the pairs of variation points and use cases including the variation points, the pairs of use cases and optional alternative flows in the use cases, and the triples of use cases, flows in the use cases, and optional steps in the flows.

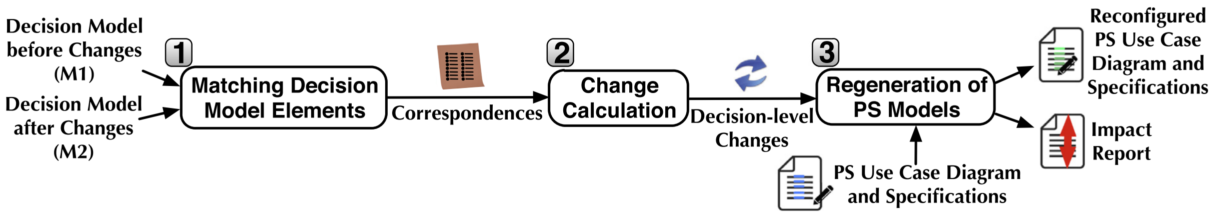

Decision Model before Changes (M1)
Decision Model after Changes (M2)
1
Matching Decision Model Elements
Correspondences
2
Change Calculation
Decision-level Changes
3
Regeneration of PS Models
PS Use Case Diagram and Specifications
Reconfigured PS Use Case Diagram and Specifications
Impact Report

Fig. 11. Overview of the model differencing and regeneration pipeline.

In Step 2, *Change calculation*, decision-level changes are identified from the corresponding model elements (see Section 6.1). A set of elements in *M1* which does not have a corresponding set of elements in *M2* is considered to be a deleted decision, which we refer to as *DeleteDecision* in the decision-level changes. Analogously, a set of model elements in *M2* which does not have a corresponding set of elements in *M1* is considered to be added (*AddDecision*). Each set of corresponding model elements with non-identical attribute values (see the red-colored attributes in Fig. 12(c)) is considered to be a decision-level change of the type *UpdateDecision*. Alternatively, we could record changes during the decision-making process. However, the user might make changes cancelling previous changes or implying some further changes. In such a case, we would have to compute cancelled changes and infer new changes. To record these changes, the approach would also have to depend on IBM DOORS, in which it is integrated. We designed an approach which is as independent as possible from any requirements management tool.

In Step 3, *Regeneration of PS models*, the PS use case diagram and specifications are regenerated only for the added, deleted and updated decisions (see Section 6.2). For instance, use cases selected in the deleted decisions are removed from the PS use case models, while use cases selected in the added decisions are added in the PS models.

6.1. Model matching and change calculation

We devise an algorithm (see Fig. 13) for the first two pipeline steps, *Matching decision model elements* and *Change calculation*, in Fig. 11. The algorithm calls some *match* functions (Lines 7–9 in Fig. 13) to identify the corresponding model elements, which represent decisions for the same variations, in the input decision models. The *match* functions implement Step 1 in Fig. 11.

- ***matchDiagramDecisions*** returns the set of pairs (*variation point, use case*) matching in the decision models (*M1* and *M2*), which are capturing which variation points are included in the use cases involved in diagram decisions;
- ***matchFlowDecisions*** returns the set of pairs (*use case, optional alternative flow*) matching in the input decision models (*M1* and *M2*), which are capturing which optional alternative flows are in the use cases involved in flow decisions;
- ***matchStepDecisions*** returns the set of triples (*use case, flow, step*) matching in the input decision models (*M1* and *M2*), which are capturing which steps are in the flows of the use cases involved in step decisions.

The corresponding model elements in the decision models in Fig. 12(b) and (c) are as follows (Lines 7–9 in Fig. 13):

- For decisions in the variation points, $U3 = \{(B6,\ B7),\ (C6, C7)\}$,
- For decisions in the optional alternative flows, $F3 = \emptyset$,
- For decisions in the use case steps, $S3 = \{(B11,\ B12,\ B13),\ (B11,\ B12,\ B14),\ (B11,\ B12,\ B15),\ (B11, B12, B16),\ (B11, B12, B17),\ (C11,\ C12,\ C13),\ (C11,\ C12, C14),\ (C11, C12, C15),\ (C11, C12, C16),\ (C11,\ C12,\ C17)\}$.

A variant use case in a variation point (*vp*) may include another variation point (*vp′*). Changing the decision for *vp* may imply another decision to be added or deleted for *vp′*. As part of Step 2, *Change Calculation*, the algorithm first identifies deleted and added diagram decisions by checking the pairs of variation points and use cases which exist only in one of the input decision models ($(U1 \setminus U3)$ and $(U2 \setminus U3)$ in Lines 10–11). Similar checks are done for flow and step decisions in the specifications (Lines 10–11). For the decision models in Fig. 12, there is no deleted or added decision ($(U1 \setminus U3 = \emptyset)$, $(U2 \setminus U3 = \emptyset)$, $(F1 \setminus F3 = \emptyset)$, $(F2 \setminus F3 = \emptyset)$, $(S1 \setminus S3 = \emptyset)$, and $(S2 \setminus S3 = \emptyset)$).

The matching pairs of variation points and their including use cases represent decisions for the same variation point ((*B6,B7*) and (*C6, C7*) in Fig. 12(b) and (c)). If the selected variant use cases for the same variation point are not the same in *M1* and *M2*, the corresponding decision in *M1* is considered as updated in *M2* (Lines 12–19). The variant use case *Provide System User Data via Diagnostic Mode* of the variation point *Method of Providing Data* is unselected in *M1* (*B6, B7* and *B9* in Fig. 12(b)), but selected in *M2* (*C6, C7* and *C9* in Fig. 12(c)). The diagram decision for the pair (*B6, B7*) in *M1* is identified as updated (Line 17). To identify updated specification decisions, the algorithm compares decisions across *M1* and *M2* that involve optional alternative flows, optional steps and steps with a variant order (Lines 22–24, 28–30 and 31–33). In our example, the triples (*B11, B12, B14*), (*B11, B12, B15*), (*B11, B12, B16*), and (*B11, B12, B17*) in Fig. 12 represent updated decisions.

6.2. Regeneration of PS use case models

After all the changes are calculated by matching the corresponding model elements in the input decision models, the parts of PS use case models affected by the changed decisions are automatically regenerated (Step 3 in Fig. 11).

Our approach first handles the diagram decision changes to reconfigure the PS use case diagram. For selected variant use cases in the added diagram decisions (i.e., in the pairs (*vp, uc*) in *ADD* in Line 36 in Fig. 13), we generate the corresponding use cases and *include* relations in the PS diagram. For selected variant use cases in deleted diagram decisions (i.e., in the pairs (*vp, uc*) in *DELETE* in Line 36), we remove the corresponding use cases and *include* relations from the PS diagram. If a selected variant use case is unselected in an updated diagram decision (i.e., in the pairs (*vp, uc*) in *UPDATE* in Line 36), we remove the corresponding use case from the PS diagram. For unselected variant use cases which are selected in the updated diagram decisions, the corresponding use cases and *include* relations are added to the PS diagram. Fig. 14 gives the regenerated parts of the PS use case diagram in Fig. 2 for *M1* and *M2* in Fig. 12.

There is no added or deleted diagram decision in *M1* and *M2* in Fig. 12. The decision for the variation point *Method of Providing Data* (i.e., (*B6, B7*) in *UPDATE* in Line 36) is updated by selecting the variant use case *Provide System User Data via Diagnostic Mode*. Only the corresponding use case and its *include* relation are added to the PS use case diagram (red-colored in Fig. 14).

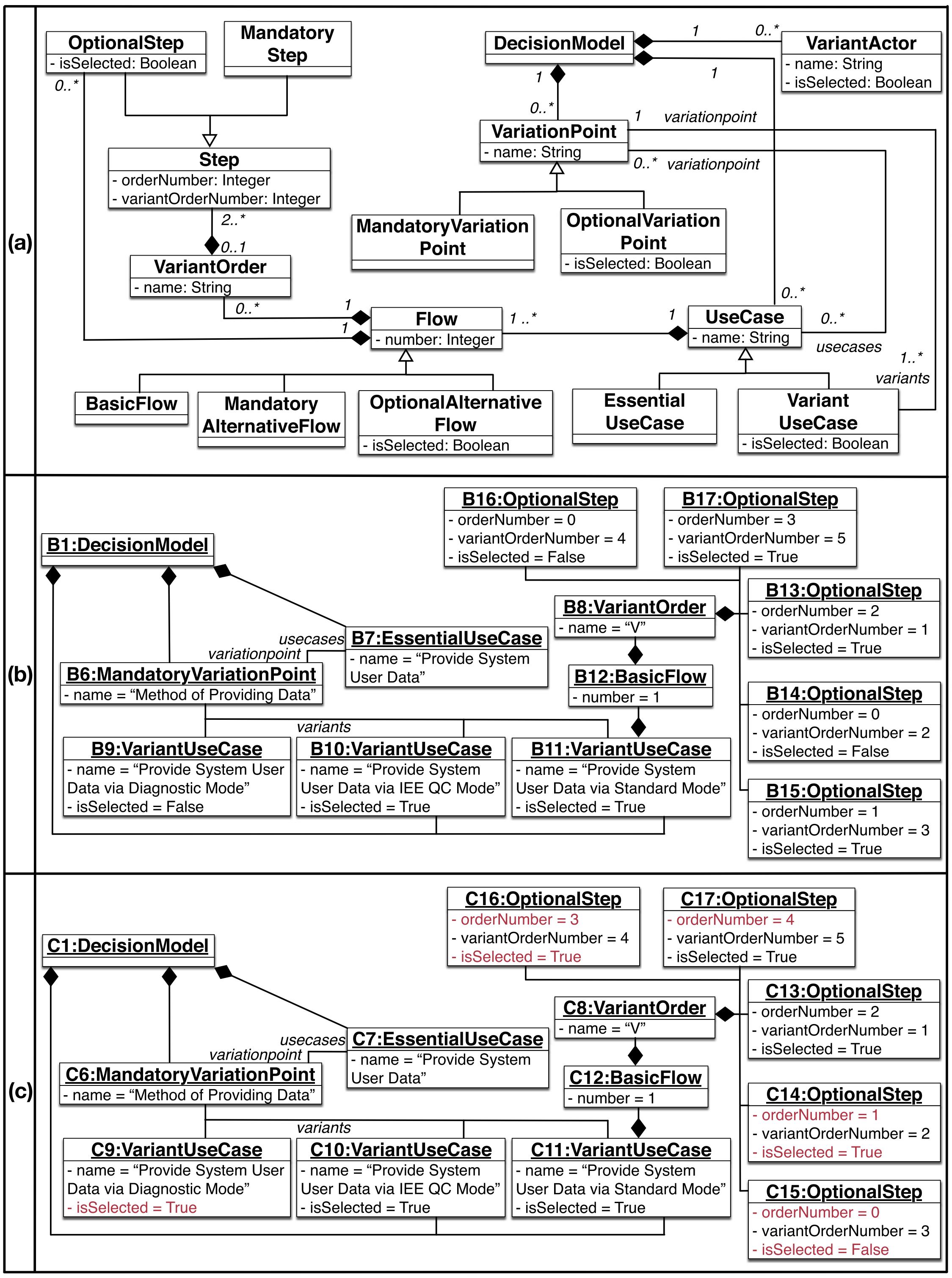


Fig. 12. (a) Decision metamodel, (b) Part of the example decision model before changes (*M1*), and (c) Part of the example decision model after changes (*M2*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Changes for diagram and specification decisions are used to regenerate the PS specifications. For diagram decision changes, we add or delete the corresponding use case specifications. Table 6 provides the regenerated parts of the PS specifications in Table 2, for *M1* and *M2* in Fig. 12.

For the variation point *Method of Providing Data* included by the use case *Provide System User Data* (i.e., (*B*6, *B*7)), we have one updated diagram decision in which the unselected use case *Provide System User Data via Diagnostic Mode* is selected. The corresponding use case specification is added (Lines 24–29 in Table 6). A new specific alternative flow is also generated for the inclusion of the newly selected use case in the specification of the use case *Provide System User Data* (Lines 12–15, red-colored).

The specification decision changes are about selecting optional alternative flows, optional steps and steps with a variant order (e.g., the triples (*B11, B12, B14*), (*B11, B12, B15*), (*B11, B12, B16*),

```
Input: Initial decision model, $M1$, New decision model, $M2$
Output: Triple of sets of decision-level changes
        (ADD, DELETE, UPDATE)
1.  Let a pair $(vp, uc)$ denote cases where $vp$ is
     a variation point and $uc$ is a use case including $vp$
2.  Let a pair $(uc, fl)$ denote cases where $uc$ is a use case
     and $fl$ is an optional alternative flow in $uc$
3.  Let a triple $(uc, fl, st)$ denote cases where $uc$ is
     a use case, $fl$ is a flow in $uc$, and $st$ is a step in $fl$
4.  Let $U1$ and $U2$ be the sets of $(vp, uc)$ in $M1$ and $M2$
5.  Let $F1$ and $F2$ be the sets of $(uc, fl)$ in $M1$ and $M2$
6.  Let $S1$ and $S2$ be the sets of $(uc, fl, st)$ in $M1$ and $M2$
7.  $U3 \leftarrow$ matchDiagramDecisions($U1$, $U2$)
8.  $F3 \leftarrow$ matchFlowDecisions($F1$, $F2$)
9.  $S3 \leftarrow$ matchStepDecisions($S1$, $S2$)
10. $DELETE \leftarrow (U1 \setminus U3) \cup (F1 \setminus F3) \cup (S1 \setminus S3)$
11. $ADD \leftarrow (U2 \setminus U3) \cup (F2 \setminus F3) \cup (S2 \setminus S3)$
12. foreach $(k \in (U3 \cap U1))$ do
13.     $z \leftarrow$ getMatchingDecision($k$, $U3$)
14.     $SUC1 \leftarrow$ getSelectedUseCases($k$, $M1$)
15.     $SUC2 \leftarrow$ getSelectedUseCases($z$, $M2$)
16.     if $(SUC1 \neq SUC2)$ then
17.         $UPDATE \leftarrow UPDATE \cup \{k\}$;
18.     end if
19. end foreach
20. foreach $(t \in (F3 \cap F1))$ do
21.     $y \leftarrow$ getMatchingDecision($t$, $F3$)
22.     if $(t.fl.isSelected \neq y.fl.isSelected)$ then
23.         $UPDATE \leftarrow UPDATE \cup \{t\}$
24.     end if
25. end foreach
26. foreach $(u \in (S3 \cap S1))$ do
27.     $m \leftarrow$ getMatchingDecision($u$, $S3$)
28.     if $(u.st$ is $OptionalStep)$ and
        $(u.st.isSelected \neq m.st.isSelected)$ then
29.         $UPDATE \leftarrow UPDATE \cup \{u\}$
30.     else
31.         if $(u.st.orderNumber \neq m.st.orderNumber)$
32.             then $UPDATE \leftarrow UPDATE \cup \{u\}$
33.         end if
34.     end if
35. end foreach
36. return $(ADD, DELETE, UPDATE)$
```

Fig. 13. Algorithm for Steps 1 and 2 in Fig. 11.

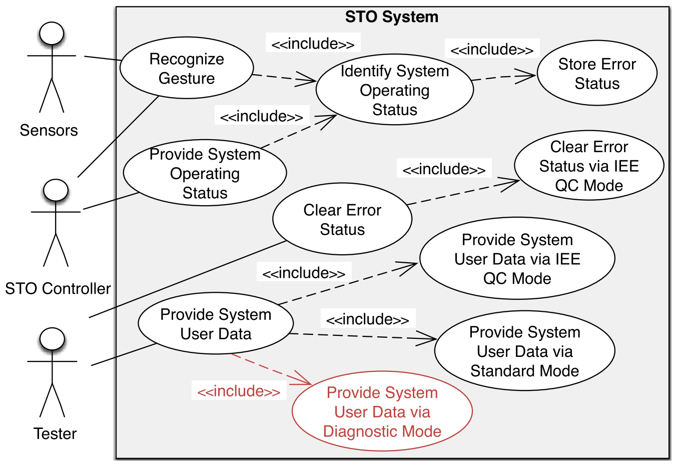

STO System
Sensors
STO Controller
Tester
Recognize Gesture
<<include>>
Identify System Operating Status
<<include>>
Store Error Status
<<include>>
Provide System Operating Status
Clear Error Status via IEE QC Mode
<<include>>
Clear Error Status
Provide System User Data via IEE QC Mode
<<include>>
Provide System User Data
<<include>>
Provide System User Data via Standard Mode
<<include>>
Provide System User Data via Diagnostic Mode

Fig. 14. Regenerated product specific use case diagram. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 6
Reconfigured Product Specific Specifications.

1	**USE CASE** Provide System User Data
2	**1.1 Basic Flow**
3	1. The tester SENDS the user data request TO the system.
4	2. The system VALIDATES THAT ‘Precondition of Provide System User Data via Standard Mode’.
5	3. INCLUDE USE CASE Provide System User Data via Standard Mode.
6	**1.2 Specific Alternative Flow**
7	RFS 2
8	1. IF ‘Precondition of Provide System User Data via IEE QC Mode’ holds THEN
9	2. INCLUDE Provide System User Data via IEE QC Mode.
10	3. ABORT.
11	4. ENDIF
12	**1.3 Specific Alternative Flow**
13	RFS 2
14	1. INCLUDE USE CASE Provide System User Data via Diagnostic Mode.
15	2. ABORT.
16	
17	**USE CASE** Provide System User Data via Standard Mode
18	**1.1 Basic Flow**
	~~1. The system SENDS trace data TO the tester.~~
19	1. The system SENDS sensor data TO the tester.
20	2. The system SENDS calibration TO the tester.
21	3. The system SENDS error data TO the tester.
22	4. The system SENDS error trace data TO the tester.
23	
24	**USE CASE** Provide System User Data via Diagnostic Mode
25	**1.1 Basic Flow**
26	1. The system SENDS the RAM data TO the tester.
27	2. The system SENDS the NVM data TO the tester.
28	3. The system SENDS the session response TO the tester.
29	4. The system SENDS the message length TO the tester.

and (*B11, B12, B17*) in Fig. 12(b)). The use case *Provide System User Data via Standard Mode* has two new steps in Lines 19 and 21 in Table 6 (i.e., (*B11, B12, B14*), and (*B11, B12, B16*) in Fig. 12(b)), while one of the steps (red-colored, strikethrough step) is removed (i.e., (*B11, B12, B15*) in Fig. 12(b)). The step number of one of the steps is changed (Line 22, blue-colored) due to the change in the order of the steps with a variant order (i.e., (*B11, B12, B17*) in Fig. 12(b)).

7. Tool support

We have implemented our change impact analysis approach as an extension of PUMConf (Product line Use case Model Configurator) (Hajri et al., 2016b). PUMConf has been developed as an IBM DOORS Plug-in. Section 7.1 provides the layered architecture of the tool while we describe the tool features with some screenshots in Section 7.2. For more details and accessing the tool, see: https://sites.google.com/site/pumconf/.

7.1. Tool architecture

Fig. 15 shows the tool architecture. It is composed of three layers (Hajri et al., 2016a): (i) the *User Interface (UI) layer*, (ii) the *Application layer*, and (iii) the *Data layer*.

We briefly introduce each layer and explain the new and extended components, i.e., the gray boxes in Fig. 15.

User Interface (UI) layer. This layer supports creating and viewing PL and PS artifacts, i.e., use case diagrams and specifications. We employ IBM Doors (http://www.ibm.com/software/products/ca/en/ratidoor/) for use case specifications and Papyrus (https://www.eclipse.org/papyrus/) for use case diagrams.

Application layer. With the new and extended components, this layer supports the main activities of our impact analysis approach in Fig. 4: *proposing a change, identifying the change impact on other decisions, applying the proposed change*, and *regenerating PS use case models*.

The *Configurator* component coordinates the other components in the application layer. The *Artifact Consistency Checker* and *Deci-*

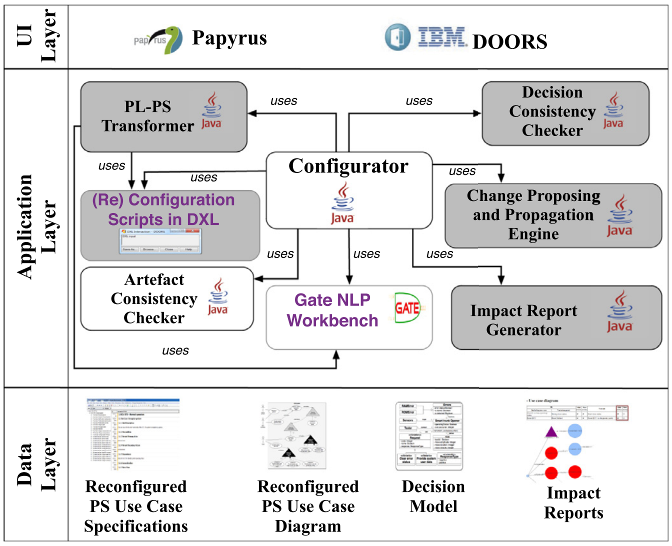

UI Layer
Papyrus
IBM DOORS
Application Layer
PL-PS Transformer
Decision Consistency Checker
Configurator
(Re) Configuration Scripts in DXL
Change Proposing and Propagation Engine
Artefact Consistency Checker
Gate NLP Workbench
Impact Report Generator
uses
Java
Data Layer
Reconfigured PS Use Case Specifications
Reconfigured PS Use Case Diagram
Decision Model
Impact Reports

Fig. 15. Layered architecture of PUMConf.

sion Consistency Checker components were introduced in previous work (Hajri et al., 2016a). The *Artifact Consistency Checker* employs Natural Language Processing (NLP) to check the consistency of the PL use case diagram and the PL use case specifications complying with the RUCM template. To perform NLP, our tool employs the GATE workbench (http://gate.ac.uk/), an open source NLP framework. The *Decision Consistency Checker* is extended to support inferring decision restrictions and checking their consistency as part of our impact analysis approach. The *PL-PS Transformer* component annotates the use case specifications using NLP to automatically generate PS use case specifications. It is extended with the pipeline in Fig. 11 to incrementally regenerate PS models. It uses scripts written in the Doors eXtension Language (DXL) to automatically (re)configure PS use case specifications. The DXL scripts are also used to load the (re)configured use case specifications into Doors.

We further implemented some new components: *Change Proposing and Propagation Engine* and *Impact Report Generator*. The *Change Proposing and Propagation Engine* supports proposing a decision change and applying the proposed change while the *Impact Report Generator* generates the impact analysis reports.

Data layer. The PL and PS use case specifications are stored in the native IBM DOORS format while the PL and PS use case diagrams are stored as UML models. The decision models are saved in Ecore (Eclipse EMF, 0000). We generate the impact reports as html pages.

7.2. Tool features

We describe the main features of our tool: *proposing a decision change, identifying the change impact on other decisions, applying the proposed change*, and *incrementally reconfiguring PS use case models.*

Proposing a change. This feature supports Step 1, *Propose a Change for a Decision*, in Fig. 4. Before applying the change, the analyst proposes the decision change to determine the change impact on other diagram decisions. In Fig. 16, the analyst decides to change the decision for the variation point *VP4* (Fig. 16(a)) and proposes selecting the unselected use case *UC10* (Fig. 16(b)).

Identifying the change impact on other decisions. For Step 2, *Identify the Change Impact on Other Decisions*, in Fig. 4, the tool automatically identifies the impact of the diagram decision changes on prior and subsequent diagram decisions. Once the analyst proposes the change, the tool provides an impact report documenting the impacted decisions along with an explanation for such impact.

Fig. 17 shows the impact report for the example change in Fig. 5, i.e., selecting the unselected *UC10* in *VP4*. We use various colors, with a legend, on variant use cases and variation points to explain the impacted decisions with the reason of the impact. When the analyst selects the unselected *UC10*, *UC12* and *UC13* are automatically selected (i.e., the orange variant use cases in Fig. 17(b)). The prior decision for *VP6* is impacted because *UC15* that is unselected is required by *UC13* which was selected after the change (i.e., the green yellow variant use case in Fig. 17(b)). The subsequent decisions for *VP3* and *VP7* are impacted because *UC8* in *VP3* and *UC14* in *VP7* are restricted by the changed decision (i.e., the red variant use cases in Fig. 17(b)). The prior decision for *VP1* is yet another impacted decision because of the cardinality constraint in *VP3* (i.e., the violet cardinality constraint in Fig. 17(b)). The cardinality constraint can no longer be satisfied with the restriction for *UC8* derived from the changed decision (i.e., the red *UC8* in Fig. 17(b)) and with the restriction for *UC6* derived from the prior decision for *VP1* (i.e., the cyan *UC6* in Fig. 17(b)).

Applying the proposed change. This feature supports Step 3, *Apply the Proposed Change*, in Fig. 4. After evaluating the impact of the proposed change, the analyst decides whether to apply the proposed change on the corresponding decision.

Incrementally reconfiguring PS use case models. This feature supports Step 4, *Regenerate Product Specific Use Case Models*, in Fig. 4. Once all the required changes are made, the tool automatically and incrementally regenerates the PS models corresponding to the changed decisions.

8. Evaluation

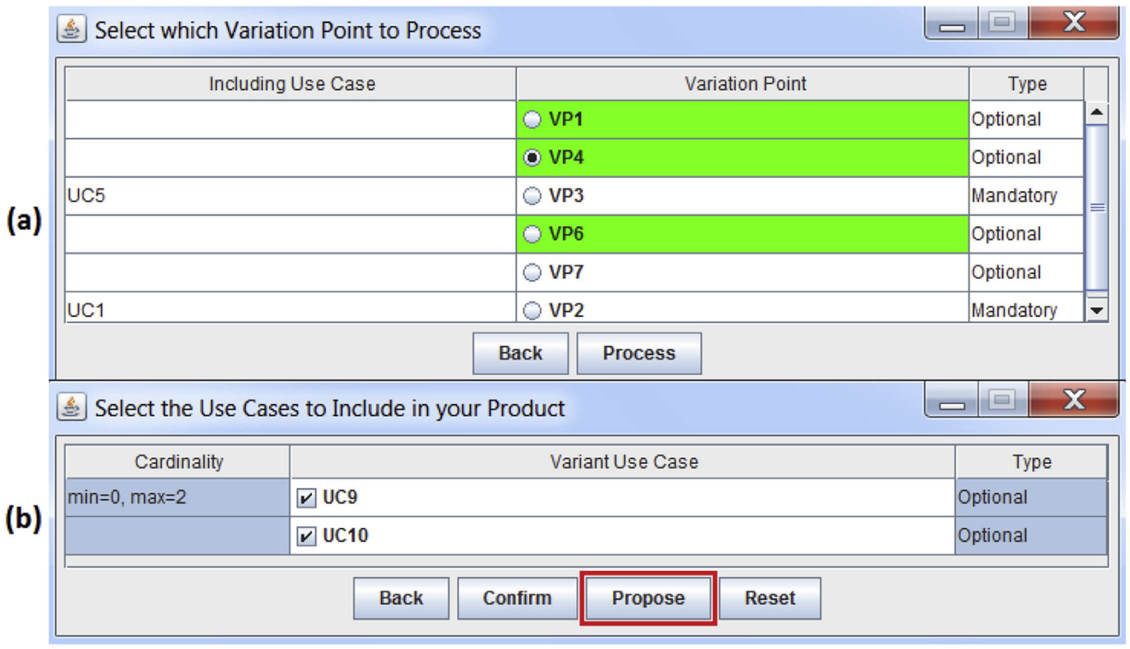
In this section, we evaluate our change impact analysis approach via reporting on (i) the results of a questionnaire survey at IEE aiming at investigating how the approach is perceived to address the challenges listed in Section 3 (Section 8.1), (ii) discussions with the IEE engineers to gather qualitative insights into the benefits and challenges of applying the approach in an industrial setting (Section 8.2), and (iii) an industrial case study, i.e., STO, to demonstrate the feasibility of the incremental reconfiguration of PS use case models (Section 8.3) for a representative system.

8.1. Questionnaire study

We conducted a questionnaire study to evaluate, based on the viewpoints of experienced IEE engineers, how well our change impact analysis approach addresses the challenges that we reported in Section 3. The study is described and reported according to the template provided by Wohlin et al. (2012).

8.1.1. Planning and design

To evaluate the output of our impact analysis approach in light of the challenges we identified earlier, we had semi-structured interviews with seven participants holding various roles at IEE: software development manager, software team leader, software engineer, system engineer, hardware development engineer, and embedded software engineer. They all had substantial industry experience, ranging from seven to thirty years. All participants, except the hardware development engineer, had previous experience with use case-driven development and modeling. The interview was preceded by presentations illustrating the background approaches (i.e., the PL use case modeling method (Hajri et al., 2015) and the use case-driven configuration approach (Hajri et al., 2016a)), our change impact analysis approach, a tool demo, and some detailed examples from STO. Interactive training sessions also took place which included questions posed to the participants about the example models and ensured that participants had reached a minimal level of understanding. We then organized three hands-on



(a)

Select which Variation Point to Process

Including Use Case	Variation Point	Type
	○ VP1	Optional
	● VP4	Optional
UC5	○ VP3	Mandatory
	○ VP6	Optional
	○ VP7	Optional
UC1	○ VP2	Mandatory

Back Process

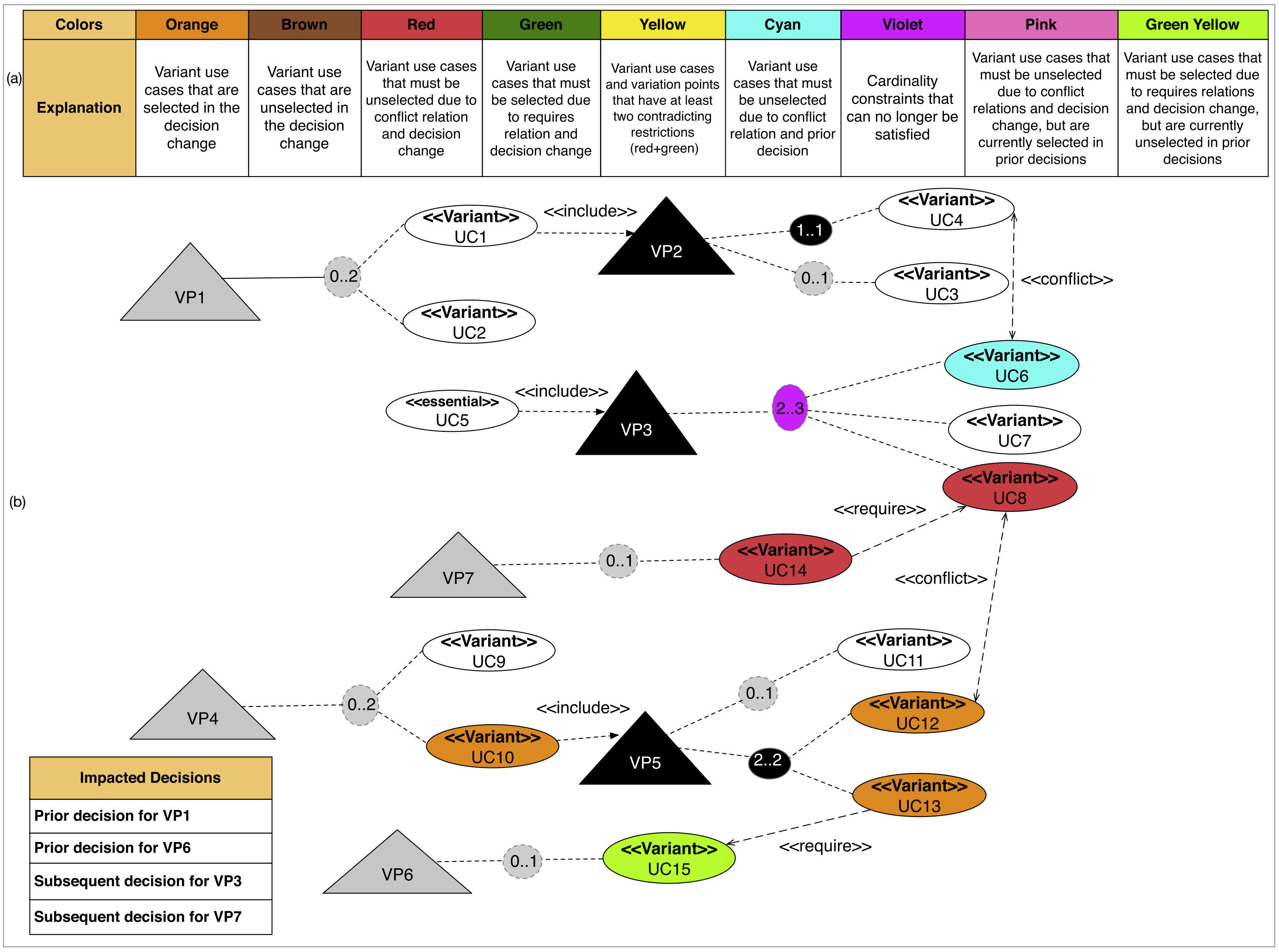
(b)

Select the Use Cases to Include in your Product

Cardinality	Variant Use Case	Type
min=0, max=2	✓ UC9	Optional
	✓ UC10	Optional

Back Confirm Propose Reset

Fig. 16. PUMConf's user interface for proposing a diagram decision change.



(a)

Colors	Orange	Brown	Red	Green	Yellow	Cyan	Violet	Pink	Green Yellow
Explanation	Variant use cases that are selected in the decision change	Variant use cases that are unselected in the decision change	Variant use cases that must be unselected due to conflict relation and decision change	Variant use cases that must be selected due to requires relation and decision change	Variant use cases and variation points that have at least two contradicting restrictions (red+green)	Variant use cases that must be unselected due to conflict relation and prior decision	Cardinality constraints that can no longer be satisfied	Variant use cases that must be unselected due to conflict relations and decision change, but are currently selected in prior decisions	Variant use cases that must be selected due to requires relations and decision change, but are currently unselected in prior decisions

(b)

Impacted Decisions
Prior decision for VP1
Prior decision for VP6
Subsequent decision for VP3
Subsequent decision for VP7

Fig. 17. PUMConf's user interface for displaying the change impact of diagram decision changes on other diagram decisions. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

sessions in which the participants could apply the configuration and the change impact analysis approaches in a realistic setting, followed by the structured interviews and data collection. In the first hands-on session, the participants were asked to make configuration decisions and resolve conflicting decisions using the guidance provided by PUMConf to generate PS use case models from the sample PL use case diagram and specifications. In the second hands-on session, they used the impact analysis results provided by PUMConf to identify the impact of the proposed decision changes on prior and subsequent decisions in PL use case diagrams. In the third session, the participants used PUMConf to incrementally reconfigure PS use case models based on the changed decisions.

To capture the perception of the IEE engineers participating in the interviews, regarding the potential benefits of our impact analysis approach and how it addresses the targeted challenges, we handed out two questionnaires including questions to be answered according to two Likert scales (Oppenheim, 2005) (i.e., agreement and probability). The questionnaires were structured for the participants to assess both our configurator and our change impact analysis approach in terms of adoption effort, correctness, comparison with current practice, and tool support. The participants were also encouraged to provide open, written comments.

8.1.2. Results and analysis

We solicited the opinions of the participants using two questionnaires named *QA* and *QB* (see Figs. 18 and 19). The objective of the questionnaire *QA* was to evaluate our use case-driven configuration approach and its tool support. We needed to know how well the participants understood and assessed the configuration approach before receiving their feedback about our impact analysis approach, which builds on it. Fig. 18(a) and (b) depict the questions in *QA* and the participants' answers. The questions of *QA* were divided into three parts: (1) configuration of PS use case diagrams (*QA*1, *QA*2 and *QA*3), (2) configuration of PS use case specifications (*QA*4 and *QA*5), and (3) the overall configuration approach and its tool support (from *QA*6 to *QA*11).

All participants, except two, agreed that our configurator is adequate and practical to capture configuration decisions for PS use case models (*QA*1 and *QA*4). Further, these participants expressed their willingness to use our tool for automatically configuring PS models in their projects (*QA*2 and *QA*5). The two participants who did not *agree* on *QA*4 stated that they need to gather more experience on various product line projects to be able to provide a precise judgment about the configurator. We note that one of those participants *disagreed* whereas the second one left the questions (from *QA*1 to *QA*6) unanswered. The former was the HW engineer, with no initial use case modeling experience, and the latter was the system engineer. In short, these two participants were the ones with the least software background.

Regarding the questions that target the overall approach and its tool support (from *QA*6 to *QA*11), the participants agreed that the effort required to learn and apply our configurator is reasonable (*QA*7). Nevertheless, one participant stated that more training is required to be able to easily follow the configuration steps (*QA*6). All participants except one were interested in using our configurator for managing product lines. The remaining participant, who is a software project manager and was the most experienced, thought that our configurator brings added value only for projects which include significant variability information, e.g., projects with more than 50 variation points (*QA*8). Moreover, the participants agreed that our configurator provides useful assistance for configuring PS use case models, when compared to the current practice in their projects (*QA*10), and ease communication between analysts and stakeholders during configuration (*QA*9).

The objective of the second questionnaire *QB* was to evaluate our change impact analysis approach. Fig. 19(a) and (b) depicts the questions and answers for *QB*. *QB* is structured in four parts: (1) identifying the impact of decision changes on other diagram decisions (from *QB*1 to *QB*3), (2) incrementally reconfiguring PS use case diagrams (*QB*4 and *QB*5), (3) incrementally reconfiguring PS use case specifications (*QB*6 and *QB*7), and (4) the overall impact analysis approach and its tool support (from *QB*8 to *QB*14).

All participants, except one, agreed that (1) our approach is sufficient to determine and explain the impact of decision changes for PL use case diagrams (*QB*1) and (2) the impact report generated after the incremental reconfiguration is sufficient to capture the changed parts of the PS use case diagram (*QB*4). The participant who disagreed on *QB*1 and *QB*4 mentioned in his comments that he lacks experience in use case-driven development and modeling and that he is not sufficiently familiar with the tool to provide a precise answer. There was a strong consensus among participants about the value of adopting our change impact analysis approach (*QB*10 and *QB*11) and about the benefits of using it to identify the impact of decision changes and to reconfigure PS models in their projects (*QB*5 and *QB*7). The participants were very positive about the approach in general and were enthusiastic about its capabilities, and most particularly the impact analysis reports provided by the tool. Nevertheless, they mentioned that the user interface needed to be more professional and ergonomic, which was not surprising for a research prototype. This was the main reason for one of the participants to disagree on *QB*3, *QB*12, and *QB*14.

8.2. Discussions with the analysts and engineers

The questionnaire study had open, written comments under each section, in which the participants could state their opinions in a few sentences about how our impact analysis approach addresses the challenges reported in Section 3. As reported in Section 8.1, the participants' answers to the questions through Likert scales and their open comments indicate that they see high value in adopting the change impact analysis approach and its tool support in an industrial setting in terms of (1) improving decision making process, (2) increasing reuse, and (3) reducing manual effort during reconfiguration. In order to elaborate over the open comments in the two questionnaires, we organized further discussions with the participants. Based on the initial comments, we identified two aspects to further discuss with the participants: industrial adoption of the approach and its limitations.

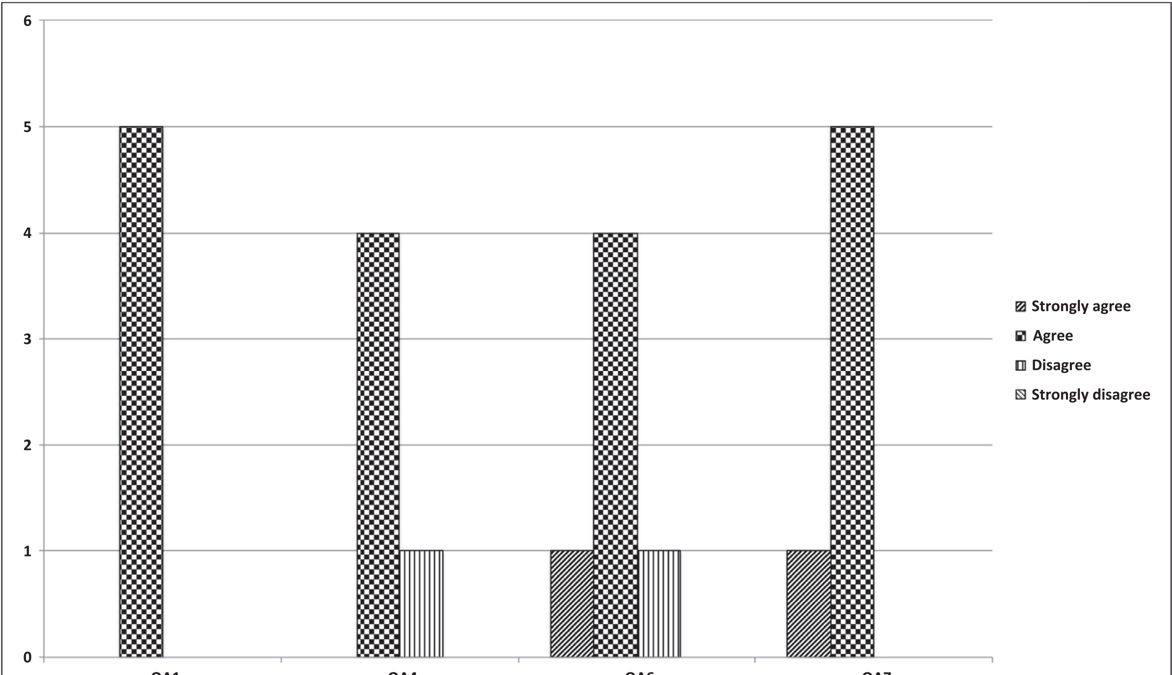
8.2.1. Industrial adoption of the approach

Our impact analysis approach is devised to support the decision-making process in the context of use case-driven configuration. Therefore, it needs to be adopted as part of our configuration approach. In the current practice at IEE, like many other environments, there is no systematic way to (re)configure product-specific use case models and to identify the change impact for evolving decisions for use case models. Although IEE engineers consider that the effort required to learn and apply our configuration and change impact analysis approach is reasonable, they also stated that the costs and benefits of adopting it should be further evaluated. This is, however, a common and general challenge when introducing new practices in software development.

8.2.2. Limitations of the approach

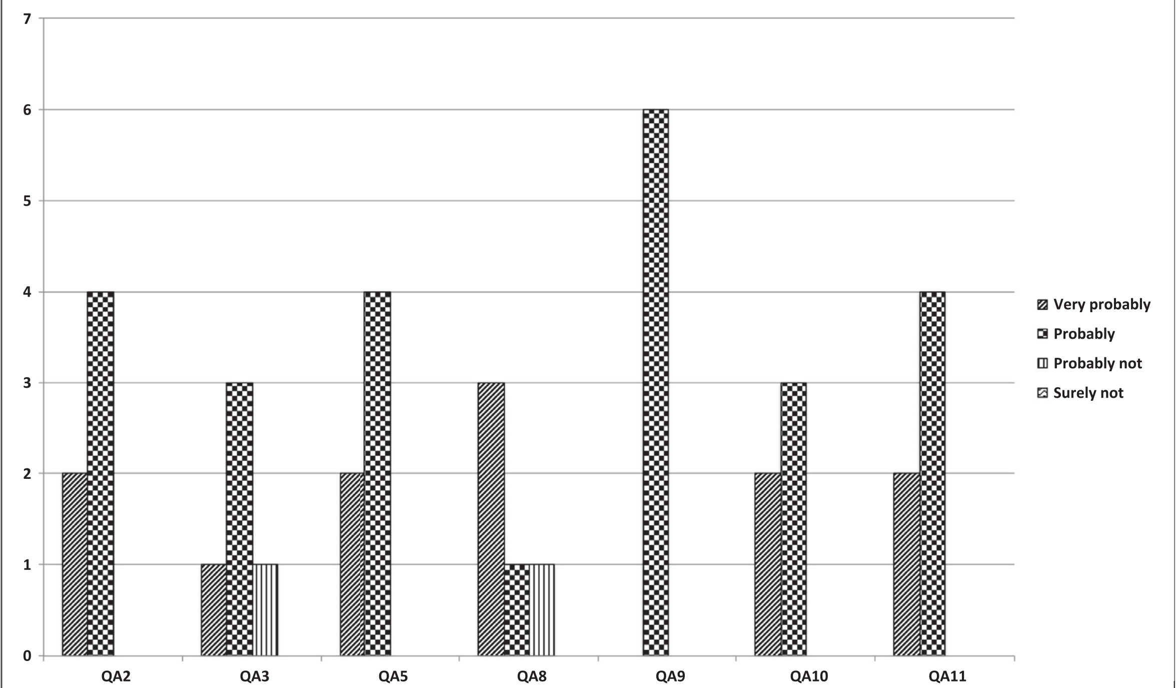
Our change impact analysis approach and its tool support currently have some limitations. First, our approach supports only evolving configuration decisions. However, changes may also occur on variability aspects of PL use case models. For instance, we may introduce a new variation point in the PL use case diagram or we can remove a variant use case for a given variation point. As stated

(a)



QA1. The decision-making in the configurator is sufficient to capture configuration decisions for product specific use case diagram.
QA4. The decision-making in the tool is sufficient to capture configuration decisions for product specific use case specifications.
QA6. The steps in our configuration method are easy to follow, given appropriate training.
QA7. The effort required to learn how to apply the configuration method is reasonable.

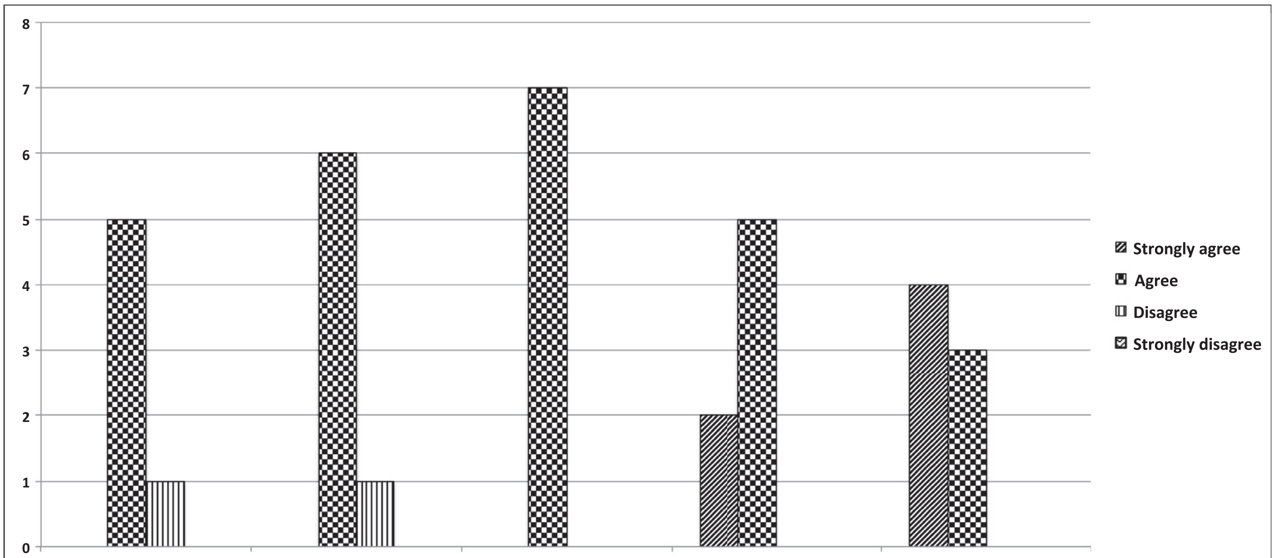
(b)



QA2. If the configurator like the one we presented were available to you, would you use it to configure product specific use case diagram in your projects?
QA3. Do you think that the presented tool provides useful assistance for identifying and resolving the inconsistent decisions in PL use case diagrams?
QA5. If the configurator like the one we presented were available to you, would you use it to configure product specific use case specifications in your projects?
QA8. Would you see value in adopting the presented method for configuring product specific use case models?
QA9. Does the presented method provide useful assistance for easing the communication between analysts and stakeholders during configuration?
QA10. Does the presented method provide useful assistance for configuring product specific use case models compared to the current practice in your projects?
QA11. Do you think that the presented tool provide useful automation for generating product specific use case and domain models?

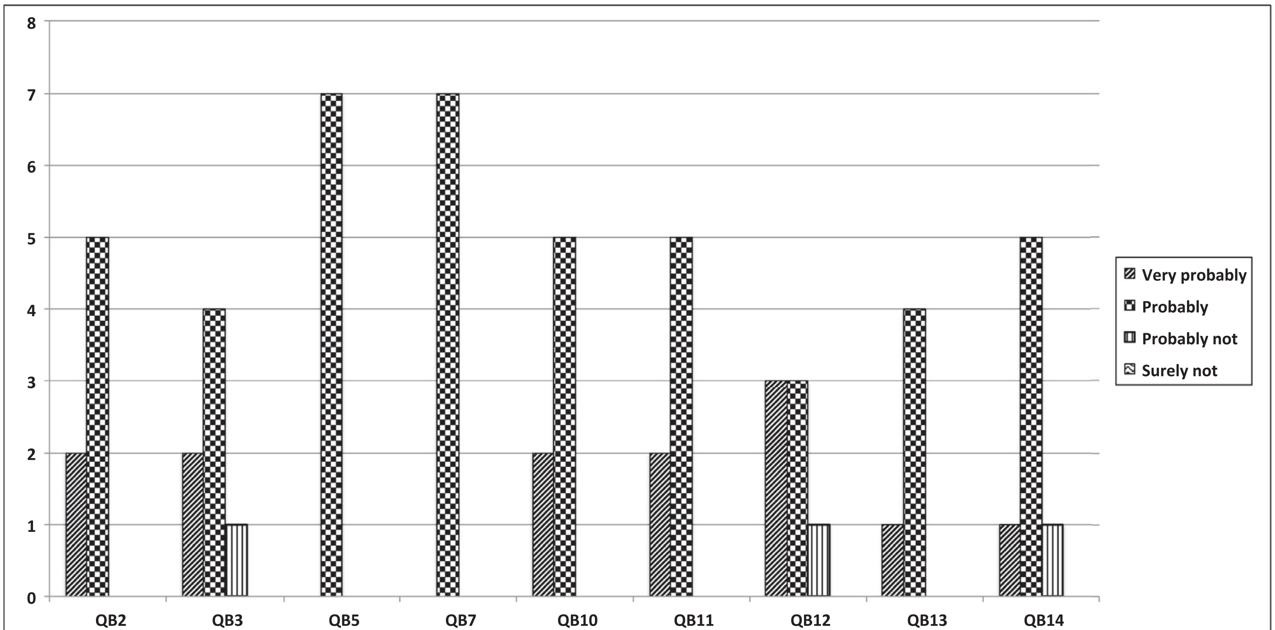
Fig. 18. Responses to the questions related to the configuration approach.

(a)



QB1. The approach is sufficient to determine the impact of decision changes for product line use case diagrams.
QB4. The impact report generated after the incremental reconfiguration is sufficient to capture the changed parts of the product specific use case diagram.
QB6. The impact report generated after the incremental reconfiguration is sufficient to capture the changed parts of the product specific use case specifications.
QB8. The steps in our change impact analysis method are easy to follow, given appropriate training.
QB9. The effort required to learn how to apply the change impact analysis method is reasonable.

(b)



QB2. If the approach like the one we presented were available to you, would you use it to determine the impact of decision changes in PL use case diagrams in your projects?
QB3. Do you think that the presented tool provides useful assistance for identifying the impact of decision changes in product line use case diagrams?
QB5. If the incremental reconfiguration approach like the one we presented were available to you, would you use it to reconfigure product specific use case diagrams in your projects?
QB7. If the incremental reconfiguration approach like the one we presented were available to you, would you use it to reconfigure product specific use case specifications in your projects?
QB10. Would you see value in adopting the presented method for identifying the impact of decision changes on other diagram decisions?
QB11. Would you see value in adopting the presented method for incrementally reconfiguring product specific use case models?
QB12. Does the presented method provide useful assistance for easing the communication between analysts and stakeholders during changing configuration decisions?
QB13. Does the presented method provide useful assistance for incrementally reconfiguring product specific use case models compared to the current practice in your projects?
QB14. Do you think that the presented tool provide useful automation for change impact analysis for evolving configuration decisions?

Fig. 19. Responses to the questions related to the change impact analysis approach.

Table 7
Product line use cases in the case study.

	# of use cases	# of variation points	# of basic flows	# of alternative flows	# of steps	# of condition steps
Essential UCs	11	6	11	57	192	57
Variant UCs	13	1	13	131	417	130
Total	24	7	24	188	609	187

by IEE engineers, it is important to evaluate the impact of PL use case model changes on configuration decisions and on PS use case models. Therefore, our approach needs to be extended for evolving PL use case models. Second, we implemented our approach as part of a prototype tool, PUMConf. The tool has already received positive feedback from IEE engineers but they stated that it needs further improvements in terms of usability. To identify potential usability improvements, we decided to conduct empirical and heuristic evaluations (Nielsen and Molich, 1990; Nielsen, 1994). With regards to the empirical evaluation, we plan to perform a user study with IEE engineers, where we will record the end user interaction with the configurator. We plan to perform the heuristic evaluation of the user interfaces according to certain rules, such as those listed in typical guideline documents (Smith and Mosier, 1986), by asking users' opinions about possible improvements of PUMConf's user interfaces.

8.3. Industrial case study

We report our findings about the feasibility of part of our impact analysis approach, i.e., incremental reconfiguration of PS use case models, and its tool support in an industrial context. In order to experiment with our incremental reconfiguration approach in an industrial project, we applied it to the functional requirements of STO.

8.3.1. Goal

Our goal was to assess, in an industrial context, the feasibility of using our approach. We assessed whether we could improve reuse and significantly reduce manual effort by preserving unimpacted parts of PS use case models, when possible, and their manually assigned traces.

8.3.2. Study context

STO was selected for the assessment of our approach since it was a relatively new project at IEE with multiple potential customers requiring different features. IEE provided their initial STO documentation, which contained a use case diagram, use case specifications, and supplementary requirements specifications describing non-functional requirements. To model the STO requirements according to our modeling method, PUM, we first examined the initial STO documentation and then worked with IEE engineers to build and iteratively refine our models (Hajri et al., 2015) (see Table 7). Due to confidentiality concerns, we do not put the entire case study online. However, the reader can download the sanitized example models from the tool's website (https://sites.google.com/site/pumconf/download-installation/).

8.3.3. Results and analysis

By using PUMConf, we, together with the IEE engineers, configured the PS use case models for four products selected among the STO products IEE had already developed (Hajri et al., 2016a). The IEE engineers made decisions on the PL models using the guidance provided by PUMConf. Among the four products, we chose one product to be used for reconfiguration of PS models (see Table 8) because it was the most recent one in the STO product family with a properly documented change history. The IEE engineers identified 36 traces from the PS use case diagram and 278 traces from the PS use case specifications that were directed to other software and hardware specifications as well as to the customers' requirements documents for external systems (see Fig. 3 for an example trace). We considered eight change scenarios derived from the change history of the initial STO documentation for the selected product (see Table 9).

Some change scenarios contain individual decision changes such as selecting unselected use cases in a variation point, while some others contain a series of individual changes to be applied sequentially (see *S2* and *S4*). For instance, *S2* starts with unselecting *Clear Error Status* in Fig. 1, which automatically deletes the decision for the variation point *Method of Clearing Error Status* and implies another decision change, i.e., unselecting *Store Error Status*.

Table 10 provides a summary of the reconfiguration of the PS use case models for the change scenarios. After each change scenario, we ran PUMConf and checked the preserved and deleted traces. As discussed, our approach preserves all the traces for the unchanged parts of the PS models, while removing the traces for the deleted parts of the PS models, which must be manually updated. To assess the savings in traceability effort while reconfiguring, we looked at the percentages of traces from the use case diagram and the use case specifications that were preserved over all change scenarios. From Table 10, we can see that between 73% and 100% (average ≈ 96%) of the use case diagram traces were preserved. Similarly, for the use case specifications, trace reuse was between 82% and 100% (average ≈ 96%). We can therefore conclude that the proposed incremental reconfiguration of PS use case models leads to significant savings in traceability effort in the context of actual configuration decision changes.

8.4. Threats to validity

The main threat to validity in our evaluation concerns the generalizability of the conclusions we derived from our industrial case study and from the participants' answers in our questionnaire study. To mitigate this threat, we applied our approach to an industrial case study, i.e., STO from our industry partner, that includes nontrivial use cases in an application domain with many potential customers and sources of variability. STO is a relatively simple but typical automotive embedded system. It can be reasonably argued that more complex systems would require more configuration support, not less. Further case studies are nevertheless necessary for improving external validity. The fact that the respondents to our questionnaire were selected to have diverse backgrounds and the consistency observed across the answers we received provide confidence about the generalizability of our conclusions among different project participants. A potential threat to internal validity is that the we have limited domain knowledge and were involved in the modeling and (re)configuration of the use case models we used in our evaluation. To minimize the risks of mistakes, we had many meetings and interviews with domain experts at IEE to verify the correctness and completeness of (1) our PL use case models, (2) the STO configurations, and (3) the output of our change impact analysis approach.

9. Related work

We cover the related work across four categories.

Table 8
Configuration results for the selected product.

Product	# of Selected Variant Use Cases	# of Selected Optional Steps	# of Selected Optional Flows	# of Decided Variant Order
P1	6	1	0	0

Table 9
Decision change scenarios.

ID	Change Scenario	Explanation
S1	Update a diagram decision	Unselecting selected use cases
S2	Update and delete diagram decisions	Unselecting selected use cases, removing other decisions
S3	Update a diagram decision	Selecting unselected use cases
S4	Update and add diagram decisions	Selecting unselected use cases, implying other decisions
S5	Update a specification decision	Selecting unselected optional steps
S6	Update a diagram decision	Selecting unselected use cases
S7	Update a diagram decision	Unselecting selected use cases
S8	Update a specification decision	Updating the order of optional steps

Table 10
Summary of the Reconfiguration of the PS Use Case Models for STO.

		Decision Change Scenarios							
		S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**
PS Model Changes	**# of Added UCs**	0	0	1	4	0	1	0	0
	# of Deleted UCs	1	4	0	0	0	0	1	0
	# of Added UC Steps	0	0	53	140	3	85	0	0
	# of Deleted UC Steps	53	140	0	0	0	0	103	0
Traces for the PS Use Case Diagram	**# of Initial Traces**	36	34	25	27	36	36	38	38
	# of Deleted Traces During Reconfiguration	2	9	0	0	0	0	0	0
	# of Manually Added Traces After Reconfiguration	0	0	2	9	0	2	0	0
	# of Preserved Traces	34	25	25	27	36	36	38	38
	% of Preserved Traces	94.4	73.5	100	100	100	100	100	100
Traces for the PS Use Case Specifications	**# of Initial Traces**	278	265	218	231	278	287	298	278
	# of Deleted Traces During Reconfiguration	13	47	0	0	0	0	20	0
	# of Manually Added Traces After Reconfiguration	0	0	13	47	9	11	0	0
	# of Preserved Traces	265	218	218	231	278	287	278	278
	% of Preserved Traces	95.3	82.2	100	100	100	100	93.2	100

Reasoning approaches for product lines. PL use case diagrams and feature models have similar modeling constructs to represent system variability in terms of variation points, variant cardinalities and dependencies. In a literature review on automated analysis of feature models (Benavides et al., 2010), three types of analysis operations on feature models are addressed: *corrective explanations, dependency analysis* and *valid partial configuration*. Our change impact analysis approach relies on a form of *dependency analysis* to identify the impact of changing configuration decisions in PL use case diagrams (*Challenge 1*). The *dependency analysis* operation takes a variability model (i.e., a feature model) and a partial configuration as input and returns a new configuration with the variants (i.e., features) that should be selected and/or unselected as a result of the dependency constraints (Benavides et al., 2010). The FaMa formaL frAMEwork (FLAME) proposed by Durán et al. (2017) specifies the semantics of the analysis operations, e.g., validity of a product, the set of all valid products and validity of a configuration, which can be employed not only for feature models, but also for other variability modeling languages. However, in FLAME, change impact analysis has not been considered as an analysis operation with its semantics in the presence of evolving configuration decisions. By using dependency constraints, in the context of PL use case modeling, our approach identifies variant use cases that should be selected or unselected as a result of a configuration decision change.

Trinidad et al. (2008) and White et al. (2010, 2008) provide techniques to automatically propose decision changes when a dependency constraint is violated by some configuration decisions in a partial configuration. In contrast, our approach identifies (potential) violations of dependency constraints when the analyst proposes a configuration decision change. We can classify the automated support for the analysis operations according to the logic paradigm it relies on: *propositional logic* (Mannion, 2002; Mannion and Camara, 2003; Batory, 2005), *constraint programming* (Benavides et al., 2005b, 2005a; Karatas et al., 2010) and *description logic* (Wang et al., 2005, 2007; Fan and Zhang, 2006). Regarding propositional logic, a variability model is first mapped into a propositional formula in conjunctive normal form (CNF). A SAT solver takes the derived propositional formula and assumptions (configuration decisions) as input and determines if the formula is either satisfiable (SAT) or unsatisfiable (UNSAT). Techniques such as HUMUS (High-level Union of Minimal Unsatisfiable Sets) (Nöhrer et al., 2012; Nöhrer and Egyed, 2013) are used to identify the contradicting configuration decisions in the presence of UNSAT. Although we map the PL use case diagram into propositional logic formulas, we do not employ any SAT solving technique. Instead, for reasons explained below, we develop our own impact analysis algorithm in our use case-driven product line context (see Section 5). When a change is introduced to a diagram decision, our algorithm checks the consistency of decisions to identify the impact on prior and subsequent decisions. A decision change can violate dependency constraints with prior decisions or restrict subsequent decisions. One important point is that our algorithm identifies not only the impacted decisions but also the cause of the change impact. In practice, the cause of the change impact is important for the analysts to identify the further changes to be made on impacted decisions. In contrast, when using SAT solvers, we only obtain as output, without any further explanation, decisions contradicting each other after the decision change (Nöhrer et al., 2012; White et al., 2010). For instance, assume that the analyst unselects the selected variant use case *Store Error Status* while there is no decision made yet for the variation point *Clearing Error Status* in Fig. 1. Our approach identifies that the subsequent decision for *Clearing Error Status* is impacted because the decision restriction previously introduced through the *require* dependency becomes invalid after the change.

One advantage of SAT solvers is that they are a mature technology that is able to deal with large-scale models (Liang et al., 2015). In a SAT solver-based approach, given a PL use case diagram, one propositional formula can be formed as a conjunction of formulas derived from each dependency in the diagram using the propositional logic mapping in Fig. 7. Given such propositional formula and a set of variable assignments (decisions), a SAT solver can determine whether there is a value assignment to the remaining variables (undecided variation points) that will satisfy the predicate (Batory, 2005). In order to find out the decisions impacted by a decision change, it would be necessary to run SAT multiple times, since configuration decisions before and after changes lead to different variable assignments. In addition, every different impact may require different variable assignments and this might be computationally demanding. We follow a different solution that navigates the graph of dependencies and assigns values to boolean expressions while verifying conflicts. This is expected to be much less computationally demanding, especially when the size of the graph to navigate is small. Based on our observations in practice, PL use case diagrams remain relatively limited in size and rarely contain more than a few dozen variant use cases, cardinality constraints and dependencies. Scalability is therefore not a significant issue for our approach.

Impact analysis approaches for product lines. In the context of product line engineering, most of the approaches in the literature focus on the evolution of variability models instead of the evolution of configuration decisions (Botterweck and Pleuss, 2014). They predict the potential further changes in a PL model, e.g., a feature model, when deciding about a change in the same model. For instance, Thüm et al. (2009) present an algorithm to reason about feature model changes. The evolution of a feature model is classified as *refactoring* (i.e., no new products are added), *specialization* (i.e., no new products are added and some existing products removed), *generalization* (i.e., new products are added and no existing products removed), and *arbitrary edits*. The presented algorithm takes two versions of the same feature model as input and automatically computes the change classification. Alves et al. (2006) provide a catalog of change operations (e.g., *add new alternative feature* and *replace mandatory feature*) for refactoring feature models. Paskevicius et al. (2012) employ a similar catalog of change operations to propagate a feature model change to other feature model elements through feature dependencies such as *parent* and *child*. Because the approach proposed by Thüm et al. (2009) does not identify the change operations applied between two versions, Acher et al. (2012) build on it to identify the differences between feature models in terms of propositional formulas. It does so by comparing configuration spaces of the feature models. Bürdek et al. (2015) propose a model differencing approach, which is similar to our model differencing pipeline in Section 6, to determine and document complex change operations between the feature model versions (i.e., feature models before and after changes). Our model differencing pipeline identifies configuration decision changes, while their approach is used to determine changes between two feature models, not between two configurations.

Seidl et al. (2012) assume that there are mappings, provided by the analyst, from feature models to artifacts such as UML class diagrams and source code. They propose a classification of feature model changes that captures the impact of these changes on the feature model mappings and the mapped artifacts. Quinton et al. (2015) propose yet another approach to ensure consistency of feature models and their mapped artifacts when feature models evolve. Dintzner et al. (2014) compute the impact of a feature model change on the existing configurations of a product line by using partial dependency information in feature models. Similar to Dintzner et al. (2014), Heider et al. (2012b,a) propose another approach using regression testing to identify the impact of variability model changes on products. For a change in a variability model of a product line, the approach identifies whether configuration decisions for the existing products need to be changed as well. Then, it reconfigures all the products in the product line for the impacted decisions. The approach also compares the reconfigured products with the previous version to inform the analysts about the changed parts of the products.

One of the main differences between our approach and all the other approaches given above is that the latter mainly focus on changes on feature models, not changes on configuration decisions, while our approach deals with configuration decision changes and their impact on other decisions in PL use case models (*Challenge 1*). We incrementally reconfigure PS use case models as a result of evolving configuration decisions (*Challenge 2*) and do not address evolving PL models. White et al. (2014) propose an automated approach for deriving, on a feature model, a set of configurations that meet a series of requirements in a multi-step configuration process. It is assumed that an initial configuration evolves to a desired configuration where the analysts do not know the intermediate configuration steps which involve configuration decision changes requiring multiple steps. The approach does not identify the impact of decision changes but calculates subsequent decisions to derive potential configuration paths between the initial and desired configurations by mapping them to a Constraint Satisfaction Problem (CSP). In contrast, our approach identifies the impact of decision changes on subsequent and prior decisions to reach the desired configuration. It guides the analyst in addressing the cause of the impact of decision changes and ensures that a valid configuration is reached.

Another main difference is that our working context is specific to use case models with a specific product line modeling method, i.e., *PUM*, which explicitly models variability information in use case models, without any additional artifact such as feature models. The benefits of use case-driven configuration have been acknowledged and there are approaches proposed in the literature (Alves et al., 2010; Alférez et al., 2014; Rabiser et al., 2010). However, to the best of our knowledge, there is no work addressing the impact analysis of evolving configuration decisions in the context of use case-driven configuration. Many configuration approaches (Alférez et al., 2009; Zschaler et al., 2009; Czarnecki and Antkiewicz, 2005; Eriksson et al., 2009, 2004) require that feature models be traced as an orthogonal model to development artifacts such as UML use case, activity and class diagrams. Alternatively, we could have developed our impact analysis approach using feature models traced to use case models. If impacted decisions on feature models can be identified, and there are trace links between feature models and use cases, these trace links can be followed to identify impacted use cases. With such a solution, feature modeling needs to be introduced into practice, including establishing and maintaining traces between feature models and use case specifications and diagrams, as well as other artifacts. At IEE and in many other development environments, such additional modeling artifacts and the associated traceability are often perceived as unacceptable overhead and a practical hindrance due to the introduction and support of additional tools (Hajri et al., 2015). We do not claim that our approach is generally superior to an approach using feature modeling. However, our approach is likely to be preferred in certain contexts where development is driven by use case modeling.

Impact analysis approaches for requirements models. There are impact analysis approaches that address change propagation in requirements, but not specifically in a product line context. Goknil et al. (2014b, 2008a) and ten Hove et al. (2009) propose a change impact analysis approach which propagates changes in natural language requirements to other requirements by using the formal semantics of requirements relations, e.g., ‘requires’, ‘refines’ and ‘conflicts’ (Goknil et al., 2011, 2008b, 2013). These requirements relations are used together with trace links between requirements and architecture models to identify the impact of requirements changes on architecture models (Goknil et al., 2014a, 2016b, 2016a). When requirements are expressed in models such as goal models, more specialized dependency types can be used for impact analysis. For instance, Cleland-Huang et al. (2005) use soft goal dependencies to analyze how changes in functional requirements impact non-functional requirements, while Amyot (2003) uses operationalization dependencies between use cases and goals to propagate change between intentional and behavioral requirements. Arora et al. (2015a,b) propose another approach for impact analysis over Natural Language (NL) requirements by employing Natural Language Processing (NLP) techniques including the use of syntactic and semantic similarity measures. The approach uses similarity measures to compute the change impact in terms of relatedness between the changed requirement and other requirements in the requirements document. Nejati et al. (2016) extend the approach to propagate requirements changes to design models in SysML. Our work was inspired from the above techniques in terms of using requirements relations to propagate changes among diagram decisions (*Challenge 1*). Our approach does not address changes in natural language requirements, but deals with propagating decision changes to other decisions through variation point-variant use case dependencies in the context of use case-driven configuration.

Incremental model generation approaches. Use case-driven configuration approaches in the literature (e.g., Eriksson et al., 2005; Fantechi et al., 2004b; Czarnecki and Antkiewicz, 2005; Alférez et al., 2009) do not support incremental reconfiguration of use cases for evolving configuration decisions (*Challenge 2*). There are also more general configuration approaches (e.g., Dhungana et al., 2011; Rosa et al., 2009) that can be customized to configure PS use case models. For instance, DOPLER (Dhungana et al., 2011) supports capturing variability information as a variability model, and modeling any type of artifact as asset models. Variability and asset models are linked by using trace relations. The approach proposed by Heider et al. (2012a,b) is an extension of DOPLER to identify the impact of changes of variability information on products. It reconfigures all the products in the product line for the impacted decisions. However, it focuses on changes in variability information, not changes in configuration decisions. It is also not incremental, limiting its applicability, as the reconfiguration encompasses all the decisions, not only the affected ones.

Considerable attention in the model-driven engineering research community has been given to incremental model generation/transformation for model changes (e.g., Hearnden et al., 2006; Kurtev et al., 2007; Jahann and Egyed, 2004; Giese and Wagner, 2009; Xiong et al., 2007), and this line of work has inspired initiatives in many software engineering domains. For instance, Vogel et al. (2009) use incremental model transformation techniques for synchronizing runtime models by integrating a general-purpose model transformation engine into their runtime modeling environment. Epsilon (Kolovos et al., 2006a, 2008, 2006b) is a model-driven development suite with a model transformation language, which provides automated support for a number of additional tasks such as model differencing, merging, validation and model-to-text transformation. Alternatively, we could also have employed such a generic model transformation engine and language to implement the incremental generation of PS use case models. However, compared to model transformation languages, in terms of loading, matching and editing text in natural language, Java provides much more flexibility for handling plain text use case specifications. As a result, we used Java to implement the generation of PS use case models in our prior work (Hajri et al., 2016a), and also to implement the incremental reconfiguration of PS models as a model differencing and reconfiguration pipeline (see Section 6). To the best of our knowledge, our approach is the first to support incremental reconfiguration of PS use case models for evolving decisions in a product family.

10. Conclusion

This paper presents a change impact analysis approach that supports evolving configuration decisions in product-line (PL) use case models. It automatically identifies the impact of decision changes on other decisions in PL use case models, and incrementally reconfigures PS use case diagrams and specifications for evolving decisions.

We aimed to improve the decision making process by informing the analyst about the impact of decision changes and to minimize manual traceability effort by automatically but incrementally reconfiguring the PS use case models, that is to only modify the affected model parts given a decision change and thus preserve as many traceability links as possible to other artifacts.

Our change impact analysis approach is built on top of our previous work (i.e., Product line Use case Modeling method and the Product line Use case Model Configurator) and is supported by a tool integrated into IBM DOORS. The key characteristics of our tool are (1) the automated identification of the impact of decision changes and their associated causes on prior and subsequent decisions in PL use case models, and (2) the automated incremental reconfiguration of PS models from PL models and evolving configuration decisions. We performed a case study in the context of automotive domain. The results from structured interviews and a questionnaire study with experienced engineers suggest that our approach is practical and beneficial to analyze the impact of decision changes and to incrementally reconfigure PS models in industrial settings.

At this current stage, our approach does not support the evolution of PL use case models. We still need to address and manage changes in variability aspects of PL models such as adding a new variation point in the PL use case diagram. This work is an intermediate step to achieve our long term objective (Hajri, 2016; Hajri et al., 2017a), i.e., change impact analysis and regression test selection in the context of use case-driven development and testing. Our plan is to support change impact analysis for evolving PL use case models to help analysts properly manage changes in such models. Additionally, we would like to provide an automated regression test selection approach for system test cases derived from use case models in product families.

Acknowledgments

We gratefully acknowledge funding from FNR and IEE S.A. Luxembourg under grants FNR/P10/03 and FNR10045046, and from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 694277).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2018.02.021.

References

Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., Merle, P., 2012. Feature model differences. In: CAiSE'12, pp. 629–645.
Alférez, M., Bonifácio, R., Teixeira, L., Accioly, P., Kulesza, U., Moreira, A., Araújo, J., Borba, P., 2014. Evaluating scenario-based spl requirements approaches: the case for modularity, stability and expressiveness. Requirements Eng. J. 19, 355–376.
Alferez, M., Kulesza, U., Moreira, A., Araujo, J., Amaral, V., 2008. Tracing between features and use cases: amodel-driven approach. In: VAMOS'08, pp. 81–88.
Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., Amaral, V., 2009. Multi-view composition language for software product line requirements. In: SLE'09, pp. 103–122.

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C., 2006. Refactoring product lines. In: GPCE'06, pp. 201–210.
Alves, V., Niu, N., Alves, C., Valença, G., 2010. Requirements engineering for software product lines: a systematic review. Inf. Softw. Technol. 52, 806–820.
Amyot, D., 2003. Introduction to the user requirements notation: learning by example. Comput. Netw. 42 (3), 285–301.
Arora, C., Sabetzadeh, M., Goknil, A., Briand, L.C., Zimmer, F., 2015a. Change impact analysis for natural language requirements: an nlp approach. In: RE'15, pp. 6–15.
Arora, C., Sabetzadeh, M., Goknil, A., Briand, L.C., Zimmer, F., 2015b. NARCIA: an automated tool for change impact analysis in natural language requirements. In: ESEC/SIGSOFT FSE'15, pp. 962–965.
Batory, D., 2005. Feature models, grammars, and propositional formulas. In: SPLC'05, pp. 7–20.
Benavides, D., Ruiz-Cortés, A., Trinidad, P., 2005a. Using constraint programming to reason on feature models. In: SEKE'05, pp. 677–682.
Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: a literature review. Inf. Syst. 35 (6), 615–636.
Benavides, D., Trinidad, P., Ruiz-Cortés, A., 2005b. Automated reasoning on feature models. In: CAiSE'05, pp. 491–503.
Botterweck, G., Pleuss, A., 2014. Evolution of software product lines. Evolving Software Systems. Springer.
Braganca, A., Machado, R.J., 2007. Automating mappings between use case diagrams and feature models for software product lines. In: SPLC'07, pp. 3–12.
Buhne, S., Halmans, G., Lauenroth, K., Pohl, K., 2006. Scenario-based application requirements engineering. Software Product Lines. Springer.
Buhne, S., Halmans, G., Pohl, K., 2003. Modeling dependencies between variation points in use case diagrams. In: REFSQ'03, pp. 59–69.
Bürdek, J., Kehrer, T., Lochau, M., Reulig, D., Kelter, U., Schürr, A., 2015. Reasoning about product-line evolution using complex feature model differences. Automated Software Engineering 1–47.
Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., Christina, S., 2005. Goal-centric traceability for managing non-functional requirements. In: ICSE'05, pp. 362–371.
Czarnecki, K., Antkiewicz, M., 2005. Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, pp. 422–437.
Dhungana, D., Grünbacher, P., Rabiser, R., 2011. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Autom. Softw. Eng. 18, 77–114.
Dintzner, N., Kulesza, U., van Deursen, A., Pinzger, M., 2014. Evaluating feature change impact on multi-product line configurations using partial information. In: ICSR'15, pp. 1–16.
Durán, A., Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A., 2017. FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. Softw. Syst. Model. 16, 1049–1082.
Eclipse EMF, https://eclipse.org/modeling/emf/.
Eriksson, M., Borstler, J., Asa, A., 2004. Marrying features and use cases for product line requirements modeling of embedded systems. In: SERPS'04, pp. 73–82.
Eriksson, M., Borstler, J., Borg, K., 2005. The pluss approach - domain modeling with features, use cases and use case realizations. In: SPLC'05, pp. 33–44.
Eriksson, M., Borstler, J., Borg, K., 2009. Managing requirements specifications for product lines - an approach and industry case study. J. Syst. Softw. 82, 435–447.
Fan, S., Zhang, N., 2006. Feature model based on description logics. In: KES'06, pp. 1144–1151.
Fantechi, A., Gnesi, S., John, I., Lami, G., Dorr, J., 2004a. Elicitation of use cases for product lines. In: PFE'03, pp. 152–167.
Fantechi, A., Gnesi, S., Lami, G., Nesti, E., 2004b. A methodology for the derivation and verification of use cases for product lines. In: SPLC'04, pp. 255–265.
Giese, H., Wagner, R., 2009. From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. 8, 21–43.
Goknil, A., Kurtev, I., van den Berg, K., 2008a. Change impact analysis based on formalization of trace relations for requirements. In: ECMDA-TW'08, pp. 59–75.
Goknil, A., Kurtev, I., van den Berg, K., 2008b. A metamodeling approach for reasoning about requirements. In: ECMDA-FA'08, pp. 310–325.
Goknil, A., Kurtev, I., van den Berg, K., 2014a. Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software 88, 112–137.
Goknil, A., Kurtev, I., van den Berg, K., 2016a. A rule-based approach for evolution of aadl models based on changes in functional requirements. ECSA Workshops.
Goknil, A., Kurtev, I., van den Berg, K., 2016b. A rule-based change impact analysis approach in software architecture for requirements changes. arXiv:1608.02757.
Goknil, A., Kurtev, I., van den Berg, K., Spijkerman, W., 2014b. Change impact analysis for requirements: a metamodeling approach. Inf. Softw. Technol. 56 (8), 950–972.
Goknil, A., Kurtev, I., van den Berg, K., Veldhuis, J.-W., 2011. Semantics of trace relations in requirements models for consistency checking and inferencing. Softw. Syst. Model. 10, 31–54.
Goknil, A., Kurtev, I., Millo, J.-V., 2013. A metamodeling approach for reasoning on multiple requirements models. In: EDOC'13, pp. 159–166.
Griss, M.L., Favaro, J., d'Alessandro, M., 1998. Integrating feature modeling with the rseb. In: ICSR'98, pp. 76–85.
Hajri, I., 2016. Supporting change in product lines within the context of use case-driven development and testing. In: Doctoral Symposium - FSE'16, pp. 1082–1084.
Hajri, I., Goknil, A., Briand, L.C., 2017a. A change management approach in product lines for use case-driven development and testing. Poster and Tool Track at REFSQ'17.
Hajri, I., Goknil, A., Briand, L.C., Stephany, T., 2015. Applying product line use case modeling in an industrial automotive embedded system: lessons learned and a refined approach. In: MODELS'15, pp. 338–347.
Hajri, I., Goknil, A., Briand, L.C., Stephany, T., 2016a. Configuring use case models in product families. Softw. Syst. Model. https://link.springer.com/article/10.1007%2Fs10270-016-0539-8.
Hajri, I., Goknil, A., Briand, L.C., Stephany, T., 2016b. PUMConf: A tool to configure product specific use case and domain models in a product line. In: FSE'16, pp. 1008–1012.
Hajri, I., Goknil, A., Briand, L.C., Stephany, T., 2017b. Incremental reconfiguration of product specific use case models for evolving configuration decisions. In: REFSQ'17, pp. 1–19.
Halmans, G., Pohl, K., 2003. Communicating the variability of a software-product family to customers. Softw. Syst. Model. 2, 15–36.
Hearnden, D., Lawley, M., Raymond, K., 2006. Incremental model transformation for the evolution of model-driven systems. In: MODELS'06, pp. 321–335.
Heider, W., Rabiser, R., Grünbacher, P., 2012a. Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. Int. J. Software Tools Technol. Trans. 5, 613–630.
Heider, W., Rabiser, R., Lettner, D., Grünbacher, P., 2012b. Using regression testing to analyze the impact of changes to variability models on products. In: SPLC'12, pp. 196–205.
IEE (International Electronics & Engineering) S.A., http://www.iee.lu/.
Jahann, S., Egyed, A., 2004. Instant and incremental transformation of models. In: ASE'04, pp. 362–365.
Karatas, A.S., Oguztuzun, H., Dogru, A., 2010. Mapping extended feature models to constraint logic programming over finite domains. In: SPLC'10, pp. 286–299.
Kolovos, D.S., Paige, R.F., Polack, F., 2006a. The epsilon object language (eol). In: ECMDA-FA'06, pp. 128–142.
Kolovos, D.S., Paige, R.F., Polack, F., 2006b. Merging models with the epsilon merging language (eml). In: MODELS'06, pp. 215–229.
Kolovos, D.S., Paige, R.F., Polack, F.A., 2008. The epsilon transformation language. In: ICMT'08, pp. 46–60.
Kurtev, I., Dee, M., Göknil, A., van den Berg, K., 2007. Traceability-based change management in operational mappings. In: ECMDA-TW'07, pp. 57–67.
Liang, J.H., Ganesh, V., Czarnecki, K., Raman, V., 2015. Sat-based analysis of large real-world feature models is easy. In: SPLC'15, pp. 91–100.
Mannion, M., 2002. Using first-order logic for product line model validation. In: SPLC'02, pp. 176–187.
Mannion, M., Camara, J., 2003. Theorem proving for product line model verification. In: PFE'03, pp. 211–224.
Mendonca, M., Wasowski, A., Czarnecki, K., 2009. SAT-based analysis of feature models is easy. In: SPLC'09, pp. 231–240.
Nebut, C., Fleurey, F., Traon, Y.L., Jezequel, J.-M., 2006a. Automatic test generation: a use case driven approach. IEEE Trans. Softw. Eng. 32 (3), 140–155.
Nebut, C., Traon, Y.L., Jezequel, J.-M., 2006b. System testing of product families: from requirements to test cases. Software Product Lines. Springer.
Nejati, S., Sabetzadeh, M., Arora, C., Briand, L.C., Mandoux, F., 2016. Automated change impact analysis between SysML models of requirements and design. FSE'16.
Nielsen, J., 1994. Usability inspection methods. In: CHI '94, pp. 413–414.
Nielsen, J., Molich, R., 1990. Heuristic evaluation of user interfaces. In: CHI '90, pp. 249–256.
Nöhrer, A., Biere, A., Egyed, A., 2012. Managing SAT inconsistencies with HUMUS. In: VaMoS'12, pp. 83–91.
Nöhrer, A., Egyed, A., 2013. C2O configurator: a tool for guided decision-making. Autom. Softw. Eng. 20, 265–296.
Oppenheim, A.N., 2005. Questionnaire Design, Interviewing and Attitude Measurement. Continuum.
Paskevicius, P., Damasevicius, R., Štuikys, V., 2012. Change impact analysis of feature models. In: ICIST'12, pp. 108–122.
Passos, L., Czarnecki, K., Apel, S., Wasowski, A., Kästner, C., Guo, J., 2013. Feature-oriented software evolution. VaMoS'13.
Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S., 2012. Model–driven support for product line evolution on feature level. J. Syst. Softw. 85, 2261–2274.
Pohl, K., Bockle, G., van der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer.
Quinton, C., Rabiser, R., Vierhauser, M., Grunbacher, P., Baresi, L., 2015. Evolution in dynamic software product lines: challenges and perspectives. In: SPLC'15, pp. 126–130.
Rabiser, R., Grünbacher, P., Dhungana, D., 2010. Requirements for product derivation support: results from a systematic literature review. Inf. Softw. Technol. 52, 324–346.
Ramesh, B., Jarke, M., 2001. Toward reference models for requirements traceability. IEEE Trans. Softw. Eng. 27 (1), 58–93.
Rosa, M.L., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., 2009. Questionnaire-based variability modeling for system configuration. Softw. Syst. Model. 8, 251–274.
Santos, I.S., Andrade, R.M., Neto, P.A.S., 2015. Templates for textual use cases of software product lines: results from a systematic mapping study and a controlled experiment. J. Softw. Eng. Res.Dev. 3 (5), 1–29.
Seidl, C., Heidenreich, F., Aßmann, U., 2012. Co-evolution of models and feature mapping in software product lines. In: SPLC'12, pp. 76–85.

Sepulveda, S., Cravero, A., Cachero, C., 2016. Requirements modeling languages for software product lines: a systematic literature review. Inf. Softw. Technol. 69, 16–36.

Smith, S.L., Mosier, J.N., 1986. Guidelines for Designing User Interface Software. Natl Technical Information.

ten Hove, D., Goknil, A., Kurtev, I., van den Berg, K., de Goede, K., 2009. Change impact analysis for SysML requirements models based on semantics of trace relations. In: ECMDA-TW'09, pp. 17–28.

Thüm, T., Batory, D.S., Kästner, C., 2009. Reasoning about edits to feature models. In: ICSE'09, pp. 254–264.

Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., Toro, M., 2008. Automated error analysis for the agilization of feature modeling. J. Syst. Softw. 81, 883–896.

Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B., 2009. Incremental model synchronization for efficient run-time monitoring. In: MODELS Workshops 2009, pp. 124–139.

Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, M.Z.Z., 2015a. Automatic generation of system test cases from use case specifications. In: ISSTA'15, pp. 385–396.

Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, M.Z.Z., 2015b. UMTG: a toolset to automatically generate system test cases from use case specifications. In: ESEC/SIGSOFT FSE'15, pp. 942–945.

Wang, H., Li, Y.F., Sun, J., Zhang, H., Pan, J., 2005. A semantic web approach to feature modeling and verification. SWESE'05.

Wang, H., Li, Y.F., Zhang, H., Pan, J., 2007. Verifying feature models using owl. J. Web Semant. 5, 117–129.

White, J., Benavides, D., Schmidt, D.C., Trinidad, P., Dougherty, B., Ruiz-Cortés, A., 2010. Automated diagnosis of feature model configurations. J. Syst. Softw. 83, 1094–1107.

White, J., Galindo, J.A., Saxena, T., Dougherty, B., Benavides, D., Schmidt, D.C., 2014. Evolving feature model configurations in software product lines. J. Syst. Softw. 119–136.

White, J., Schmidt, D.C., Benavides, D., Trinidad, P., Ruiz-Cortés, A., 2008. Automated diagnosis of product-line configuration errors in feature models. In: SPLC'08, pp. 225–234.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A., 2012. Experimentation in Software Engineering. Springer.

Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H., 2007. Towards automatic model synchronization from model transformations. In: ASE'07, pp. 164–173.

Yue, T., Briand, L.C., Labiche, Y., 2013. Facilitating the transition from use case models to analysis models: approach and experiments. ACM Trans. Softw. Eng. Method. 22 (1).

Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U., 2009. Vml* – a family of languages for variability management in software product lines. In: SLE'09, pp. 82–102.

Ines Hajri received in 2012 the MS degree in Modeling from the Higher Institute of Management of Tunis, Tunisia. She is currently working towards the PhD degree at the SnT Centre for Security, Reliability, and Trust, the University of Luxembourg, Luxembourg. Her research interests are in software engineering with an emphasis on requirements engineering, model-driven engineering, product lines, and change impact analysis in software systems. She also worked as quality assurance engineer at a multinational company in Tunisia. More details can be found at: http://people.svv.lu/hajri/.

Arda Goknil is a research associate at Software Verification & Validation Lab in Interdisciplinary Centre for Security, Reliability and Trust (SnT) in University of Luxembourg. Between 2011 and 2013, he was a postdoctoral fellow at AOSTE research team of INRIA in France. He received his PhD from the Software Engineering Group (TRESE) of the University of Twente in the Netherlands in 2011. The dissertation title is "Traceability of Requirements and Software Architecture for Change Management". His current research is about Requirements Engineering and Architectural Design activities with a special emphasis on change management, product line engineering and testing.

Lionel C. Briand is professor and FNR PEARL chair in software verification and validation at the SnT centre for Security, Reliability, and Trust, University of Luxembourg. He also acts as vice-director of the centre. Lionel started his career as a software engineer in France (CS Communications & Systems) and has conducted applied research in collaboration with industry for more than 20 years. Until moving to Luxembourg in January 2012, he was heading the Certus center for software verification and validation at Simula Research Laboratory, where he was leading applied research projects in collaboration with industrial partners. Before that, he was on the faculty of the department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was full professor and held the Canada Research Chair (Tier I) in Software Quality Engineering. He has also been the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland, USA. Lionel was elevated to the grade of IEEE Fellow for his work on the testing of object-oriented systems. He was also granted the IEEE Computer Society Harlan Mills award and the IEEE Reliability Society engineer-of-the-year award for his work on model-based verification and testing. In 2016, he was also the recipient of an ERC Advanced grant from the European Commission. His research interests include: software testing and verification, model-driven software development, search-based software engineering, and empirical software engineering. Lionel has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the coeditor-in-chief of Empirical Software Engineering (Springer) and is a member of the editorial boards of Systems and Software Modeling (Springer) and Software Testing, Verification, and Reliability (Wiley).

Thierry Stephany, born and currently residing in Luxembourg, is a software development manager with over 18 years of experience in developing safety critical automotive systems. His experience ranges over five companies, including IEE, Siemens, Dupont and CFL.