Noname manuscript No.
(will be inserted by the editor)

Configuring use case models in product families

Ines Hajri · Arda Goknil · Lionel C. Briand · Thierry Stephany

the date of receipt and acceptance should be inserted later

Abstract In many domains such as automotive and avionics, the size and complexity of software systems is quickly increasing. At the same time, many stakeholders tend to be involved in the development of such systems, which typically must also be configured for multiple customers with varying needs. Product Line Engineering (PLE) is therefore an inevitable practice for such systems. Furthermore, because in many areas requirements must be explicit and traceability to them is required by standards, use cases and domain models are common practice for requirements elicitation and analysis. In this paper, based on the above observations, we aim at supporting PLE in the context of use case-centric development. Therefore, we propose, apply, and assess a use case-driven configuration approach which interactively receives configuration decisions from the analysts to generate Product Specific (PS) use case and domain models. Our approach provides the following: (1) a use case-centric product line modeling method (PUM), (2) automated, interactive configuration support based on PUM, and (3) an automatic generation of PS use case and domain models from Product Line (PL) models and configuration decisions. The approach is supported by a tool relying on Natural Language Processing (NLP), and integrated with an industrial requirements management tool, i.e., IBM Doors. We successfully applied and evaluated our approach to an industrial case study in the automotive domain, thus showing evidence that the approach is practical and beneficial to capture variability at the appropriate level of granularity and to configure PS use case and domain models in industrial settings.

Ines Hajri, Arda Goknil, and Lionel C. Briand
SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
Thierry Stephany
International Electronics & Engineering (IEE), Contern, Luxembourg
E-mail: {ines.hajri, arda.goknil, lionel.Briand}@uni.lu
thierry.stephany@iee.lu

1 Introduction

In various domains such as automotive and avionics, software systems are quickly getting larger and more complex. These systems often consist of various interacting subsystems, e.g., lighting systems, engine controller, and sensing systems. Further, many suppliers are typically involved in system development. Each supplier is mostly specialized in developing one or two of these subsystems. For example, in the automotive domain, while one supplier provides a sensing system monitoring the driver seat for occupancy by means of a pressure sensitive sensor, another supplier may develop an engine controller using the output of the sensing system to prevent unintentional vehicle starts and unnecessary fuel consumption. These suppliers develop multiple versions of the same product since they work with many manufacturers (customers). Therefore, given the complexity arising from the context described above, systematic and supported Product Line Engineering (PLE) is crucial in their software development practice, from requirements analysis to implementation and testing.

Our work was motivated by discussions with IEE S.A. (in the following "IEE") [3], a leading supplier in automotive sensing systems enhancing safety and comfort in vehicles produced by major car manufacturers worldwide. Such systems monitor the physical environment by means of physical components (e.g., electrical field sensors, pressure sensitive sensors, and force sensing resistors), detect events or changes in the existence of objects and humans (e.g., seat occupant classification, gesture recognition, and driver presence detection), and provide the corresponding output to other subsystems (e.g., airbag control unit, trunk controller, and engine controller). At IEE, similar to many other development environments, use cases (including use case diagrams and use case specifications) are the main artifacts employed to elicit requirements and communicate with cus-

tomers. In order to clarify the terminology used in the requirements and to provide a common understanding of the application domain concepts, use cases are often accompanied by a domain model formalizing concepts and their relationships, often under the form of a class diagram with constraints.

For each new product family, it is common to start a development project with an initial customer providing requirements for a single product. The analysts elicit and document the initial product requirements as use cases and a domain model. The use cases and domain model are copied, modified, and then maintained for each new customer. This practice is called *clone-and-own* reuse [25] and requires the entire use cases and domain model to be evaluated manually to identify changes since nothing indicates what the variable requirements are and where they are located. The clone-and-own approach is fully manual, error-prone, and time-consuming in industrial practice. Therefore, more efficient and automated techniques are required to manage reuse of common and variable requirements given as use cases and domain model across products in a product line. One significant step in that direction is automated configuration that aims at guiding configuration decisions for Product Line (PL) use cases and domain model. In our context, the analysts explicitly specify PL variability, and automatically generate Product Specific (PS) use cases and domain model for a configured product. This is expected, in the context of use case-driven development, to facilitate the reuse of use case models for multiple products. After assessing relevant techniques in the literature, we developed a complete use case-driven configuration approach supporting embedded system development, though we expect it to be easily applicable in other contexts. We target, to the largest extent possible, common software modeling practices, to achieve widespread applicability.

The benefits of use case-driven configuration have been acknowledged and there are proposed approaches in the literature [9,61,6]. Many studies [34,27,8] provide configuration approaches which require that feature models be traced as an orthogonal model to artifacts such as UML use case, activity and class diagrams. In order to employ these approaches in industrial practice, the analysts need to provide feature models with their traces to use cases and related artifacts. The evolution of feature models also requires these traces to be maintained manually by the analysts. Due to deadline pressure and limited resources, many software development companies find such additional traceability and maintainability effort to be impractical. Moon et al. [53,52] propose a method that generates PS use cases from PL use cases without using any feature model. However, the proposed method requires Primitive Requirements (PR) (i.e., building blocks of complex requirements) to be specified by the analysts and traced to the use case diagram and specifications via *PR - Context* and *PR - Use Case* matrices. The configuration takes place by selecting the *PR*s in the matrices without any automated guidance.

To avoid, or at least minimize, additional traceability and maintenance effort required by the approaches in the literature, we first need a modeling method with which we can model variability information explicitly in the use case diagram, specifications, and domain model, without additional artifacts. Therefore, in our previous work [44], which this paper extends, we proposed and assessed the Product line Use case modeling Method (PUM), which enables the analysts to capture and document variability in PL use case diagrams, use case specifications, and domain models. For PL use case diagrams, we employ the diagram extensions proposed by Halmans and Pohl [45]. These extensions overcome the shortcomings of textual representations of variability, such as implicit variants and variation points. Further, for PL use case specifications, we employ Restricted Use Case Modeling (RUCM) [79], which includes a template and restriction rules to reduce imprecision and incompleteness in use cases. RUCM was a clear choice since it reduces ambiguity and facilitates automated analysis of use cases [78,80,73,74]. However, since it was not originally meant to model variability, we introduced some PL extensions to capture variability in use case specifications [44]. To be able to capture variability in PL domain models, we rely on the stereotypes (i.e., *variation*, *variant* and *optional*), proposed by Ziadi and Jezequel [82] for UML class diagrams.

In this paper, we propose, apply, and assess a use case-driven configuration approach based on PUM. Our goal is to provide a degree of configuration automation that enables effective product-line management in use case-driven development, without requiring additional modeling artifacts and traceability effort. Our approach supports four activities. First, the analysts model the variability information explicitly in a PL use case diagram, its use case specifications, and its corresponding domain model. Second, the consistency of the PL use case diagram and specifications are checked and inconsistencies are reported if there are any. For instance, a variation point in the use case diagram might be missing in the corresponding use case specification or a use case specification may not conform to the extended RUCM template. Third, the analyst is guided to make configuration decisions based on variability information in the PL models. The partial order of decisions to be made is automatically identified from the dependencies among variation points and variant use cases. In the case of contradicting configuration decisions, such as two decisions resulting in selecting variant use cases violating some dependency constraints, we automatically detect and report them. The analyst must then backtrack and revise the decisions to resolve these inconsistencies. Alternatively, we could employ constraint solvers

(i.e., SAT solver, BDD solver and Prolog solver) to identify a priori possible contradicting decisions so as to avoid them. However, according to our observation at IEE, customers are also involved in the decision-making process in which they frequently re-evaluate, backtrack and revise their decisions. Therefore, it is important for them to have the possibility to make contradicting decisions and revise prior ones as a result. Fourth, based on configuration decisions, the PS use case and domain models are generated from the PL use case and domain models. To support these activities, we developed a tool, *PUMConf (Product line Use case Model Configurator)*. The tool automatically checks the consistency of the PL models, identifies the partial order of decisions to be made, determines contradicting decisions, and generates PS use case and domain models. To summarize, the contributions of this paper are:

- a configuration approach that is specifically tailored to use case-driven development, and that guides the analysts and customers in making configuration decisions in product lines to automatically generate PS use case and domain models;
- tool support integrated with an industrial requirements management tool (i.e., IBM Doors) as a plug-in, which relies on Natural Language Processing (NLP) to report inconsistencies in PL use case models and contradicting configuration decisions, and to automatically generate PS use case and domain models;
- an industrial case study demonstrating the applicability and benefits of our configuration approach.

This paper is structured as follows. Section 2 introduces the industrial context of our case study to illustrate the practical motivations for our configuration approach. Section 3 discusses the related work in light of our needs. In Section 4, we provide an overview of the approach. Section 5 provides a brief description of PL use case and domain modeling proposed in our previous work, which this paper extends. In Section 6, we illustrate our approach through example models. In Sections 7 and 8, we provide the details of the core technical parts of our approach: consistency checking of configuration decisions and generation of PS use case and domain models. Section 9 presents our (publicly available) tool support for configuration, while Section 10 presents our industrial case study, involving an embedded system called *Smart Trunk Opener (STO)*, along with results and lessons learned. We conclude the paper in Section 11.

2 Motivation and Context

Our configuration approach is developed for the context of embedded software systems, interacting with multiple other external systems, and developed according a use case-driven process, by a supplier for multiple manufacturers (customers). In such a context, requirements variability is communicated to customers and an interactive configuration process is followed for which guidance and automated support are needed. For instance, for each product in a product family, IEE negotiates with customers how to resolve variation points in requirements, in other words how to configure the product line.

In this paper, we use *Smart Trunk Opener (STO)* as a case study, to motivate 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.

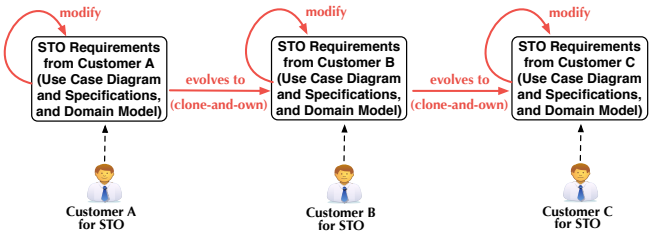


Fig. 1 Clone-and-Own Reuse at IEE for a Product Family

The current use case-driven development practice at IEE, like in many other environments, is based on clone-and-own reuse [25] (see Fig. 1). IEE starts a project with an initial customer. The product requirements are elicited from the initial customer and documented as a use case diagram, use case specifications, and a domain model. For each new customer in the product family, the IEE analysts need to clone the current models, and negotiate variabilities with the customer to produce a new use case diagram, set of specifications, and domain model (see *clone-and-own* in Fig. 1). As a result of the negotiations, the IEE analysts make changes in the cloned models (see *modify*). With such practice, variants and variation points (i.e., where potential changes are made) are not documented and IEE analysts, together with the customer, need to evaluate the entire use cases and domain model.

Fig. 2 depicts part of the initial UML use case diagram of the STO product, which describes four main functions: *recognize gesture*, *provide system operating status*, *clear error status*, and *provide system user data*.

In the *clone-and-own* reuse, the initial diagram forms the baseline to negotiate the STO requirements with other potential customers. For instance, a second customer could require all use cases except *Store Error Status*, *Clear Error*

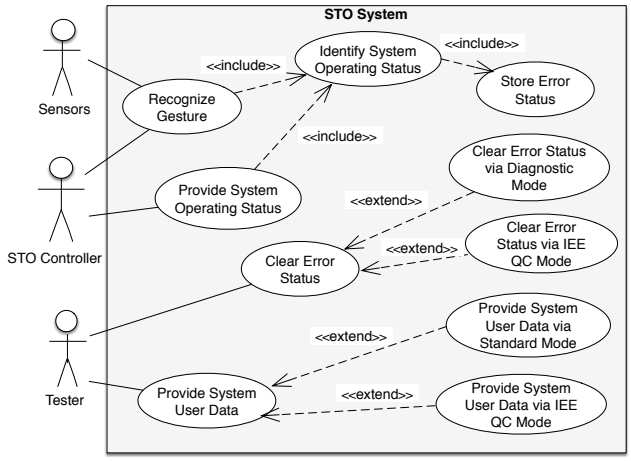


Fig. 2 Part of the UML Use Case Diagram for STO

Status, and two use cases extending *Clear Error Status*. A third customer could need another method of providing system user data as a new use case *Provide System User Data via Diagnostic Mode*, which extends *Provide System User Data* but does not exist in the initial diagram in Fig. 2. The IEE analysts need to clone the diagram for each new customer, negotiate the diagram changes with the customer, and then modify it.

One solution to automate the current practice is to have PL use cases in which variability information is explicitly represented. A configurator can guide the analysts and customers to automatically generate PS use cases by processing the variability information in PL use cases. However, UML does not allow to explicitly represent variability information in the diagram, e.g., which use cases are mandatory and which ones are variant. For instance, in Fig. 2, there are three variation points: *Clear Error Status*, *Store Error Status*, and *Provide System User Data*. *Clear Error Status* and *Store Error Status* are optional while *Provide System User Data* is mandatory. We also need to represent cardinality constraints over these variation points, i.e., the number of variants to be chosen. For instance, there are at least three ways of operationalizing *Provide System User Data*: *Provide System User Data via Diagnostic Mode* is optional while the other two are mandatory. In addition, there are dependencies between variation points, e.g., having *Store Error Status* in an STO product requires having *Clear Error Status* in the same product.

A use case diagram is accompanied by a set of use case specifications providing detailed use case descriptions. The configuration also has to take into account these detailed descriptions since some variability information (e.g., optional use case steps and flows) cannot be captured in a use case diagram but only in specifications. However, a use case specification usually conforms to a standard use case template which, in general, does not provide any means to document variability information [26,47,10]. Capturing the most popular guidelines, the Cockburn's template [26] has been so far followed at IEE to document use case specifications (see Table 1).

Table 1 Some Initial STO Use Case Specifications

1	**USE CASE** Recognize Gesture
2	1. The system *'identifies system operating status'*.
3	2. The system receives the move capacitance from the sensors.
4	3. The system confirms the movement is a valid kick.
5	4. The system informs the trunk controller about the valid kick.
6	**Extensions**
7	3a. The movement is not a valid kick.
8	3a1. The system sets the overuse counter.
9	
10	**USE CASE** Identify System Operating Status
11	**Main Success Scenario**
12	1. The system checks Watchdog reset and RAM.
13	2. The system checks the upper and lower sensors.
14	3. The system checks if there is any error detected.
15	**Extensions**
16	2a. Sensors are not working properly.
17	2a1. The system identifies a sensor error.
18	3a. There is an error in the system.
19	3a1. The system *'stores error status'*.
20	
21	**USE CASE** Provide System User Data
22	1. The tester requests receiving system user data via standard mode.
23	2. The system *'provides system user data via Standard Mode'*.
24	**Extensions**
25	1a. The tester requests receiving system user data via IEE QC mode.
26	1a1. The system *'provides system user data via IEE QC Mode'*.
27	
28	**USE CASE** Provide System User Data via Standard Mode
29	**Main Success Scenario**
30	1. The system sends the calibration data to the tester.
31	2. The system sends the sensor data to the tester.
32	3. The system sends the trace data to the tester.
33	4. The system sends the error data to the tester.
34	5. The system sends the error trace data to the tester.

Variation points and variant use cases are not visible in Table 1. These are the main elements for which the customer has to make a decision. Mandatory and variant use cases, e.g., *Recognize Gesture* in Lines 1-8 and *Provide System User Data via Standard Mode* in Lines 28-34, cannot be distinguished from each other. Variation points are given as extensions of the basic flow, e.g., executing *Store Error Status*, *Provide System User Data via Standard Mode*, and *Provide System User Data via IEE QC Mode* in Lines 18-19 and 25-26. However, this is no different from the execution of a mandatory use case in another use case (the *include* relationship), e.g., *Identify System Operating Status* in Line 2. With standard templates, it is also not possible to specify optional steps or their order, which may vary across products, e.g.,

the steps in Lines 30-34 which are optional with a variant order. Without explicit variability information, the analysts manually clone the specifications, evaluate the entire documentation to negotiate potential changes with the customer, and finally update the cloned specifications. For instance, for a new customer asking a diagnostic mode, the IEE analysts manually add a new use case specification *Provide System User Data via Diagnostic Mode* and yet another extension to *Provide System User Data* (see Lines 24-26). On the other hand, if the same customer does not require calibration and sensor data to be sent to the tester in the standard mode, the analysts need to manually delete the corresponding use case steps from *Provide System User Data via Standard Mode* (Lines 30 and 31).

We observe that IEE, like many other companies, employs a domain model to document and clarify domain entities mentioned in use case specifications. In a product family, variability also occurs in domain entities and needs to be managed together with use cases for each customer.

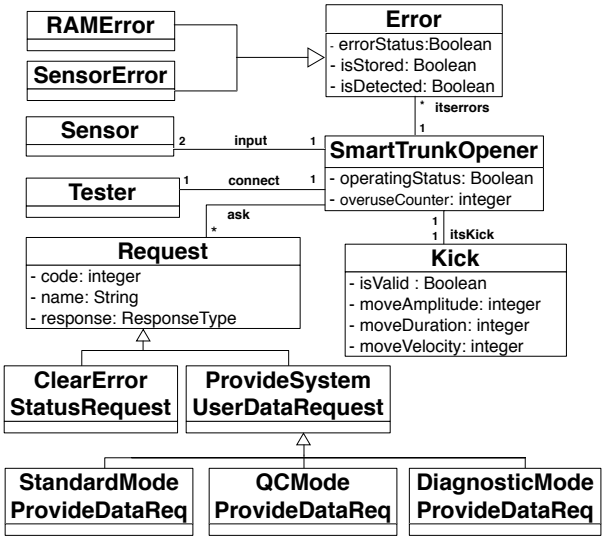


Fig. 3 Simplified Portion of the Initial STO Domain Model

Fig. 3 presents part of the STO domain model for the initial customer in the *clone-and-own* reuse context. Some domain entities (e.g., *ClearErrorStatusRequest* and *DiagnosticModeProvideDataReq*) are not requested for every STO product while some STO products require additional domain entities (e.g., *VoltageDiagnostic*) for their domain model. Without having explicit variability information of what entity is being optional, the IEE analysts need to evaluate each entity in the initial domain model for each new customer.

Within our context, we identify two challenges that should also apply to other environments and that need to be considered in reusing use cases and a domain model for a product family:

Challenge 1: **Modeling Variability Information with Least Possible Modeling Overhead.** The analysts need to explicitly document variability information (e.g., variant use cases, variation points, and optional steps) for use cases and a domain model to be communicated with the customers during product configuration. Relating feature models to use cases and domain model is the most straightforward option but has shortcomings in terms of additional modeling and traceability effort. For example, it would not be easy for analysts and customers to comprehend and visualize all variability information traced to use case diagram, use case specifications and domain model. In STO, we identified 11 mandatory and 13 variant use cases which contain 7 variation points, 12 constraints associated with these variation points, and 7 variant dependencies. The STO use cases include 211 use case flows (24 basic flows and 188 alternative flows) with 5 optional steps while the STO domain model contains 11 variant domain entities. The variability information scattered across all these use case flows with trace links from feature models would be communicated to customers and used to configure a product.

Challenge 2: **High Degree of Automation in Use Case-Driven Configuration.** In order to facilitate use case-driven configuration in industrial practice, a high degree of automation is a must while the analysts are interactively guided for their decisions. Before the configuration, it should be automatically confirmed that all artifacts with variability information, including use case diagram, specifications, and domain model, are consistent. Any inconsistency in these artifacts may cause invalid configuration outputs. Adding to the complexity affecting the decision-making process during configuration, there may be contradicting decisions and hierarchies among decisions. During the configuration process, the analysts need to be interactively informed about contradicting decisions and the order of possible decisions. With interactive guidance and proper tool support, the analysts can fix inconsistent PL artifacts and resolve decision contradictions, which leads to the automatic generation of PS use cases and their domain model.

We addressed the first challenge in our previous work [44] with the *Product line Use case modeling Method (PUM)*. In the remainder of this paper, we focus on how to best address the second challenge in a practical manner, in the context of use case-driven development, while relying on PUM to minimize the modeling overhead.

3 Related Work

In this section, we cover the related work across three categories.

Configuration Techniques for Requirements Variability. Eriksson et al. [34] provide an approach to manage natural-language requirements specifications in the software prod-

uct line context. Variability is captured and managed using a feature model while requirements in various forms, e.g., use cases, textual requirements specifications and domain model, are traced to the feature model. The analyst selects the features in the feature model to be included in the product. By following traces from the selected features to the requirements, the approach filters those requirements that are relevant for a specific product in the product line. These filtered requirements are then exported as product requirements specifications. The approach does not support any automated decision-making solution (e.g., decision ordering, decision consistency checking, and inferring decisions) for selecting features (*Challenge 2*). In addition, the analyst has to manually assign traces between features and requirements at a very low level of granularity, i.e., sequences of use case steps (*Challenge 1*). pure::variants [5] is a tool to manage all parts of software products with their components, restrictions and terms of usage. Its extension, pure::variants for IBM DOORS [4], enables the analyst to capture variability as features in a feature model, and trace them to requirements specifications in IBM DOORS. It transforms the requirements specifications into product requirements based on the selected features in the feature model. Compared to the approach proposed by Eriksson et al. [34], pure::variants for IBM DOORS provides a better automated support, i.e., an automated contradiction detection for feature models. However, the analyst still suffers from the same modeling overhead, when pure::variants is employed. The analyst needs to manually establish traces at a very low level granularity and maintain these traces when the feature model or requirements specifications evolve. There are many similar approaches, eg., [42,24,22,7,21], which require modeling and maintenance overhead with poor automated configuration support. Our configuration approach attempts to minimize this overhead by capturing variability information in use case and domain models (*Challenge 1*).

Moon et al. [53,52] propose a method that generates PS use cases from PL use cases without using any feature model. However, the proposed method requires that Primitive Requirements (PR) (i.e., building blocks of complex requirements) be specified by the analyst and traced to the use case diagram and specifications via the *PR - Context* and *PR - Use Case* matrices. The analyst has to manually encode traceability information in these matrices (*Challenge 1*). The configuration takes place by selecting the *PR*s in the matrices without any automated decision-making support (*Challenge 2*).

John and Muthig [46] introduce some product line extensions to use case diagrams and specifications to be able to capture variant use cases without a feature model. They propose a new artifact, called decision model, to represent variation points textually in a tabular form. Each variation point has multiple facts which represent decisions. For each decision, there are actions which describe configuration operations for use cases, e.g., removing parts of a use case. The analyst is expected to configure, with the help of the decision model, the product specific use case diagram and specifications but such a decision model can quickly become too complex for the analyst to comprehend. There is no automated tool support reported for the approach (*Challenge 2*). Faulk [37] proposes the use of a similar decision model to generate PS requirements specifications from PL requirements specifications. Biddle et al. [16] provide support for configuring use case specifications through parametrization. Parameters can be specified anywhere in the name or body of a parameterized use case. The manual assignment of values to parameters is considered as configuring product specific use case specifications (*Challenge 2*). Fantechi et al. [36, 35] propose Product Line Use Cases (PLUC), an extension of the Cockburn use case template with three kinds of tags (i.e., alternative, parametric, and optional). It is not possible with these tags to explicitly represent mandatory and optional variants. Variants and variation points are hidden in use case specifications conforming to PLUC. These two approaches [16,36] do not support variability in the use case diagram and domain model. They also lack automated support for the decision-making process including decision ordering and detection of contradicting decisions (*Challenge 2*).

Annotation- and Composition-based Configuration for Scenario-based Requirements. Some approaches specialize in configuring scenario-based requirements using annotation- and composition-based techniques [6]. The Product Line Use case modeling for Systems and Software engineering approach (PLUSS) proposed by Eriksson et al. [34,32,33] uses feature models to configure requirements in multiple forms including scenario-based requirements models (e.g., use cases and activity diagrams). PLUSS employs annotations throughout requirements to represent how they are related to features. Czarnecki and Antkiewicz [27] propose another configuration approach based on annotation of scenarios using feature models. Activity diagrams are used to specify scenarios. Traces between feature models and activity diagrams are given as special annotations on activity diagrams. To annotate activity diagrams, the approach employs model templates, which contain the union of the model elements, e.g., presence conditions and meta-expressions, in all valid template instances, i.e., annotated activity diagrams. A product is specified by creating a feature configuration based on the feature model. The model template is instantiated automatically by using the feature configuration. The generated template instance is an activity diagram of the specified product. Although the template instantiation is automated, feature configuration is manual (*Challenge 2*). The analyst also has to manually create a feature model and a model template for annotations.

Bonifácio et al. [20,18] propose a framework for modeling the composition process of scenario variability mechanisms (MSVCM). They provide a *weaver* (configurator) that takes a PL use case model, a feature model, a product configuration, and configuration knowledge as input. The product configuration artifact identifies a specific product, which is characterized by a configuration of features in the feature model, while the configuration knowledge relates features to transformations used for automatically generating the PS use case model. These two artifacts are manually created by the analyst (*Challenge 2*). The Variability Modeling Language for Requirements (VML4RE) [8,83] presents a similar solution for the composition of use case diagrams and their selected scenarios represented by activity diagrams. It supports the definition of traces between feature models and requirements (e.g., use case diagram and activity diagram). VML4RE provides a simple set of operators to specify the composition of requirements models for generating PS requirements models. There are more composition-based approaches [71,54,17] to configure scenario-based requirements using feature models.

All these configuration approaches given above require additional modeling and traceability effort for feature models (*Challenge 1*) while most of them do not provide a high degree of automation for the decision-making process (*Challenge 2*). There are approaches [66,72,76] that support the identification and extraction of variable features from given requirements but these approaches still require a considerable manual intervention in the identification of features. In addition, the detailed functionality of a feature is still shown in the traced requirements documents, and this requires frequent context switching, which is not practical in industrial projects. Stoiber and Glinz [67] propose the modularization of variability information in decision tables to avoid context switching, but the analyst still needs to manually encode all decision constraints and traces in such tables, which can easily get too complex to comprehend (*Challenge 1*). Bonifácio et al. [19] argue that annotation-based approaches entangle the representation of common and variant behavior, whereas the composition-based approaches provide a better separation of variant behavior. They compared an annotation-based approach, i.e., PLUSS, with a composition-based approach, i.e., MSVCM, to investigate whether the composition-based approach causes extra costs for modularizing scenario specifications. They concluded that although MSVCM improves modularity, it requires more time to derive PL specifications, and more investments on training.

Configuration Tools. Nie et al. [56] describe the key automation functionalities that configuration tools should support: *inferring decisions*, *consistency checking*, *decision ordering*, *collaborative configuration*, and *reverting decisions* (*Challenge 2*). Our tool, *PUMConf*, automatically infers new configuration decisions based on prior decisions and variation point-variant dependencies. The consistency of all inferred and prior decisions are automatically checked. The analyst can also revert configuration decisions in order to maintain the consistency. *PUMConf* provides decision ordering to guide the analyst in which sequence a set of decisions should be made, by taking into account the hierarchies among variation points. Currently, *PUMConf* does not support collaborative configuration in terms of PS use case models. Collaborative configuration is defined as coordinating the configuration of multiple systems where the configuration of one system depends on the configuration of other systems [56]. We need to extend our PL use case modeling method in such a way that the analyst is able to model dependencies among PL use case models of multiple systems. Our tool can then be extended to support collaborative configuration using such dependencies.

Configuration tools in the literature partially support the key automation functionalities within a context not specific to use case-driven configuration (*Challenge 2*). Le Rosa et al. [62] provide a questionnaire-based system configuration tool to capture system variability based on questionnaire models composed of questions that refer to a set of *facts* to be set to true or false. When the questionnaire is answered by the analyst, the tool assigns values to facts, and derives an individualized system by using the resulting valuation. The tool supports the key functionalities, except collaborative configuration. Another configuration tool is C2O, presented by Nohrer et al. [59,57,58]. The tool enables the analysts to make configuration decisions in an arbitrary order while it guides them by rearranging the order of decisions (decision ordering), inferring decisions to avoid follow-on conflicts (inferring decisions), and provides support in fixing conflicts at a later time (consistency checking and reverting decisions). No support for collaborative configuration is reported for C2O. Myllarniemi et al. [55] present Kumbang, a prototype configurator for product individuals from configurable software product families. The tool focuses on the configuration of architecture models. It supports the key functionalities except inferring decisions and collaborative configuration. SPLOT [51] is a web-based configurator benefiting from SAT solvers and binary decision diagrams to support reasoning and interactive configuration on feature models. COVAMOF [63,64] supports only architecture configuration, while DOPLER [1,30] is a more general configurator which can be customized for multiple artifacts such as components, test cases, or documentation fragments.

The tools given above are either general configurators (e.g., [1,30,62]) or custom configurators for artifacts such as architecture and feature models (e.g., [55,63,64]), which are quite different than our target artifacts. General configurators could be employed to configure PS use case and domain models. For instance, DOPLER [1,30] supports capturing the variability information using decision models and

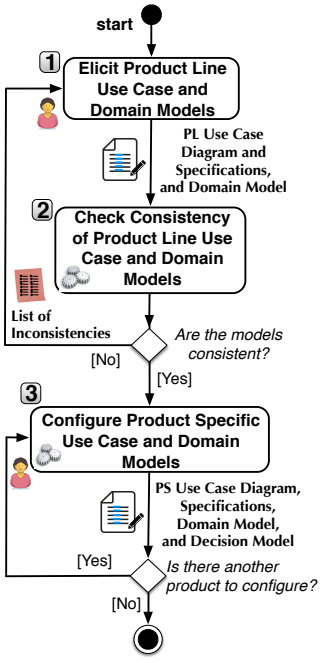
modeling any type of artifact as asset models. Decision and assets are linked by using traceability relations. The analyst has to model variability information in a decision model by using DOPLERVML, a modeling language for defining product lines. Even if use cases and domain models can automatically be translated into an asset model, the analyst still has to manually encode the decisions in the decision model and assign the traceability relations between decision and asset models, which are some inclusion links. Having all these decision and asset models with their explicit traces is exactly the type of modeling practice that we try to avoid in our approach (*Challenge 1*). In addition, DOPLER requires considerable effort and tool-specific internal knowledge to be customized for the consistency checking of configuration decisions and generation of PS use case and domain models.

4 Overview of Our Approach

The process in Fig. 4 presents an overview of our approach. In Step 1, *Elicit product line use case and domain models*, the analyst elicits PL use cases and a domain model with the use case diagram, the RUCM template, and their product line extensions.

Step 1 is manual. Its output includes (1) a *PL use case diagram*, which captures variability, and its constraints and dependencies, (2) *PL use case specifications*, which detail the variability information captured in the diagram, and (3) a *PL domain model*, which captures variability in domain entities (*Challenge 1*). In Step 2, *Check consistency of product line use case and domain models*, our approach automatically checks the consistency of use case diagram, use case specifications (also with the RUCM template), and domain model to report inconsistencies (*Challenge 2*). Steps 1 and 2 are iterative: the PL diagram, specifications, and domain model are updated until full consistency is achieved. We discuss these two steps in Sections 5 and 9.

Fig. 4 Overview of the Approach



In Step 3, *Configure product specific use case and domain models*, the user is asked to input configuration decisions regarding variation points captured in PL use case and domain models to automatically configure the product line into a product. The configuration step is the main focus of this paper. It is described in Section 6.

Step 3 includes an automated, iterative, and interactive decision-making activity (*Challenge 2*). The partial order of configuration decisions to be made is automatically identified from the dependencies among variation points and variant use cases. The analyst is asked to input the configuration decisions in the given partial order. When a decision is made, the consistency of the decision with prior decisions is checked. There might be contradicting decisions in the PL use case diagram such as two decisions resulting in selecting variant use cases violating some dependency constraints. These are automatically determined and reported *a posteriori* and the analyst can backtrack and revise his decisions. Alternatively, the analyst could be guided through the configuration space in such a way that decisions that may lead to contradictions are avoided *a priori* [62]. SAT solvers can be employed to incrementally prune the configuration space in an interactive configuration [13] while CSP solvers are used to handle additional modeling elements in terms of variables (e.g., sets and finite integer domains) and constraints (not only propositional formulas) [15, 14]. However, the use of the SAT and CSP solvers in an iterative and interactive product configuration can be challenging since (a) it can quickly become infeasible to compute inferences, which dynamically prune the configuration space, when the number of variables to be computed is large and (b) it may require considerable implementation effort and internal tool knowledge to use these solvers for computing inferences and detecting and reporting contradictions [13, 38]. According to our observation in industry, customers are also involved in the decision-making process. They need to account for the entire configuration space, including contradicting decisions, because they frequently re-evaluate decisions and possibly update them. Therefore, we decided to have an *a posteriori* approach for the consistency checking of configuration decisions, and implemented our own algorithm fitting our context.

Our motivation is to rely, to the largest extent possible, on a solution that can be easily customized for further extensions addressing automated reconfiguration, including change impact analysis on PL use case and domain models. One may argue that existing configuration tools [62, 64, 59, 28] could be reused within the context of use case-driven configuration but these tools either need feature models [28] or require the variability information to be depicted independently of specific notations or languages, by means of a set of facts [62, 59]. As discussed earlier, not only does this not match our practical needs but, furthermore, we also need a custom solution to generate PS use case and domain models based on the decisions. The entire configuration approach is illustrated by an example in Section 6. We provide

the details of our decision consistency checking algorithm in Section 7, whereas Section 8 presents the generation of PS use case and domain models from PL models.

5 Elicitation of Variability in Use cases

Our configuration approach starts with the activity of elicitation of PL use case and domain models. This activity is based on the Product line Use case modeling Method (PUM) which we present in our previous paper [44]. The artifacts of PUM is a PL use case diagram using product line extensions proposed by Halmans and Pohl [45, 23], PL use case specifications using RUCM extensions which we propose, and a PL domain model using stereotypes provided by Ziadi and Jezequel [82]. In this section, we give a brief description of these PL artifacts.

5.1 Use Case Diagram with Product Line Extensions

For use case diagrams, we employ the PL extensions proposed by Halmans and Pohl [45, 23] since they support explicit representation of variants, variation points, and their dependencies. We do not introduce any further extensions. In this section, we briefly describe the extensions (see Fig. 5).

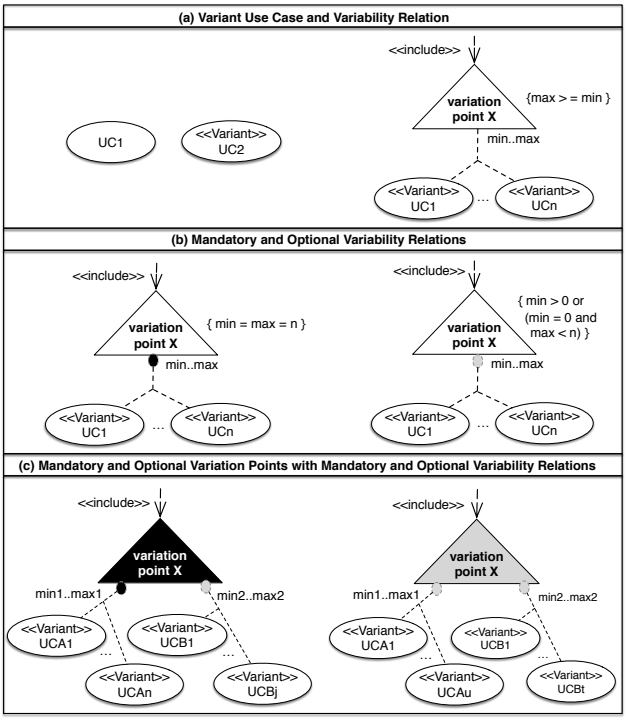


Fig. 5 Graphical Notation of Product Line Extensions for Use Case Diagrams

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 (Fig. 5(a)). A variation point given as a triangle is associated to one, or more than one use case using the 'include' relation. 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 for a variation point. A relation is mandatory where ($min = max = n$). The customer has no choice when a mandatory relation relates mandatory variants to a variation point [45]. Optional and mandatory relations are depicted with light-grey and black filled circles, respectively (Fig. 5(b)). A use case is either *Essential* or *Variant*. The notation for variation points, in Fig. 5, can also be used for variation points in actors [45]. In addition, the extensions allow the analysts to represent the dependencies *requires* and *conflicts* among variation points and variant use cases, e.g., a variant use case *requires* a variation point and a variation point *conflicts* with another variation point [23].

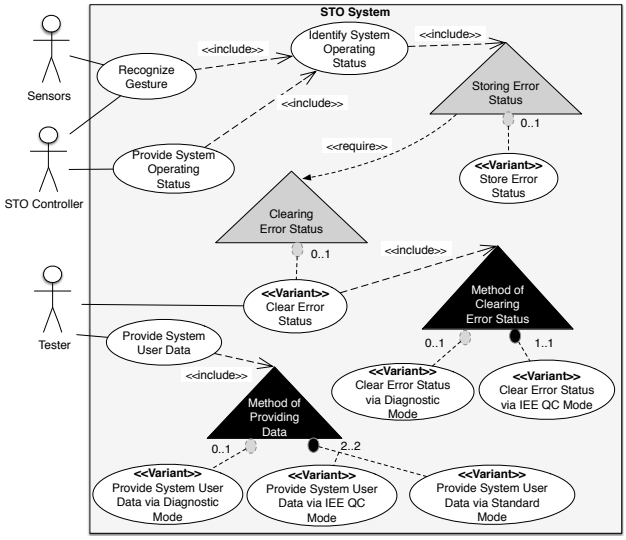


Fig. 6 Part of the Product Line Use Case Diagram for STO

Fig. 6 gives part of the product line use case diagram for STO. We document two optional and two mandatory variation points. The mandatory variation points indicate where the customer has to make a selection for an STO product. 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 (the mandatory variability relation in the 'Method of Providing Data' variation point with a cardinality of '2..2'). In addition, the customer can select the method of sending data via diagnostic mode, i.e., the 'Provide System User Data via Diagnostic Mode' variant use

case with an optional variability relation. In STO, the customer may decide that the system does not store the errors determined while the system identifies its operating status (the 'Identify System Operating Status' essential use case and the 'Storing Error Status' optional variation point). The *require* dependency relates the two optional variation points such that if the customer selects the variant use case in the 'Storing Error Status' variation point, the variant use case in the 'Clearing Error Status' variation point has to be selected.

Use case diagrams capture variants, variation points, their cardinalities and dependencies. Some further variability information can be given in use case specifications. For instance, in Fig. 6, the 'Identify System Operating Status' use case includes the 'Storing Error Status' optional variation point. Only the corresponding use case specification indicates in which flows of events the variation point is included.

5.2 Restricted Use Case Modeling (RUCM) and its Extensions

This section briefly introduces the RUCM template and its PL extensions which we proposed. RUCM provides restriction rules and keywords constraining the use of natural language in use case specifications [79]. We employ RUCM in the elicitation of PL use case specifications since it was designed to make use case specifications more precise and analyzable, while preserving their readability. But since it was not originally designed for PL modeling, we had to introduce PL extensions.

Table 2 provides some of the STO use cases written according to the extended RUCM rules. In RUCM, use cases have basic and alternative flows (Lines 2, 8, 13, 16, 22, 27, 33 and 38). In Table 2, we omit some alternative flows and some basic information such as actors and pre/post conditions.

A basic flow describes a main successful path that satisfies stakeholder interests. It contains use case steps and a postcondition (Lines 3-7, 23-26 and 39-43). A step can be one of the following interactions: an actor sends a request and/or data to the system (Lines 34); the system validates a request and/or data (Line 4); the system replies to an actor with a result (Line 7). A step can also capture the system altering its internal state (Line 18). In addition, the inclusion of another use case is specified as a step. This is the case of Line 3, as denoted by the keyword '*INCLUDE USE CASE*'. All keywords are written in capital letters for readability.

The keyword '*VALIDATES THAT*' (Line 4) indicates a condition that must be true to take the next step, otherwise an alternative flow is taken. In Table 2, the system proceeds to Step 3 (Line 5) if the operating status is valid (Line 4).

Alternative flows describe other scenarios, both success and failure. An alternative flow always depends on a condition in a specific step of the basic flow. In RUCM, there are three types of alternative flows: *specific*, *bounded* and *global*. 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 alternative flow refers to any step in the basic

Table 2 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.

flow. For specific and bounded alternative flows, the keyword '*RFS*' is used to refer to one or more reference flow steps (Lines 9, 14, 17, and 28).

Bounded and global alternative flows begin with '*IF .. THEN*' for the condition under which the alternative flow is taken (Line 10). Specific alternative flows do not necessarily begin with '*IF .. THEN*' since a guard condition is already indicated in its reference flow step (Line 4).

Our RUCM extensions are twofold: (i) new keywords and restriction rules for modeling interactions in embedded systems and restricting the use of existing keywords; (ii) new keywords for modeling variability in use case specifications.

We introduce extensions into RUCM regarding the usage of '*IF*' conditions and the way input/output messages are expressed. PUM follows the guidelines that suggest not to use multiple branches within the same use case path [48], thus enforcing the usage of '*IF*' conditions only as a means to specify guard conditions for alternative flows. PUM introduces the keywords '*SENDS .. TO*' and '*REQUESTS .. FROM*' to distinguish system-actor interactions. According to our experience, in embedded systems, system-actor interactions are always specified in terms of messages. For instance, Step 3 in Table 2 (Line 5) indicates an input message from the sensors to the system while Step 5 (Line 7) contains an output message from the system to the STO Controller. Additional keywords can be defined for other types of systems.

To reflect variability in use case specifications in a restricted form, we introduce into the RUCM template the notion of variation point and variant, complementary to the diagram extensions in Section 5.1. Variation points can be included in basic or alternative flows of use cases. We employ the '*INCLUDE <VARIATION POINT : ... >*' keyword to specify the inclusion of variation points in use case specifications (Lines 29 and 35). Variant use cases are given with the '*<VARIANT >*' keyword (Line 37). The same keyword is also used for variant actors related to a variation point given in the use case diagram.

There are types of variability (e.g, optional steps and optional alternative flows) which cannot be captured in use case diagrams due to the required level of granularity for product configuration. To model such variability, as part of the RUCM template extensions, we introduce optional steps, optional alternative flows and a variant order of steps. Optional steps and optional alternative flows begin with the '*<OPTIONAL>*' keyword (Lines 8 and 39-43). In addition, the order of use case steps may also vary. We use the 'V' keyword before the step number to express the variant step order (Lines 39-43). A variant order occurs with optional and/or mandatory steps. It is important because variability in the system behavior can be introduced by multiple execution orders of the same steps. For instance, the steps of the basic flow of the 'Provide System User Data via Standard Mode' use case are optional. Based on the testing procedure followed in the STO product, the order of sending data to the tester also varies. In the product configuration, the customer has to decide which optional step to include in which order in the use case specification.

5.3 Product Line Domain Model

We encounter two cases (i.e., *variation* and *optionality*) to model variability in domain entities. The first case is an inheritance hierarchy where some subentities are optional and alternative of another entity while others are mandatory. In such a case, the abstract entity is the variation with its variant subentities. In the second case, the domain entity is optional, but not part of any inheritance hierarchy. In order to support these two cases in the PL domain model, we employ the stereotypes (i.e., *variation*, *variant*, and *optional*) which Ziadi and Jezequel [82] propose to model variability in UML class diagrams (see Fig. 7).

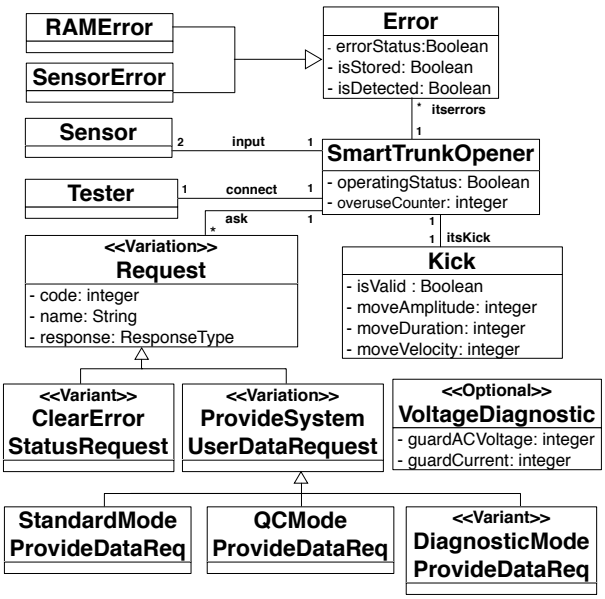


Fig. 7 Simplified Portion of the Product Line Domain Model for STO

The *Variant* and *Variation* stereotypes are used to specify variability associated with an inheritance hierarchy. The variation is the abstract class (*Request* and *ProvideSystemUserDataRequest*) while variants are given as subclasses with the *Variant* stereotype (*ClearErrorStatusRequest* and *DiagnosticModeProvideDataReq*). *QCModeProvideDateReq* and *StandardModeProvideDataReq* do not have any stereotype, thus implying these are mandatory for all STO products. The *Optional* stereotype is for optional entities which are not part of any inheritance hierarchy (e.g., *VoltageDiagnostic*).

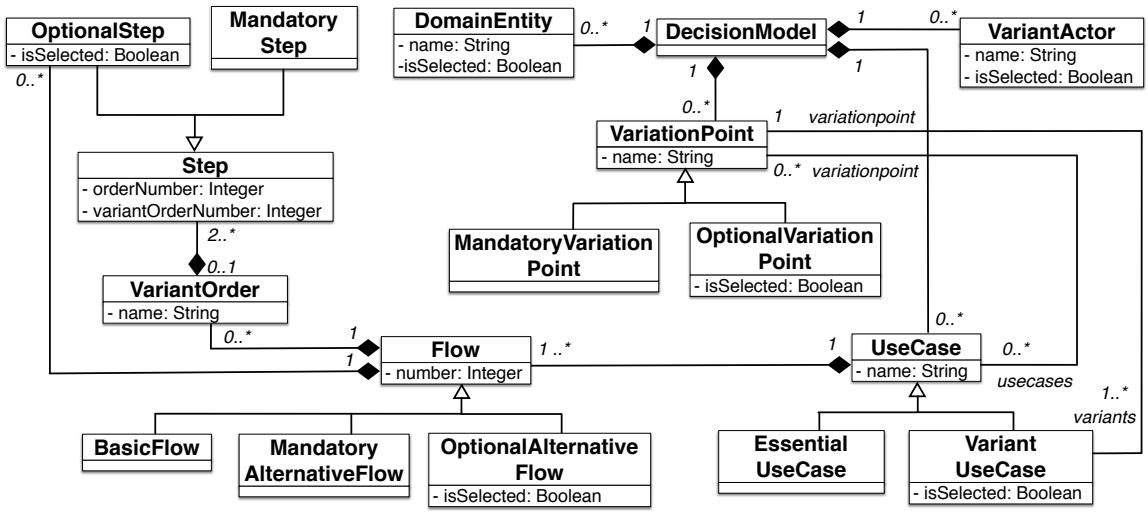


Fig. 8 Decision Metamodel

The PL domain model in Fig. 7 does not contain any variant dependency such that the selection of one variant domain entity disables (or enables) the selection of some other variant entities. Ziadi and Jezequel [82] propose to specify such dependencies in the Object Constraint Language (OCL). Alternatively, some special stereotypes, e.g., *requires* and *conflicts*, can also be employed for UML class associations to specify the variant dependencies in PL domain models.

6 Configuration of Product Specific Use Case and Domain Models

The product configuration is a decision-making process, where the variability information is examined to select the desired features for the product. A product in a product family is defined as a unique combination of features selected during configuration. In this paper, we rely on variability information given in the PL use case diagram, specifications and domain model. The user selects (1) the desired use cases in the PL use case diagram, (2) use case elements in the PL use case specifications, and (3) domain entities in the PL domain model, to generate the PS use case diagram, specifications, and domain model.

Our configuration mechanism relies on a use case configuration function. The configuration function takes a PL use case diagram, a set of PL use case specifications, and a PL domain model as input, and produces a PS use case diagram, a set of PS use case specifications, a PS domain model, and a decision model which captures configuration decisions. Such a decision model is important since the analyst/customer may need to update decisions to reconfigure the PS models for the same product.

The decision model conforms to a decision metamodel, which is described in Fig. 8. We will shortly describe its elements.

There are four main use case elements for which the user has to make 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). In the PL domain model, the user makes decisions for the *Variant* and *Optional* entities. All these decisions are saved in the decision model, whose structure is formalized by the decision metamodel.

The reader is referred to Supplementary Material[1] for the algorithm of the configuration function. At a high level, the algorithm first traverses the use case diagram for variation points, and then processes use case specifications for optional alternative flows, optional steps, and variant orders. Then the domain model is traversed for *Variant* and *Optional* domain entities. In the following, we explain the steps of the algorithm with an illustrative example. The example is a slight adaptation of part of our industrial case study since we needed some additional modeling elements to illustrate the complete set of features of the algorithm. Fig. 9 depicts an example PL use case diagram with four variation points, eight variant use cases, and one essential use case.

The main steps of the configuration algorithm for use case diagrams are:

– *Identifying variation points in the diagram to start the configuration.* In Fig. 9, there are four variation points and two of them are included by variant use cases in another variation point (i.e., *UC2* in *VP1* includes *VP2*, and

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

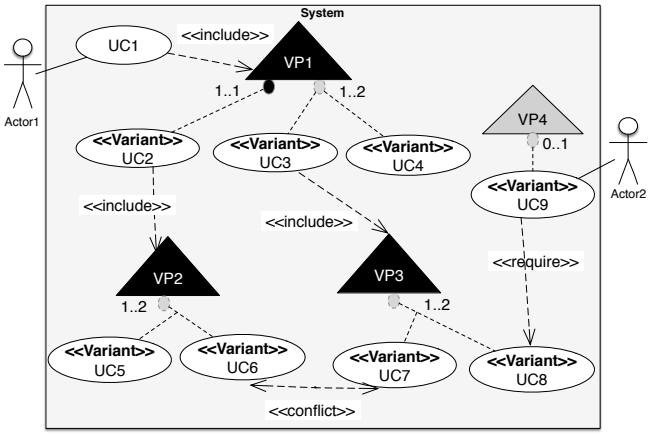


Fig. 9 Example Product Line Use Case Diagram

UC3 includes *VP3*). The algorithm automatically filters out the variation points included by variant use cases because the analyst can make a decision for these variation points only if the including variant use case is selected for the product. For instance, *VP2* will not be considered if *UC2* is not selected in *VP1*. The user can start making decisions with either *VP1* or *VP4*.

- *Getting a decision for each variation point and resolving contradicting decisions.* This is an iterative step. For each variation point identified, the analyst is asked to make a decision. A decision is about selecting, for the product, variant use cases in the variation point. After the analyst makes the decision, the algorithm first checks the associated cardinality constraints. The use case diagram is then traversed to determine previous decisions contradicting the current decision. If there is any contradiction, the analyst is expected to update one or more decisions to resolve the contradiction.
  - The analyst makes a decision in *VP4*, which is selecting *UC9* for the product. The decision complies with the cardinality constraint. No contradiction is identified since there is no previous decision. Therefore, the decision is saved in the decision model.
  - The analyst proceeds with *VP1*. *UC2* and *UC4* are selected while *UC3* is not selected for the product. The decision complies with the cardinality constraint in *VP1*. The algorithm identifies a contradiction with the decision in *VP4*. Since the user does not select *UC3*, *UC7* and *UC8* in *VP3* included by *UC3* are also automatically not selected. However, *UC9* requires *UC8* in the product. The analyst is asked to resolve the contradiction by updating either the decision for *VP4* or the decision for *VP1*.
  - The analyst updates the decision in *VP1* to resolve the contradiction. Only *UC2* and *UC3* are selected in the updated decision, which complies with the cardinality constraint. At this point, there is no contradiction identified. *UC3* is already selected but the analyst has not yet made the decision for *VP3*. Therefore, the decision is saved in the decision model.
  - The analyst is asked to make further decisions for the variation points included by the selected variant use cases. *UC2* and *UC3* include *VP2* and VP3, respectively. In the decisions for *VP2* and *VP3*, the analyst selects *UC6* and *UC8*. The decisions comply with the associated cardinality constraints, and there is no contradiction identified with the previous decisions. While *UC6* is selected, the user decides not to have *UC7* (UC6 conflicts with UC7 in Fig. 9). *UC9* and *UC8* are selected in *VP4* and *VP3*, respectively (UC9 requires UC8). The decisions are saved in the decision model. All variation points are addressed after the analyst selects *UC2*, *UC3*, *UC6*, *UC8*, and *UC9*.
- *Generating the PS use case diagram from the PL diagram.* By using the decisions stored in the decision model, the algorithm automatically generates the PS use case diagram from the PL diagram (see Fig. 10 for the PS diagram generated from Fig. 9). The transformation is based on a set of transformation rules further described in Section 8. For instance, for *UC1*, *VP1* and the selected *UC2* in Fig. 9, the algorithm creates *UC1* and *UC2* with an *include* relation in Fig. 10.

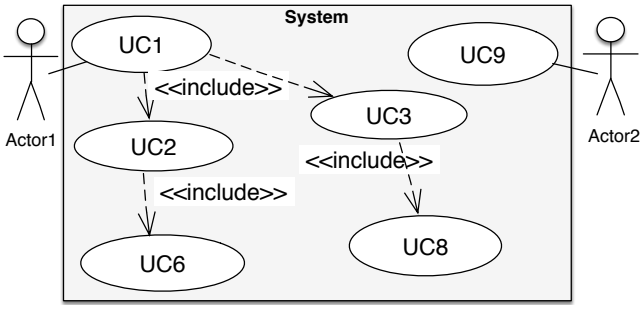


Fig. 10 Generated Product Specific Use Case Diagram

One may argue that the *include* relations with the single, including use cases in Fig. 10 are redundant because an *include* relation is used to show that the behavior of an included use case is inserted into the behavior of multiple including use cases. Alternatively, we could choose an approach which directly inserts (copy-and-paste) the behavior of the included use cases into the including use case specifications. However, to ease the traceability between PL and PS use case diagrams, we prefer to employ include relations in the PS use case diagram even if they are for single, including use cases. For instance, *UC2*, *UC3*, *UC6* and *UC8* in Fig. 10 can be directly traced to the variant use cases *UC2*, *UC3*, *UC6* and *UC8* in Fig. 9, respectively.

After the PL use case diagram, the algorithm handles the PL use case specifications. Table 3 provides the PL specifications of some use cases in Fig. 9. The algorithm has two steps for use case specifications:

Table 3 Some Example Specifications of the PL Use Cases in Fig. 9

1	**USE CASE** UC1
2	**1.1 Basic Flow**
3	1. The system VALIDATES THAT the operating status is valid.
4	2. The system REQUESTS the measurements FROM the sensors.
5	3. INCLUDE <VARIATION POINT: VP1>.
6	4. The system VALIDATES THAT the computed value is valid.
7	5. The system SENDS the computed value 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 1
15	1. ABORT.
16	**1.4 Specific Alternative Flow**
17	RFS 4
18	1. The system increments the counter by the increment step.
19	2. ABORT.
20	
21	<**VARIANT**>**USE CASE** UC2
22	**1.1 Basic Flow**
23	V1. The system SENDS measurement errors TO the STO Controller.
24	V2. <OPTIONAL>The system VALIDATES THAT RAM is valid.
25	V3. The system VALIDATES THAT the sensors are valid.
26	4. The system VALIDATES THAT there is no error detected.
27	**1.2 Specific Alternative Flow**
28	RFS V2
29	1. ABORT.
30	**1.3 Specific Alternative Flow**
31	RFS V3
32	1. <OPTIONAL>The system SENDS diagnosis TO the STO Controller.
33	2. ABORT.
34	**1.4 Specific Alternative Flow**
35	RFS 4
36	1. INCLUDE <VARIATION POINT: VP2>.
37	2. ABORT.

– *Getting decisions for each optional step, optional alternative flow, and variant order group.* In Table 3, there are two variation points (Lines 5 and 36), one variant use case (Lines 21-37), two optional steps (Lines 24 and 32), one optional alternative flow (Lines 8-12), and one variant order group (Lines 23-25). The decisions for variant use cases have already been made in the PL diagram (selecting *UC2* and *UC3* in *VP1* and *UC6* in *VP2*). Therefore, in this step, the analyst is only asked to make decisions for optional steps, optional alternative flows, and variant order groups. For example, the user selects only one of the optional steps (Line 24) with the order *V2*, *V1*, and *V3* (Lines 23-25). The optional bounded alternative flow is not selected. These decisions are saved in the decision model.

Table 4 Some of the Generated PS Use Case Specifications

1	**USE CASE** UC1
2	**1.1 Basic Flow**
3	1. The system VALIDATES THAT the operating status is valid.
4	2. The system REQUESTS the measurements FROM the sensors.
5	3. The system VALIDATES THAT 'Precondition of UC2'.
6	4. INCLUDE UC2.
7	5. The system VALIDATES THAT the computed value is valid.
8	6. The system SENDS the computed value TO the STO Controller.
9	**1.2 Specific Alternative Flow**
10	RFS 1
11	1. ABORT.
12	**1.3 Specific Alternative Flow**
13	RFS 3
14	1. INCLUDE UC3.
15	2. RESUME STEP 5.
16	**1.4 Specific Alternative Flow**
17	RFS 5
18	1. The system increments the counter by the increment step.
19	2. ABORT.
20	
21	**USE CASE** UC2
22	**1.1 Basic Flow**
23	1. The system VALIDATES THAT RAM is valid.
24	2. The system SENDS measurement errors TO the STO Controller.
25	3. The system VALIDATES THAT the sensors are valid.
26	4. The system VALIDATES THAT there is no error detected.
27	**1.2 Specific Alternative Flow**
28	RFS 1
29	2. ABORT.
30	**1.3 Specific Alternative Flow**
31	RFS 3
32	1. ABORT.
33	**1.4 Specific Alternative Flow**
34	RFS 4
35	1. INCLUDE UC6.
36	2. ABORT.

– *Generating PS use case specifications from PL use case specifications.* The PL use case specifications are automatically transformed into the PS specifications based on configuration decisions and a set of transformation rules (see Section 8). Table 4 provides some of the PS use case specifications generated from Table 3. First, some of the variation points in the PL specifications (Lines 5 and 36 in Table 3) are transformed based on the decisions for the PL diagram. For instance, for *VP1*, the configurator creates two include statements for *UC2* and *UC3* (Lines 6 and 14 in Table 4) with a validation step (Line 5 in Table 4) and a corresponding specific alternative flow where *UC3* is included (Lines 12-15). The created validation step checks if the precondition of *UC2* is met. If the condition holds, *UC2* is executed in the basic flow (Line 6). If not, the newly created alternative flow is taken and *UC3* is executed (Line 14). For *VP2*, only *UC6* is included in the PS specification of *UC2* since the

user selects only *UC6*. After handling variation points, selected optional steps and optional alternative flows are included in the PS specifications (Line 23). Variant order groups are ordered in the PS specifications according to the given decision (Lines 23-26).

Lastly, the configuration algorithm collects decisions for each optional and variant domain entities in order to generate the PS domain model from the PL domain model. The algorithm does not currently take into account the variant dependencies in PL domain models. It would need further extensions for detecting configuration decisions violating constraints imposed by variant dependencies. The detection of such decisions in PL domain models is very similar to the consistency checking of configuration decisions in PL use case diagrams (see Section 7.3). Therefore, we would only need to adapt the current consistency checking algorithm for PL domain models. The PS domain model generation is straightforward. In addition to mandatory entities, each selected optional and variant entity is copied into the PS domain model without product line stereotype. The generated model is pruned for the cases where a *is-a* relation has only a single subclass. Fig. 11 provides the PS domain model generated from the PL domain model in Fig. 7.

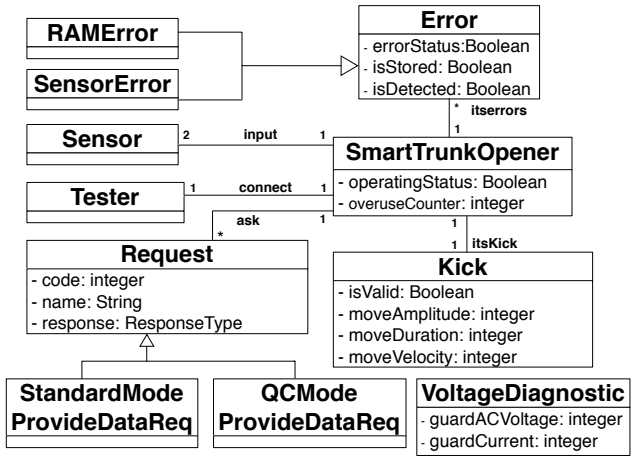


Fig. 11 Generated Product Specific Domain Model

The PL domain model contains two variant entities (*ClearErrorStatusRequest* and *DiagnosticModeProvideDataReq*) and one optional entity (*VoltageDiagnostic*). The analyst selects only *VoltageDiagnostic* for the PS domain model. Therefore, only *VoltageDiagnostic* among variant and optional entities is copied into the PS domain model in Fig. 11. In the PL model, there are two main request types, i.e., *ClearErrorStatusRequest* and *ProvideSystemDataRequest*, with sub request types. Since *ClearErrorStatusRequest* is not selected by the analyst, *ProvideSystemDataRequest* is the only remaining subclass of *Request* in the inheritance hierarchy. During pruning of the PS model, it is removed and the subclasses *StandardModeProvideDataReq* and *QCModeProvideDataReq* become direct specializations of *Request*.

Another output of our configuration approach is a decision model which conforms to the decision metamodel in Fig. 8. The decision model stores all the configuration decisions for the use case diagram, the use case specifications, and the domain model. Fig. 12 depicts the decision model resulting from the example configuration using the PL use case diagram in Fig. 9, the PL use case specifications in Table 3, and the PL domain model in Fig. 7.

The decision model in Fig. 12 contains several instances of *DomainEntity*, *VariationPoint*, *UseCase* for which the analyst made the decisions described above. For instance, the *VP1* instance of *MandatoryVariationPoint* is associated with three instances *UC2*, *UC3* and *UC4* of *VariantUseCase*, while the decision for the variation point is encoded as *True* or *False* using the *isSelected* attribute. The decision models can be employed for further use such as representing decisions for reconfiguration of the same product and comparison of the decisions in multiple products for regression test selection within the context of product line testing, as further described in Section 11.

7 Consistency Checking of Configuration Decisions

In this section, we present the details of the consistency checking of configuration decisions in the PL use case diagram. The objective of consistency checking (Section 7.1) is to identify contradicting decisions for variation points in a configurable PL use case diagram (Section 7.4). The consistency checking algorithm (Section 7.3) is based on mapping from the PL use case diagram to propositional logic (Section 7.2).

7.1 Objective and Assumptions

Consistency checking of configuration decisions is vital at collecting decisions from the analyst. PS models cannot be generated from inconsistent decisions. In our configuration approach, consistency checking aims at determining contradicting decisions for the variation points in the PL use case diagram. Two or more configuration decisions may contradict each other if they result in violating some variation point and variant dependency constraints (i.e., *require* and *conflict*). 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.

The analyst needs to update decisions in order to resolve contradictions. Our approach follows the *Fix right away with selective (multiple) undo* strategy [58] in which only involved

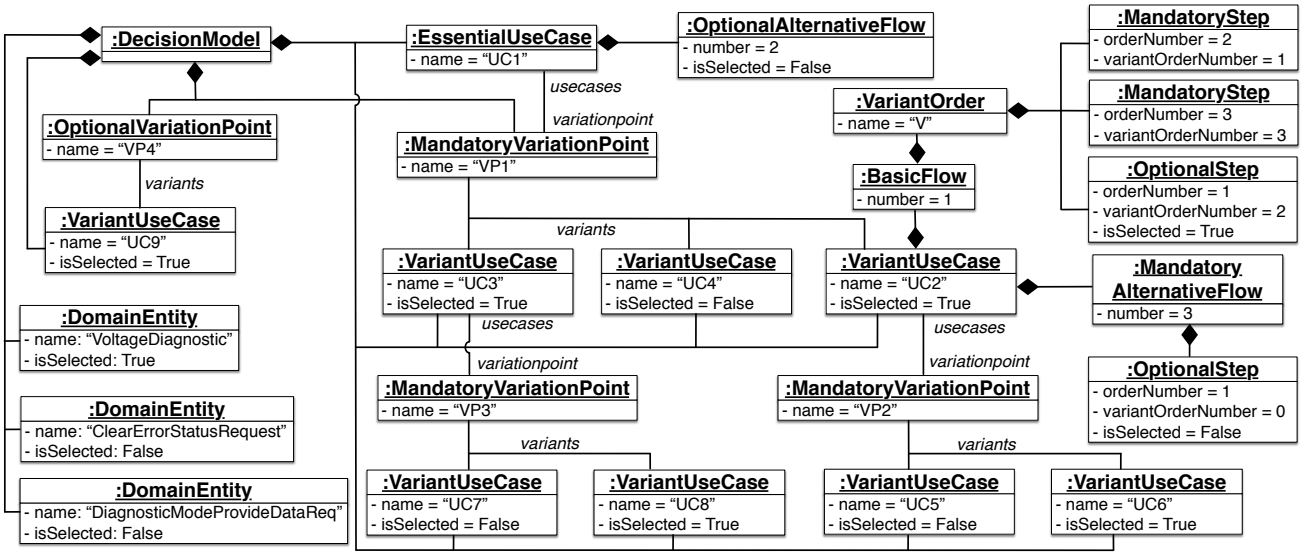


Fig. 12 Example Decision Model Resulting from the Example Configuration

decisions are updated to return the configuration to a consistent state immediately when the analyst introduces a contradicting decision. To do that, we automatically identify the decisions involved in the contradiction. We chose this strategy based on our discussions with IEE analysts because each time a decision is made during configuration with customers, analysts would like to keep decisions in the PL diagram consistent and not to have multiple contradictions at a time. Tolerating contradictions during decision-making makes it hard to communicate with untrained customers for reasoning on decisions and resolving contradictions. For the resolution of decision contradictions in the PL use case diagram, our underlying assumption is that the diagram is configurable. A PL use case diagram is configurable if at least one valid PS use case diagram can be generated from the PL diagram. It may not be configurable because of an incorrect combination of variant and variation point dependencies (see Section 7.4).

7.2 Propositional Logic Mappings

Our consistency checking algorithm is based on 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 $PLD = \{u_1, ..., u_n\}$; 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. Figure 13 gives the corresponding propositional formulas for each pattern involving dependencies, variation points, and

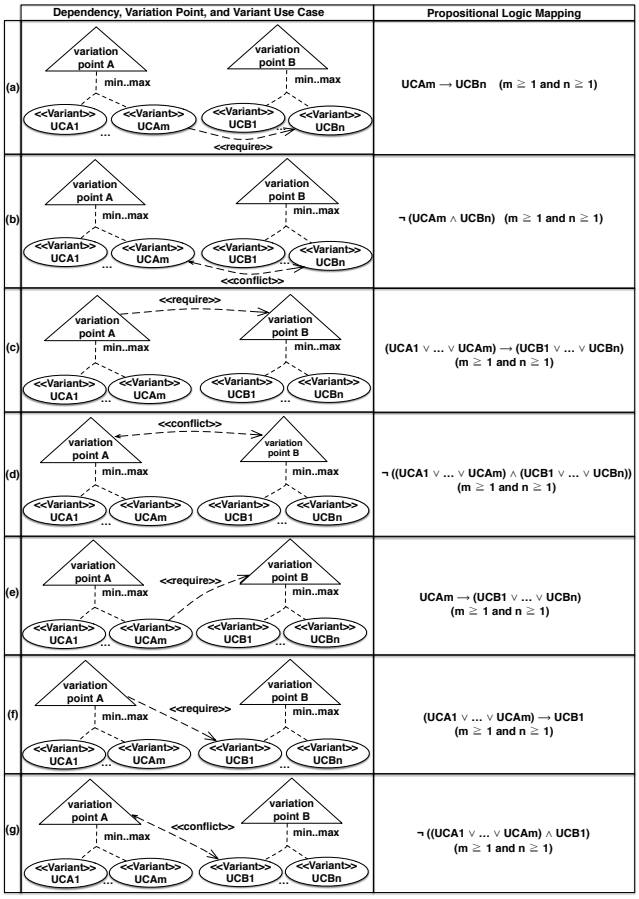


Fig. 13 Mapping from PL Use Case Diagram to Propositional Logic

variant use cases, where propositions capture logical relationships among variant use cases. For instance, according to the corresponding propositional formula in Fig. 13(a), if use case UCA_m is selected for a product then the selection logically implies that use case UCB_n is also se-

lected. Fig. 13(c) depicts the mapping when there is a *require* dependency between two variation points A and B. In such a case, if at least one of the variant use cases in variation point A ($UCA_1 \vee \ ... \ \vee UCA_m$) is selected, then at least one of the variant use cases in variation point B ($\rightarrow UCB_1 \vee \ ... \ \vee UCB_n$) should also be selected.

In a SAT solver based approach, for the entire PL use case diagram, one propositional formula can be formed as a conjunction of formulas derived from each dependency in the diagram using the mapping. 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 [13]. This approach may not be feasible for complex industrial projects when the number of variables to be computed is large. Therefore, to determine contradicting decisions, we follow an approach different than determining if there exists an interpretation that satisfies a propositional formula derived from the entire PL use case diagram (Section 7.3).

7.3 Consistency Checking Algorithm

For a given decision regarding a variation point in the PL diagram, our approach only checks the satisfaction of the propositional formulas derived from its dependencies. The number of variables taken into account in such approach is much smaller than the number of variables derived from the entire diagram to be computed by an approach using SAT solvers. Assume that we have two variant use cases *Ua* and *Ub* where *Ua requires Ub* and *Ua* is selected. The corresponding propositional formula in Fig. 13(a) is not satisfied only if *Ub* is unselected in prior decisions and there is no other further decision to be made for *Ub*. Therefore, we only check if *Ub* is unselected and cannot be selected in further decisions. If there is no decision made for *Ub* yet, we do not need to check if the corresponding formula is satisfied. The satisfaction of the formula is checked only if there is a valuation of the variable in the formula for Ub based on configuration decisions. However, decisions for other variant use cases might imply a decision for *Ub*. Our approach automatically infers those implicit decisions to be taken into account in the valuation of formulas.

Alg. 1 describes the part of our configuration algorithm related to consistency checking. For each new decision made by the analyst, the algorithm checks if the formulas derived for the decided use case elements are satisfied. If the formulas are not satisfied, the algorithm returns contradicting decisions to the analyst. The analyst updates the decisions to resolve the contradiction. In order to illustrate the algorithm, we rely on the example, contradicting decisions in Fig. 9.

The analyst makes a decision d for each variation point vp, which is either included by an essential use case uc or not included by any use case (Lines 2, 4 and 5). For each new decision d, the algorithm checks if there is any contradicting, prior decision. Decision d is a quadruple of variation point vp, essential use case uc including vp, set of selected variant use cases SUC in vp, and set of unselected variant use cases $NSUC$ in vp (Line 8). For the decisions in *VP1* and *VP4* in Fig. 9, d is ($VP1$, $UC1$, $\{UC2, UC4\}$, $\{UC3\}$) and ($VP4$, $null$, $\{UC9\}$, $\emptyset$), respectively. The algorithm first determines if d complies with the cardinality constraint in vp (Line 9). If the cardinality constraint is satisfied, the algorithm checks if d contradicts any prior decision (Lines 10-29); otherwise, the analyst is asked to update decision d for the cardinality constraint (Line 31). We call some check functions (i.e., *checkConflictingVP*, *checkRequiringVP*, *checkRequiredVP*, *checkConflictingUC*, *checkRequiringUC*, and

Alg. 1: Part of *config*

Inputs : PL use case diagram, PLD,
Set of PL use case specifications, PLS,
PL domain model, $PLDM$
Output: Set of PS use case models

1 Let DC be the empty set for completed decisions;
2 Let L be the set of pairs of variation points vp and use cases uc such that use cases are essential and they include the variation points, or the variation points are not included by any use case;
3 $T \leftarrow L$;
4 **while** $L \neq \emptyset$ **do**
5 $dp \in L$;
6 Let SUC be the set of variant use cases selected in $dp.vp$;
7 Let $NSUC$ be the set of variant use cases unselected in $dp.vp$;
8 Let d be the quadruple $(dp.vp, dp.uc, SUC, NSUC)$;
9 **if** *(d satisfies cardinality constraints in $dp.vp$)* **then**
10 Let C be the empty set for contradicting decisions;
11 $C \leftarrow$ **checkConflictingVP**($dp.vp$, DC, d, PLD);
12 $C \leftarrow C \cup$ **checkRequiringVP**($dp.vp$,DC,d,PLD);
13 $C \leftarrow C \cup$ **checkRequiredVP**($dp.vp$,DC,d,PLD);
14 **foreach** *($u \in SUC$)* **do**
15 $C \leftarrow C \cup$ **checkConflictingUC**(u,DC,d,PLD);
16 $C \leftarrow C \cup$ **checkRequiringUC**(u,DC,d,PLD);
17 **end foreach**
18 **foreach** *($u \in NSUC$)* **do**
19 $C \leftarrow C \cup$ **checkRequiredUC**(u, DC, d, PLD);
20 **end foreach**
21 **if** *($C = \emptyset$)* **then**
22 $DC \leftarrow DC \cup \{d\}$;
23 $L \leftarrow L \backslash \{dp\}$;
24 Let $new_p = \{(vp, uc) \mid uc$ ***includes*** $vp \wedge uc \in SUC \wedge (vp, uc) \notin T\}$;
25 $L \leftarrow L \cup new_p$;
26 $T \leftarrow T \cup new_p$;
27 **else**
28 **updateDecisions**($C \cup \{d\}$);
29 **end if**
30 **else**
31 **updateDecisions**($\{d\}$);
32 **end if**
33 **end while**
34 **...**

checkRequiredUC) to determine whether the propositional logic formulas, derived from the dependencies to/from the diagram elements decided in d, are satisfied by d and set of prior decisions DC (Lines 10-20). If there is a formula not satisfied, there is at least one prior decision that is contradicting d. The algorithm reports contradicting decisions to be updated by the analyst in the *updateDecisions* function (Line 28). If there is no contradicting decision, d is approved and considered *completed* (Lines 21-23). The selected variant use cases may include variation points. The pairs of those variation points and their including use cases are considered for further decisions (Lines 24-26). Each *check* function in Alg. 1 checks the propositional formulas in one or more mappings in Fig. 13.

- ***checkConflictingVP*** checks the formulas for selected variation point vp conflicting with a variation point or a variant use case (Fig. 13(d) and (g)),
- ***checkConflictingUC*** checks the formulas for selected variant use case u conflicting with a variation point or a variant use case (Fig. 13(b) and (g)),
- ***checkRequiringVP*** checks the formulas for selected variation point vp requiring a variation point or a variant use case (Fig. 13(c) and (f)),
- ***checkRequiredVP*** checks the formulas for unselected variation point vp required by a variation point or a variant use case (Fig. 13(c) and (e)),
- ***checkRequiringUC*** checks the formulas for selected variant use case u requiring a variation point or a variant use case (Fig. 13(a) and (e)),
- ***checkRequiredUC*** checks the formulas for unselected variant use case u required by a variation point or a variant use case (Fig. 13(a) and (f)).

In Fig. 9, the decision in $VP1$ contradicts the prior decision in $VP4$ because of the *require* dependency. This is determined by the function *checkRequiredUC*, whose algorithm is presented in Alg. 2. For the rest of the *check* functions, the reader is referred to Supplementary Material.

Alg. 2 checks the formulas in Fig. 13(a) and (f) for an unselected variant use case required by a variation point or a variant use case. For instance, in Fig. 13(a), when $UCBn$ is unselected and there is no further decision made for $UCBn$, it checks if $UCAm$ is selected in any prior decision. If $UCAm$ is already selected, it reports a contradiction.

Alg. 2 takes as input use case uc unselected in decision d, set of prior decisions DC, decision d, and PL use case diagram PLD, while it returns the set of decisions contradicting d. The inputs of *checkRequiredUC* for the example in Fig. 9 are $UC3$, $\{D1\}$, $D2$, and the PL diagram in Fig. 9 where $D1 = (VP4, null, \{UC9\}, \emptyset)$ and $D2 = (VP1, UC1, \{UC2, UC4\}, \{UC3\})$. The functions used in Alg. 2 are the following:

Alg. 2: *checkRequiredUC*

Input : Use case, uc, Set of decisions, DC, Decision, d, PL use case diagram, PLD
Output: Set of contradicting decisions

```
1  Let RES be the empty set for contradicting decisions;
2  if ((uc is not selected in the completed decisions) and (there is
   no further decision to be made for uc)) then
3      Let EX be the empty set for inferred, unselected elements;
4      EX ← {uc} ∪ inferUnselectedElements(uc, d, DC, ∅, PLD);
5      foreach (dm ∈ DC) do
6          Let SUC be the set of selected use cases in dm;
7          Let vp be the variation point in dm;
8          if (SUC ≠ ∅) and ((EX ∩ getRequiredElements(vp,
           PLD))≠ ∅) then
9              RES ← RES ∪ {dm};
10         end if
11         foreach (u ∈ SUC) do
12             Let I be the empty set for inferred, selected elements;
13             I ← {u} ∪ inferSelectedElements(u, dm,
               DC ∪ {d}, ∅, PLD);
14             foreach (e ∈ I) do
15                 if (EX ∩ getRequiredElements(e, PLD) ≠ ∅)
                   then
16                     RES ← RES ∪ getInvolvedDecisions(e,
                       dm, d, DC);
17                 end if
18             end foreach
19         end foreach
20     end foreach
21     return RES;
22 else
23     return RES;
24 end if
```

- ***inferUnselectedElements*** infers unselected elements for use case uc unselected in decision d (Line 4),
- ***inferSelectedElements*** infers selected elements for use case u selected in decision dm (Line 13),
- ***getRequiredElements*** returns use cases and variation points required by other variation points or variant use cases (vp in Line 8 and e in Line 15),
- ***getInvolvedDecisions*** returns all decisions contradicting decision d for element e in decision dm (Line 16).

Alg. 2 starts with checking whether uc, unselected in decision d, is also unselected in the set of prior decisions DC, while it is also not possible to make any further decision for uc (Line 2). If uc is already selected in another decision or if there is still yet another decision to be made for uc, the function returns an empty set of contradicting decisions (Line 23); otherwise, the function checks whether there is any selected use case which requires uc (Lines 3-21). For $UC3$, the decision can be made only via the pair ($VP1$, $UC1$) since $UC1$ is the only use case including $VP1$. $D2$ is the decision made via the pair ($VP1$, $UC1$) where $UC3$ is unselected.

When uc is unselected in d, there might be other variant use cases automatically unselected. These use cases are in the variation points included by uc (Line 4). The function $inferUnselectedElements$ returns the inferred, unselected use cases and variation points for use case uc unselected in decision d. For $D2$, it infers $UC7$, $UC8$, and $VP3$ ($EX = \{UC3, UC7, UC8, VP3\}$). $VP3$ is included only by $UC3$. $UC7$ and $UC8$ in $VP3$ cannot be selected after $UC3$ is unselected in $D2$. Since all variant use cases in $VP3$ are automatically unselected, $VP3$ is also considered *unselected*. If any of these elements in the set of unselected elements EX is required by a variation point selected in prior decision dm in DC, the current decision d contradicts dm (Lines 6-10). The variant use cases selected in dm (SUC in Line 6) might cause other variant use cases automatically to be selected. These are the use cases with a mandatory variability relation in the variation points included by the use cases selected in dm (Line 13). The function *inferSelectedElements* infers those variant use cases to check whether they require any unselected element in EX (Lines 11-19). For $D1$, there is no inferred variant use case. There is only $UC9$ which is selected for $VP4$ in $D1$ ($I = \{UC9\}$ for $u = UC9$). Only $UC9$, selected in $D1$, requires $UC8$ in EX (Line 15). There might be other prior decisions contributing to the selection of $UC9$. The function *getInvolvedDecisions* returns all these decisions (Line 16). There is only $D1$ in which $UC9$ is selected. Therefore, the function *checkRequiredUC* returns only $D1$ in the set of contradicting decisions ($RES = \{D1\}$ in Lines 16 and 21).

7.4 Non-configurable PL Use Case Diagrams

As stated in Section 7.1, our assumption for consistency checking is that the PL diagram is configurable. Fig. 14 provides an example of a non-configurable PL diagram.

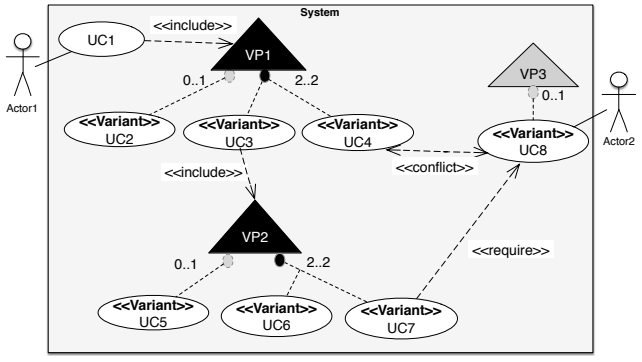


Fig. 14 An Example of a Non-Configurable PL Use Case Diagram

Essential use case *UC1* includes mandatory variation point *VP1* which has three variant use cases through optional and mandatory variability relations. Variant use cases *UC3* and *UC4* in *VP1* are always selected because of the mandatory variability relation (cardinality constraint '*2..2*' in *VP1*). *UC3* includes another mandatory variation point, *VP2*, which has three variant use cases *UC5*, *UC6* and *UC7*. *UC6* and *UC7* are always selected because of the mandatory variability relation (cardinality constraint '*2..2*' in *VP2*). Therefore, variant use cases *UC3*, *UC4*, *UC6*, and *UC7* are automatically selected for every product. *UC7 requires* another variant use case, *UC8*, while *UC8 conflicts* with *UC4*. The *require* dependency implies that if *UC7* is selected for a product, *UC8* should also be selected for the same product. Since *UC7* is automatically selected for every product, *UC8* should always be selected for every product. On the other hand, *UC8* cannot be selected for any product because it conflicts with *UC4*, which is also automatically selected for every product. Therefore, it is not possible to generate a valid PS use case diagram from the PL diagram in Fig. 14. The combination of the dependencies *require* between *UC8* and *UC7*, *conflict* between *UC8* and *UC4*, and *include* between *UC3* and *VP2* and the cardinality constraints (*2..2*) in *VP1* and *VP2* is the reason of non-configurability in Fig. 14. Such combination of dependencies and cardinality constraints in a non-configurable PL use case diagram needs to be resolved before making configuration decisions. Otherwise, in the non-configurable PL diagram, there will always be contradicting decisions which are impossible to resolve.

The detection of non-configurable models has been addressed in the context of feature modeling and product line requirements specifications [14, 70, 69]. There are also techniques [40, 39, 41] to identify incorrect combinations of requirements dependencies in a broader context. Existing techniques (e.g., [62, 40, 68, 75, 49, 50, 65, 31]) could be adapted for PL use case diagrams in our configuration approach as part of Step 1, *Elicit Product Line Use Case and Domain Models*, in Fig. 4. Before making decisions, the analyst could automatically check if the PL use case diagram is configurable and, if necessary, could resolve the incorrect combination of dependencies and cardinality constraints.

8 Generation of PS Use Case Models

After the decisions are made, the PS use case and domain models are generated from the PL models. The generation of PS models are implemented in the *PL-PS Transformer* component in *PUMConf* (Fig. 15). The details of the component architecture of *PUMConf* are given in Section 9.

The *PL-PS Transformer* contains three subcomponents: *Diagram Transformer*, *Specification Transformer* and *Domain Model Transformer*. The subcomponents are *update-in-place* transformations [29] in which a model is both an input and output. Each subcomponent takes one of the PL models and relevant decisions in the decision model as input, and produces the corresponding PS model as output.

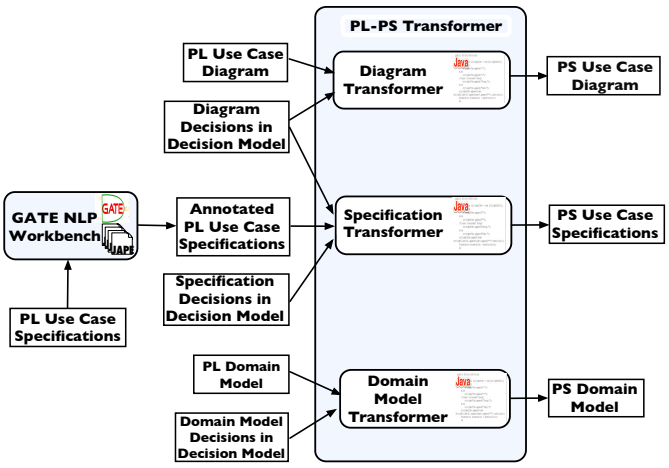


Fig. 15 Overview of the PS Use Case and Domain Model Generation

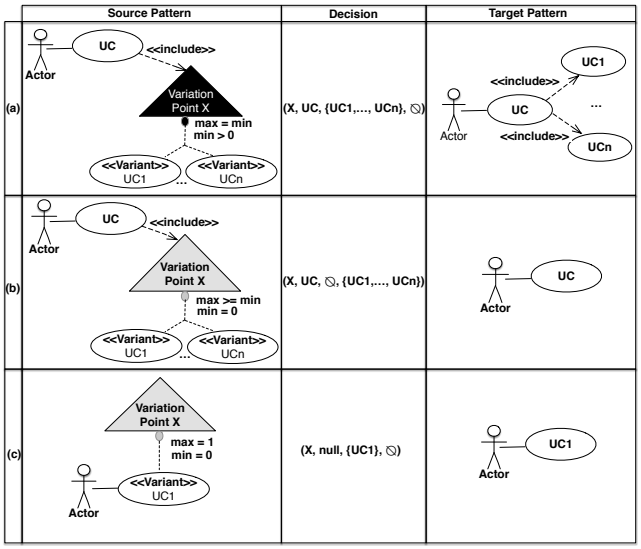


Fig. 16 Example Source and Target Patterns with Decisions for Use Case Diagrams

All PL and PS use case specifications are stored as plain text in the native IBM DOORS format. During the consistency checking of use case and domain models (see Step 2 in Fig. 4), for the RUCM keywords and types of steps, use case specifications in IBM DOORS are annotated using NLP provided by the GATE workbench (see Section 9). The *Specification Transformer* uses the annotations to distinguish RUCM steps and types of alternative flows in matching transformation rules (Section 9). It processes the plain text instead of instances of the RUCM metamodel [81]. Compared to model transformation languages, Java provides much more flexibility for handling annotated plain text in terms of loading, matching and editing the text. Therefore, we used Java to implement the *Specification Transformer*. To provide the uniformity in the *PL-PS Transformer*, other subcomponents are also implemented in Java.

The *Diagram Transformer* takes the PL use case diagram and diagram decisions as input, and generates the PS use case diagram as output. The PS diagram generation is based on mappings between patterns in PL and PS diagrams driven by decisions. For instance, for a variant use case in the PL diagram, there is a corresponding use case in the PS diagram only if the variant use case is selected during decision-making. Fig. 16 gives example source and target patterns with decisions for use case diagrams.

Fig. 16 has three columns, i.e., *source pattern*, *decision*, and *target pattern*, to represent example mappings between PL and PS diagrams. Decisions for the diagrams are represented as quadruples (see Section 7). In Fig. 16(a), the analyst selects all variant use cases in mandatory variation point X included by essential use case UC. The variation point, *include* relation and unselected variant use cases are removed while each selected variant use case is transformed into a use case included by UC in the PS diagram. In Fig. 16(b), optional variation point X is included by essential use case UC while all variant use cases are unselected. Only essential use case X is kept in the PS diagram. The source pattern in Fig. 16(c) represents optional variation point X which has one variant use case (i.e., $UC1$). X is not included by any other essential or variant use case. When $UC1$ is selected, only variation point X is removed from the PS diagram.

The *Specification Transformer* takes both diagram and specification decisions to generate PS use case specifications (see Fig. 17 for example mappings between PL and PS specifications). Diagram decisions are used to transform use case steps where variation points are included. In Fig. 17(a), the source pattern represents an example specification where the variation point X in Fig. 16(a) is included in between other use case steps. For the same decision, represented as a quadruple, in Fig. 16(a) where all variant use cases are selected, the *VALIDATES THAT* and *INCLUDE* steps in the basic flow and a set of alternative flows are generated for the PS specification. One of the variant use cases ($UC1$) is executed in the basic flow (the step where $UC1$ is included) if its precondition holds (the *VALIDATES THAT* step). The rest of the selected use cases are executed in the generated alternative flows. One of the alternative flows is taken if the *VALIDATES THAT* statement in the basic flow fails. In Fig. 17(b), the source PL specification for Fig. 16(b) is transformed into the PS specification based on the diagram decision where no variant use case is selected. The step where the variation point X is included is removed for the PS specification.

Fig. 17(c) represents an example source pattern which contains multiple optional steps in a variant order. The analyst has to decide which optional steps are retained and in which order in the PS specification. The specification decision for each optional step is represented as a triple of the step name, a boolean variable which is true when the step is

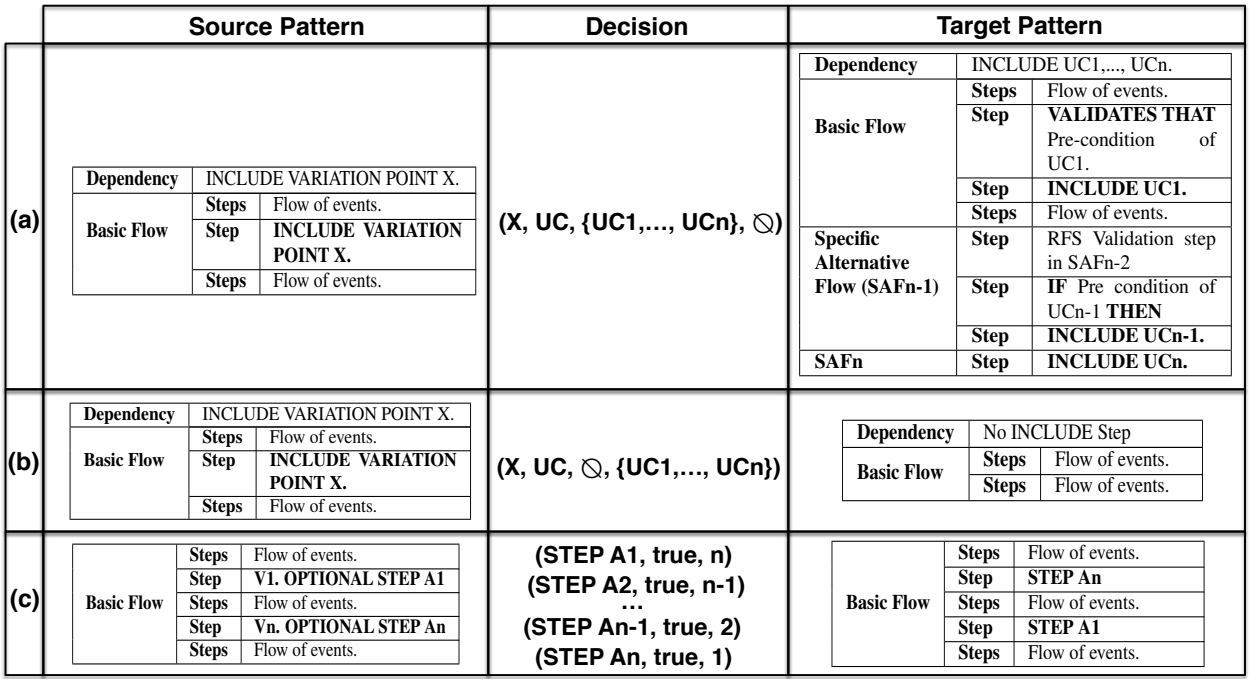


Fig. 17 Example Source and Target Patterns with Decisions for Use Case Specifications

selected, and the decided order number. In the example, each optional step is selected in a reverse order for the PS specification (see the decision column in Fig. 17(c)). Based on the decisions, the selected optional steps (*OPTIONAL STEP* A_1, ..., *OPTIONAL STEP* A_{n-1}, *OPTIONAL STEP* A_n) are preserved in the reverse order (*STEP* A_n, *STEP* A_{n-1}, ... , *STEP* A_1) in the target pattern while there might be common steps in between the selected optional steps.

The *Domain Model Transformer* takes the PL domain model and domain model decisions as input to generate a PS domain model. As we discussed in Section 6, the generation of a PS domain model is straightforward: in addition to mandatory entities, all selected optional and variant domain entities are kept, while their PL stereotypes are removed. The *is-a* relations having only a single subclass are also removed to prune the generated model. In the next section, we present the tool support and architecture, which provide more detailed information about the interaction of the *PL-PS Transformer* with other components of the tool.

9 Tool Support

We have implemented our configuration approach in a prototype tool, *PUMConf (Product line Use case Model Configurator)*. Section 9.1 provides the layered architecture of the tool while we describe the tool features with some screenshots in Section 9.2. For more details and accessing the tool executables, see: `https://sites.google.com/site/pumconf/`.

9.1 Tool Architecture

The tool architecture is composed of three layers (see Fig. 18): (i) the *User Interface (UI) layer*, (ii) the *Application layer*, and (iii) the *Data layer*.

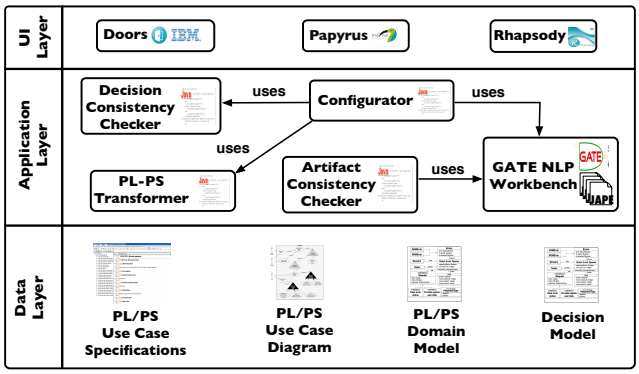


Fig. 18 Layered Architecture of PUMConf

User Interface (UI) Layer. This layer supports the activity of eliciting product line use cases and domain models to create or update the PL artifacts (see Fig. 4). It also enables the viewing of the generated PS artifacts. We employ IBM Doors (`www.ibm.com/software/products/ca/en/ratidoor/`) for use case specifications, Papyrus (`https://www.eclipse.org/papyrus/`) for use case diagrams, and IBM Rhapsody (`www.ibm.com/software/products/en/ratirhapfami`) for domain models. IBM Doors does not put any restriction on the structure of use cases and thus allows the adoption of the RUCM template. Halmans and Pohl [45] do not provide any metamodel or UML profile

for the PL use case diagram extensions. Therefore, we implemented our own UML profile in Papyrus to enable the use of the Papyrus model editor for creating and editing PL use case diagrams. Our choice of IBM Rhapsody for domain models is based on the modeling practice at IEE, which has been based on Rhapsody. For domain modeling, we could also employ the Papyrus model editor which we use to present the decision model.

Application Layer. This layer supports the main activities of our approach in Fig. 4: *checking consistency of PL use case and domain models* and *configuring PS use case and domain models*. It contains four main components implemented in Java: *Configurator*, *Artifact Consistency Checker*, *Decision Consistency Checker*, and *PL-PS Transformer*. To access these *Application Layer* components through the *UI Layer*, we implemented an IBM DOORS plugin (see Fig. 19).

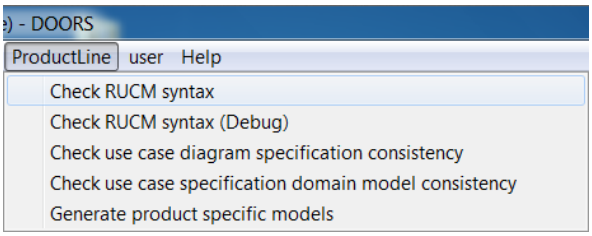


Fig. 19 Menu to Activate IBM DOORS Plug-ins for PUMConf

The *Configurator* component is a coordinator that manages two other components, i.e., *Decision Consistency Checker* and *PL-PS Transformer*. In addition to the *Configurator*, the user has direct access, via the DOORS plug-in, to the *Artifact Consistency Checker* which employs NLP to check the consistency of the PL use case diagram, the use case specifications complying with the RUCM template, and the domain model. The *Decision Consistency Checker* implements the parts of the configuration algorithm where a decision for the PL use case diagram is received from the analyst and its consistency with previous decisions is checked. The algorithm does not attempt to find a valid configuration but simply traverses the diagram for decisions to recursively check whether the implications of any variation point - variant use case dependency (i.e., *include*, *require*, and *conflict*) are violated by the decisions (see Section 7).

The *PL-PS Transformer* component is the Java implementation of the transformation rules for the use case diagram, specifications and domain model. Before the application of the transformation rules, use case specifications need to be annotated by using NLP (see Section 8).

To perform NLP in use case specifications, the *Configurator* and *Artifact Consistency Checker* components use a regular expression engine, called JAPE [43], in the GATE workbench (`http://gate.ac.uk/`), an open-source Natural Language Processing (NLP) framework. We implemented the extended RUCM restriction rules in JAPE. With NLP, use cases are first split into tokens. Second, Part-Of-Speech (POS) tags (i.e., *verb*, *noun*, and *pronoun*) are assigned to each token. By using the RUCM restriction rules implemented in JAPE, blocks of tokens are tagged to distinguish RUCM steps (i.e., *output*, *input*, *include*, and *internal operations*) and types of alternative flows (i.e., *specific*, *alternative*, and *global*). The output of the NLP contains the annotated use case steps. The *Configurator* passes the annotations to the *PL-PS Transformer* in order to match the transformation rules while the *Artifact Consistency Checker* processes these annotations with the use case diagram and domain model to generate the list of inconsistencies among the artifacts. For instance, the consistency of use case specifications and domain model is checked by comparing the use case specification entities identified by the NLP application with the entities in the domain model. For each domain entity identified through NLP, the *Artifact Consistency Checker* generates an entity name by removing all white spaces and putting all first letters following white spaces in capital. If the entity name does not appear either as class name or as an attribute name in the domain model, or if the entity name is only mentioned in the optional parts of use case specifications while it appears as a mandatory entity in the domain model, an inconsistency is reported. The consistency checking could be extended with syntactic and semantic similarity checking techniques [11, 12] to tackle inconsistent naming conventions in the comparison.

Data Layer. All the use case specifications are stored in the native IBM DOORS format while the domain model is exported into the XMI format by the Rhapsody XMI toolkit. The PL use case diagram and the generated PS diagram are stored using the UML profile mechanism, while the decision model is saved in Ecore [2].

9.2 Tool Features

We describe the most important features of our tool: *managing PL use case and domain models*, *checking consistency of PL use case and domain models*, *getting configuration decisions from the analyst*, *checking consistency of decisions*, and *displaying decisions*. These features support the steps of the modeling process given in Fig. 4.

Managing PL use case and domain models. This feature supports Step 1, *Elicit Product Line Use Case and Domain Models*, in Fig. 4. The analyst can create, update, and delete the PL use case diagram, specifications, and domain model by using the selected modeling tools (i.e., IBM Doors, Papyrus, and Rhapsody) adopted in *PUMConf*.

Checking consistency of PL use case and domain models. The consistency of the PL use case and domain models needs to be ensured in Step 2, *Check Consistency of Product Line Use Case and Domain Models*, in Fig. 4 before the analyst makes decisions about the variability information.

Our tool automatically checks (1) if the PL use case specifications conform to the RUCM template and its product line extensions, (2) if the PL use case diagram is consistent with the PL use case specifications, and (3) if the PL domain model is consistent with the PL use case specifications (see Fig. 19). Fig. 20 presents an example output of the consistency checking of the PL use case diagram and specifications in Section 6.

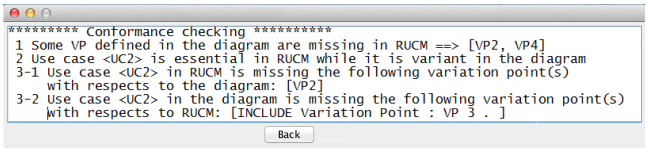


Fig. 20 PUMConf User Interface for Reporting Inconsistencies

Four types of inconsistencies are reported in Fig. 20: missing variation points in the specifications, variant use cases given as essential ones in the corresponding specifications, and missing variation points in the diagram.

Getting configuration decisions from the analyst. During Step 3, *Configure Product Specific Use Case and Domain Models*, in Fig. 4, the tool first determines the list of variation points to be decided, based on the dependency structure of variation points, i.e., *include*. The analyst makes a decision for each variation point in the list, while the tool checks the consistency of the decision with prior decisions. A decision may cause further decisions to be made for some other variation points, i.e., included by the variant use cases selected in the decision. In such cases, after each decision the tool automatically updates the list of variation points to be decided. Fig. 21 presents the user interface for getting the decision for variation point *VP4* in Fig. 9.

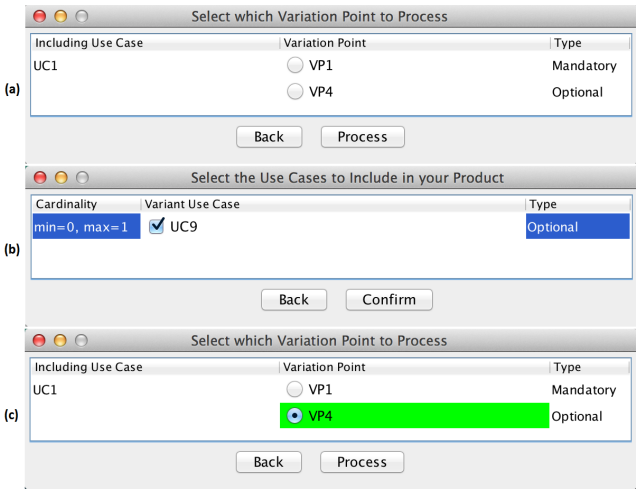


Fig. 21 PUMConf's User Interface for (a) listing variation points driving decisions, (b) selecting variant use cases for the selected variation point, and (c) showing the updated list of variation points after the decision.

In Fig. 21(a), the tool lists the variation points *VP1* and *VP4* but not *VP2* and *VP3* since the analyst can make a decision for *VP2* and *VP3* only after *UC2* and *UC3* are selected in *VP1* (see Fig. 9). The analyst makes a decision in *VP4* by selecting *UC9* in Fig 21(b). After the decision is confirmed to be consistent with previous decisions, *VP4* is highlighted in green, indicating that the decision has been validated and recorded (see Fig. 21(c)).

After the decisions for the PL diagram are made, the analyst proceeds with the PL use case specifications and domain model. Fig. 22 presents the PUMConf's user interface for selecting optional steps in the PL specification of use case $UC2$ in Table 3.

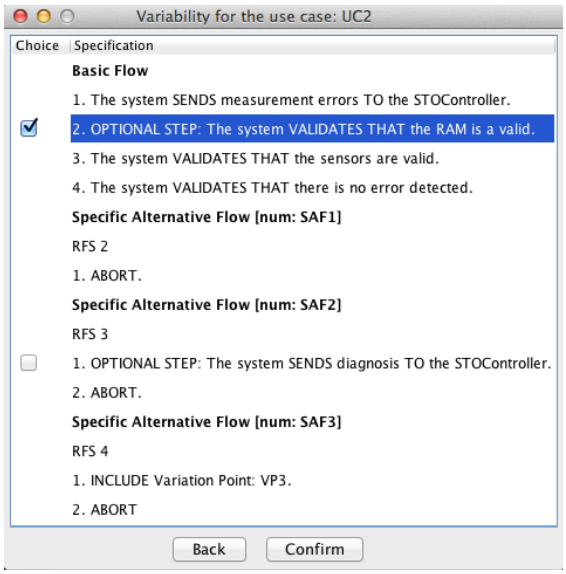


Fig. 22 PUMConf's User Interface for Selecting Optional Steps in Use Cases.

In Fig. 22, the tool lists the entire use case specification including the *optional* tags on optional steps. The analyst makes a decision for each optional step and each optional alternative flow in the specification (none in $UC2$). After these decisions are made, the tool asks the order of the variant order steps if there are any in the specification.

Checking consistency of decisions. After each decision is made in the diagram, the tool checks its consistency with prior decisions. If there is any contradicting decision, the analyst is asked to update the current and/or previous decisions causing the contradiction.

After the decision regarding *VP4* in Fig. 21(b), the analyst proceeds with *VP1* in Fig. 21(c) and then selects *UC2* and *UC4* (Fig. 23(a)). Please note that *UC2* is automatically selected since it is a mandatory variant use case implied by the cardinality constraint in *VP1*. When the analyst submits the decision, the tool automatically checks if it contradicts prior decisions. A contradiction for the decisions in

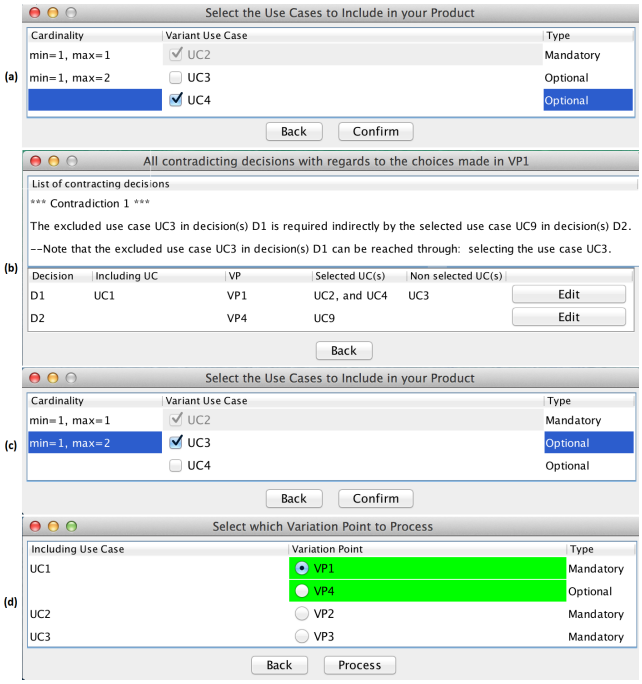


Fig. 23 PUMConf's User Interface for (a) selecting variant use cases for the corresponding variation point, (b) explaining the contradicting decisions, (c) updating the contradicting decision, and (d) showing the updated list of variation points after the updated decision.

VP1 and *VP4* is reported (Fig 23(b)). The upper part of the user interface in Fig. 23(b) provides an explanation for the contradiction while the bottom part lists the decisions involved in the contradiction, with an *Edit* button to update the corresponding decision. To resolve the contradiction in Fig. 23(b), the analyst updates the decision in *VP1* by selecting *UC2* and *UC3* but not *UC4* (Fig. 23(c)). After the decision is updated, the tool checks again for contradictions. It confirms that there is no more contradiction and all decisions are consistent. *UC2* and *UC3* include *VP2* and *VP3*, respectively. Therefore, after *UC2* and *UC3* are selected, the pairs $(UC2, VP2)$ and $(UC3, VP3)$ are automatically given to the analyst to make further decisions (Fig. 23(d)).

Displaying decisions. After the configuration is completed with the generation of the PS use case and domain models, the analyst may need to reconfigure. The tool presents the entire set of decisions for the product with a user interface similar to Fig. 21(a) and (b).

10 Evaluation

In this section, we evaluate our configuration approach via reporting (i) an industrial case study, i.e., STO, to demonstrate its feasibility (Section 10.1), (ii) the results of a questionnaire based survey at IEE aiming at investigating how PUMConf is perceived to address the challenges listed in Section 2 (Section 10.2), and (iii) discussions with the IEE analysts to gather more insights into the benefits and challenges of applying it in an industrial setting (Section 10.3).

10.1 Industrial Case Study

We report our findings about the feasibility of our approach and its tool support in an industrial context. In order to experiment with PUMConf in an industrial project, we applied it to the functional requirements of STO.

10.1.1 Goal

Our goal was to assess, in an industrial context, the feasibility of using PUMConf to improve variability modeling and reuse in the context of use case and domain models. STO was selected for this assessment since it was a relatively new project at IEE with multiple potential customers requiring different features.

10.1.2 Study Context

IEE is a typical supplier in the automotive domain, producing sensing systems (e.g., Vehicle Occupant Classification, Smart Trunk Opener, and Driver Presence Detection) for multiple automotive manufacturers. In IEE's business context, like in many others, use cases are central development artifacts which are used for communicating requirements among stakeholders, such as customers. In other words, in the context of product lines, IEE's software development practices are strongly use case-driven and analysts elicit requirements and produce a new version of use cases for each new customer and product. As a result, IEE needs to adopt PLE concepts (e.g., variation points and variants) to identify commonalities and variabilities early in requirements analysis. These concepts are essential for communicating variability to customers, documenting it for software engineers, and supporting decision making during the elicitation of customer specific requirements [45].

Like in many other environments, the current practice at IEE is based on *clone-and-own* reuse [25]. IEE starts a new product family with an initial customer providing requirements of a single product in the product family. The initial product requirements are elicited and documented as use cases and a domain model which are copied and then maintained for each new customer. Changes are then made manually in the copied models.

IEE provided their initial STO documentation, which contained a use case diagram, use case specifications, and supplementary requirements specifications describing non-functional requirements and domain concepts. The initial documentation was the output of their current clone-and-own reuse practice. That documentation contains variabil-

ity information only in the form of some brief textual notes attached to the relevant use case specifications.

To model the STO requirements according to our product line use case modeling method, PUM, we first examined the initial STO documentation. Since the initial documentation contains almost no structured variability information, we had to work together with IEE engineers to build and iteratively refine our models. When we started to study the STO documentation, the STO project was in its initial phase and there was only one prototype implementation to discuss with some potential customers. One may argue that it is not always easy to identify variations in requirements when a new project starts. However, the IEE analysts stated that, most of the time in their domain of applications, requirements and their variability can be identified with the first customer.

10.1.3 Results

After studying the initial STO documentation and meeting with the IEE analysts, we built the PL use cases and domain model for STO. The diagram in Fig. 6, the use case specifications in Table 2, and the domain model in Fig. 7 are part of the PL models we derived as a result of our modeling effort. Tables 5 and 6 report on the size of the entire PL use cases and domain model for STO.

Table 5 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 Use Cases	11	6	11	57	192	57
Variant Use Cases	13	1	13	131	417	130

Our modeling method, as part of our configuration approach, provided better assistance for capturing and analyzing variability information compared to the current, more informal practice at IEE. With the PL extensions, for example, we could unveil variability information not covered in the initial STO documentation. For instance, the use case diagram extensions helped us identify and model that *Clear Error Status via IEE QC Mode* is mandatory while *Clear Error Status via Diagnostic Mode* is optional (see Fig. 6), which was not previously documented.

When discussions start with a customer regarding a specific product, the IEE analysts need to make decisions on variability aspects documented in PL use case models. At a later stage, when we met again with the IEE analysts for discussing configuration needs, IEE had already developed various STO products for different car manufacturers. By using PUMConf, we, together with the IEE analysts, configured the PS use case and domain models for four products selected among the STO products IEE had already developed. The IEE analysts made the configuration decisions on the PL models using the guidance provided by PUMConf. Table 7 summarizes the results of the configuration for the STO products using our approach.

Table 6 Size of the Domain Model

	Essential Part	Variant Part
# of Entities	42	12
# of Attributes	64	11
# of Associations	28	6
# of Inheritance Relations	22	20

Table 7 Results Summary for the Configuration of STO Use Case and Domain Models for Various Car Manufacturers

Product	# of Selected Variant Use Cases	# of Selected Optional Steps	# of Decided Variant Orders	# of Selected Variant Entities
P1	4	1	0	4
P2	2	1	0	1
P3	6	1	0	5
P4	4	1	0	3

The first column is the number of variant use cases selected by the analysts for each product. In the PL use case diagram, there are six variant use cases with a mandatory variability relation, which are automatically selected. Table 7 does not include the automatically selected variant use cases. The PL specifications have five optional steps in the same variant order group to decide (see the variant use case *Provide System User Data via Standard Mode* in Table 2). Only one optional step is selected for each product (second column in Table 7), thus not requiring any decision on the variant order (third column). The fourth column in Table 7 presents the number of entities selected among twelve variant entities (see Table 6).

All the generated PS use case and domain models were confirmed by the IEE analysts to be correct and complete. The PL models that we derived from the initial STO documentation were sufficient to make all the configuration decisions needed in PUMConf to generate the correct and complete PS models for the STO products.

10.2 Questionnaire Study

We conducted a questionnaire study to evaluate, based on the viewpoints of IEE engineers, how well our configuration approach addresses the challenges that we identified in capturing requirements variability and configuring PS use cases. The study is described and reported according to the template provided by Wohlin et al. [77].

10.2.1 Planning and Design

To evaluate the output of PUMConf in light of the challenges we identified earlier, we had a semi-structured interview with seven participants holding various roles at IEE:

software development manager, software team group leader, software lead engineer, system engineer, and embedded software engineer. All participants had experience with use case-driven development and modeling. The interview was preceded by presentations illustrating the PL extensions of use case and domain models, PUMConf steps, a tool demo, and detailed examples from STO. Interactive sessions included questions posed to the participants about the models. In the sessions, the participants took a more active role and gave us feedback. We also organized three hands-on sessions in which the participants could apply the proposed modeling method and the PUMConf tool. In the first hands-on session, the participants were asked to find inconsistencies in the faulty PL use case diagram and specifications. In the second session, they were using PUMConf to identify and resolve contradicting configuration decisions in the STO PL use case and domain models. In the third session, the participants used PUMConf to configure PS use case and domain models from the STO PL models.

To capture the perception of engineers participating in the interviews, regarding the potential benefits of PUMConf and how it addresses the targeted challenges, we handed out two questionnaires including questions to be answered according to two Likert scales [60] (i.e., agreement and probability). The questionnaires were structured for the participants to assess our modeling method and our configurator, PUMConf, in terms of adoption effort, expressiveness, comparison with current practice, and tool support. The participants were also encouraged to provide open, written comments.

10.2.2 Results and Analysis

We solicited the opinions of the participants using questionnaires (see Fig. 24). The objective of the first questionnaire was to assess our product line use case modeling method. Fig. 24(a) and (b) depict the questions and answers from the participants for the first questionnaire.

All participants agreed that the PL extensions for use case diagrams are simple enough to enable communication between engineers and customers ($QA1$) and thus they would probably use such extensions in their projects ($QA2$). Except for one case, all participants agreed that the extensions provide enough expressiveness to capture variability information in their projects ($QA3$). The participant who disagreed on $QA3$ commented that a few customers do not employ use cases in their development practice and, in these cases, the IEE analysts opt for informal discussions about the product. However, in all cases, use cases are nevertheless employed at IEE as part of their internal practice. All participants stated that the PL specification extensions are simple enough ($QA4$) and variability captured in the diagram is adequately reflected in the specifications ($QA6$). They also all agreed that they would use the specification extensions to capture variability information ($QA5$). There was also a strong consensus among the participants about expressiveness and simplicity of the PL domain model extensions ($QA7$ and $QA8$). The last part of the questionnaire focuses on the overall modeling method in terms of expressiveness, usefulness and adoption effort ($QA9$ - $QA14$). The participants provided a very positive feedback for the method in general but they also stated that (i) additional practice and training was still needed to become familiar with the method and tool support, (ii) customers also needed to be trained regarding the extensions, RUCM and the tool support, and (iii) the software development of a few customers is not use case-driven. These were the reasons stated for the disagreement of two participants on $QA10$ and $QA12$.

The objective of the second questionnaire was to assess our use case-driven configuration approach and its tool support. Fig. 24(c) and (d) depict the corresponding answers. The second questionnaire was structured in four different parts: configuring the PS use case diagram ($QB1$ - $QB3$), configuring the PS use case specifications ($QB4$ and $QB5$), configuring the PS domain model ($QB6$ and $QB7$), and the overall configuration approach and tool support ($QB8$ - $QB13$). All participants agreed that the configurator was adequate to capture configuration decisions for PS use cases and domain models, and they would use the configurator to configure PS models in their projects ($QB1$ - $QB7$). For the overall configuration approach and tool support ($QB8$ - $QB13$), two participants raised issues similar to those of the first questionnaire. Customers also need training to get familiar with the configuration approach and tool support ($QB8$ and $QB9$). Since a few customers do not rely on use case modeling, IEE analysts would in these cases use the configurator only for internal communication and documentation during product development ($QB11$). On the other hand, all participants saw value in adopting the configuration approach ($QB10$), and they agreed that the configurator provides useful assistance for configuring PS use case and domain models, compared to the current practice in their projects ($QB12$).

10.2.3 Threats to Validity

The main threat to validity in our case study concerns the generalizability of the conclusions and lessons learned. To mitigate this threat, we applied PUMConf to an industrial case study that includes nontrivial use cases in an application domain with multiple customers and numerous sources of variability. Though their number is small, we selected the respondents to our questionnaire and interviews to hold various, representative roles and with substantial industry experience. We can also confidently say that the software development practice at IEE is typical of embedded system devel-

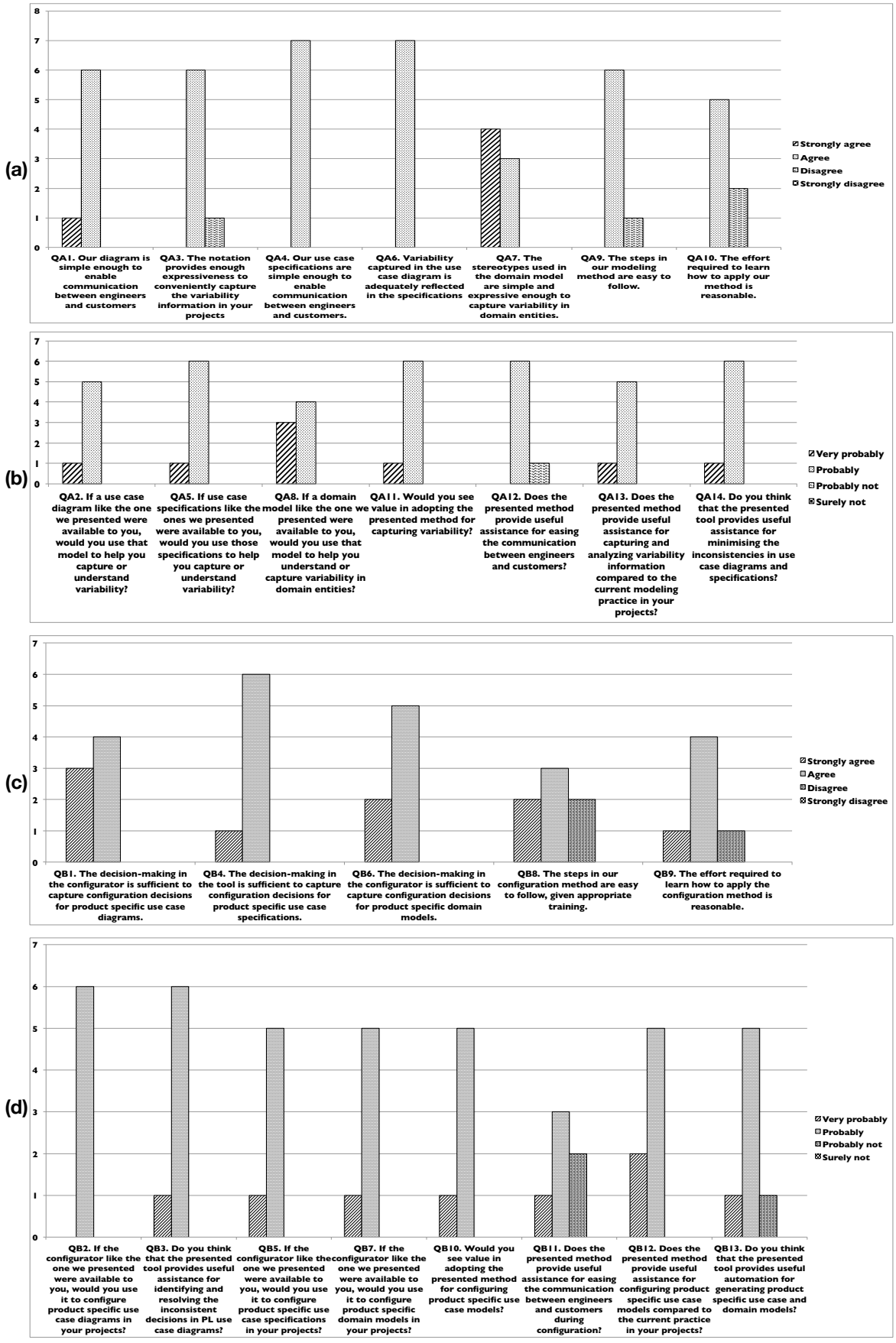


Fig. 24 Responses to the Questions Related to the Product Line Use Case Modeling Method and Configuration Approach

opment in the automotive domain. To limit threats to internal validity, we had many meetings with the IEE analysts in the STO project to verify the correctness and completeness of our models.

10.3 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 PUMConf addresses the challenges reported in Section 2. As reported in Section 10.2, the participants' answers to the questions through Likert scales and their open comments indicate that they see high value in adopting the configuration approach and its tool support in an industrial setting in terms of increasing reusability, minimizing modeling effort, and providing effective automation. In order to elaborate over the open comments in the two questionnaires, we organized further discussions with the participants. Based on the feedback in the comments, we identified three aspects to discuss with the participants: *modeling effort*, *degree of automation*, and *limitations of the configuration approach*.

10.3.1 Modeling Effort

In the current practice at IEE, like in many other environments, there is no systematic way to model variability information in use case and domain models. As mentioned before, the IEE analysts take only brief notes attached to use case specifications to indicate what may vary in the specification. They are reluctant to use feature models traced to use case specifications because of two main issues: (i) having feature models requires considerable additional modeling effort with manual assignment of traces at a very low level of granularity, e.g., sequences of use case steps; and (ii) they find it hard and distracting to switch from feature models to use cases and vice versa during the decision-making process. The PL extensions in Section 5 enable the analysts to model variability information directly in use case and domain models without any feature modeling. The IEE analysts stated that the effort required to apply the extensions for modeling variability information was reasonable. By having variability information in use case and domain models, the analysts could focus on one artifact at a time to make configuration decisions. They considered the extensions to be simple enough to enable communication between analysts and customers, but they also mentioned that training customers may be more of a challenge since the company may need customers' consent to adopt PUMConf. Thus, its costs and benefits should be made clear to customers.

10.3.2 Degree of Automation

In our discussions with the analysts at IEE, we noticed that: (i) the current clone-and-own reuse practice has no systematic way and automated support to decide what to include in PS use case and domain models; (ii) typically, multiple analysts and engineers from both the customer and supplier sides are involved in the decision-making process; (iii) the analysts and engineers have to spend several days to manually review the entire set of requirements cloned from the previous product; and (iv) the intended updates on the cloned use case and domain models are manually carried out by the IEE analysts. On the other hand, PUMConf consists of various automated use case modeling and configuration activities in the context of product lines. The decision making process is automated in the sense that the IEE analysts are guided through the PL artifacts for collecting and verifying configuration decisions, while the generation of PS artifacts does not require any human intervention. Using PUMConf, the IEE analysts only select a set of relevant variant use cases, optional steps, and variant entities for a product. Corresponding PS use case and domain models are obtained from the PL models automatically, which greatly reduces the complexity of the entire configuration process. Though modeling variability in PL models is mostly manual, PUMConf provides automatic consistency checking for these models and feedback to the analyst to help them refine and correct the models. The IEE analysts considered the automated consistency checking of decisions and the generation of PS artifacts to be highly valuable.

10.3.3 Limitations of the Configuration Approach

Our configuration approach and tool support have some limitations at this current stage. First, our modeling method supports only functional requirements. As stated by IEE analysts, there are numerous types of non-functional requirements (e.g., security, timing, and reliability) which may play a key role in variability associated with functional requirements. It is crucial to capture and configure such aspects as well. Second, we do not address and support the evolution of PL use case and domain models. When a new project starts, requirements and their variations might not be fully known. As a result, in early stages, analysts are expected to redefine variation points and variants in requirements specifications through frequent iterations. We were told such changes need to be managed and supported to enable analysts to converge towards consistent and complete requirements and variability information. Third, PUMConf is currently implemented as a plugin in IBM DOORS, in combination with commercial modeling tools used at IEE, i.e., IBM Rhapsody, and Papyrus. PUMConf highly depends on the outputs of these tools. The analysts mentioned that these tools might be re-

placed with other tools or the newer versions of the same tools in the future. Future changes in the tool chain will need to fulfill the following constraints: (i) a new modeling tool for PL diagrams should be extensible in such a way that we can implement the PL diagram extensions in its use case metamodel, (ii) a new requirements management tool should not enforce its own template and restriction rules that conflict with the RUCM and PL specification extensions, and (iii) a new tool for domain modeling should support the profiling mechanism which enables analysts to model the domain with the PL stereotypes.

11 Conclusion

This paper presents a configuration approach that is dedicated to environments relying on use case-driven development. It guides customers in making configuration decisions and automatically generates use case diagrams, use case specifications, and domain models for configured products. Our main motivations are to provide a high degree of automation during configuration and to rely exclusively on variability modeling for commonly used artifacts in use case-driven development, thus avoiding unnecessary modeling overhead and complexity. Our configuration approach builds on our previous work (i.e., Product line Use case Modeling method) and is supported by a tool relying on natural language processing and integrated into IBM DOORS, that aims at (1) checking artifact consistency, (2) identifying partial order of decisions to be made, (3) detecting contradicting decisions, and (4) generating product-specific use case and domain models. The key characteristic of our approach is that variability is directly captured in product line use case and domain models, at a level of granularity enabling both precise communication with various stakeholders, at different levels of details, and automated product configuration. We performed a case study in the context of automotive embedded system development. The results from structured interviews and a questionnaire study with experienced engineers suggest that our approach is practical and beneficial to configure product use case and domain models in industrial settings.

In our current tool support, we assume that product-line use case diagrams are configurable, i.e., there is at least one valid product use case diagram that can be generated from a product-line diagram. We further plan to improve the proposed approach and configurator (PUMConf) to identify non-configurable product-line diagrams before making configuration decisions.

Apart from the product line extensions, which are independent from any application domain, our approach does not make use of any further extensions specific to the embedded, automotive domain. Therefore, PUMConf should be applicable to other domains and should not require any significant adaptation as long as software development is use case-driven.

PUMConf does not currently support the detection of decisions violating constraints imposed by the variant dependencies in PL domain models. We plan to improve the tool for detecting such decisions, which is very similar to the detection of contradicting decisions in PL use case diagrams.

For resolving contradicting decisions in PL use case diagrams, our approach follows the strategy in which the detected contradiction is fixed right away and the configuration is returned to a consistent state. Our motivation stemmed from the observation that, in the considered business context, tolerating contradictions in decision-making significantly increases the complexity of communication with customers. On the other hand, from a general standpoint, fixing a contradiction immediately may not always be the optimal solution. Therefore, we plan to extend our approach to support multiple contradiction resolution strategies.

We assume that complete product specific models are generated from product line models in a single (albeit complex and iterative) stage, which is performed jointly by a team that combines the activities of all the different parties involved in the configuration process. This is a valid assumption in the observed business context, as well as in many other similar contexts. Given this assumption, our tool provides the possibility of backtracking to change configuration decisions made in preceding steps of the same stage. However, other organizations may require a multi-stage configuration process, involving multiple physically dispersed configuration teams, which have to perform their configuration steps in a pre-defined order. Each step may need to be performed by different experts at different times in physically different locations, possibly using different configuration tools. In such multi-stage configuration, supporting backtracking to earlier stages for resolving contradictions may not be an option.

Our evaluation does not address the usability of PUMConf, especially in terms of resolving contradicting decisions. As future work, we plan to conduct an extensive user study with engineers to evaluate the effort that needs to be made in resolving contradicting decisions in PUMConf.

PUMConf is only a first step to achieve our long term objective, i.e., change impact analysis and regression test selection in the context of use case-driven development and testing. Change can occur both in configuration decisions and variability aspects of product-line models. For decision changes in a product, the impact on other decisions needs to be assessed and re-configuration should be considered in the product-specific model. Further, the impact on the execution of test cases should be assessed. In contrast, changes on product-line use case models require impact assessment on decisions for each individual product and may entail re-

configuration and regression test selection in several products. Our plan for the next stages is to support change impact analysis to help analysts properly manage change in contexts where product-line and products are constantly and concurrently evolving.

Acknowledgments

Financial support was provided by IEE and FNR under grants FNR/P10/03 and FNR10045046.

References

1. DOPLER (Decision Oriented Product Line Engineering for effective Reuse), http://www.ase.jku.at/dopler/
2. Eclipse EMF, https://eclipse.org/modeling/emf/
3. IEE (International Electronics & Engineering) S.A., http://www.iee.lu/
4. pure::variants for IBM Rational DOORS, http://www.pure-systems.com/DOORS.174.0.html
5. pure::variants, http://www.pure-systems.com/pure_variants.49.0.html
6. Alférez, M., Bonifácio, R., Teixeira, L., Accioly, P., Kulesza, U., Moreira, A., Araújo, J., Borba, P.: Evaluating scenario-based spl requirements approaches: the case for modularity, stability and expressiveness. Requirements Engineering Journal **19**, 355–376 (2014)
7. Alferez, M., Kulesza, U., Moreira, A., Araujo, J., Amaral, V.: Tracing between features and use cases: A model-driven approach. In: VAMOS'08, pp. 81–88 (2008)
8. Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., Amaral, V.: Multi-view composition language for software product line requirements. In: SLE'09, pp. 103–122 (2009)
9. Alves, V., Niu, N., Alves, C., Valença, G.: Requirements engineering for software product lines: A systematic review. Information and Software Technology **52**, 806–820 (2010)
10. Armour, F., Miller, G.: Advanced Use Case Modeling: Software Systems. Addison-Wesley (2001)
11. Arora, C., Sabetzadeh, M., Goknil, A., Briand, L.C., Zimmer, F.: Change impact analysis for natural language requirements: An nlp approach. In: RE'15, pp. 6–15 (2015)
12. Arora, C., Sabetzadeh, M., Goknil, A., Briand, L.C., Zimmer, F.: NARCIA: an automated tool for change impact analysis in natural language requirements. In: ESEC/SIGSOFT FSE 2015, pp. 962–965 (2015)
13. Batory, D.: Feature models, grammars, and propositional formulas. In: SPLC'05, pp. 7–20 (2005)
14. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems **35**(6), 615–636 (2010)
15. Benavides, D., Trinidad, P., Ruiz-Cortes, A.: Automated reasoning on feature models. In: CAiSE'05, pp. 491–503 (2005)
16. Biddle, R., Noble, J., Tempero, E.: Supporting reusable use cases. In: ICSR'02, pp. 210–226 (2002)
17. Blanes, D., Gonzalez-Huerta, J., Insfran, E.: A multimodel approach for specifying the requirements variability on software product lines. In: ISD'14, pp. 329–336 (2014)
18. Bonifácio, R., Borba, P.: Modeling scenario variability as croscutting mechanisms. In: AOSD'09, pp. 125–136 (2009)
19. Bonifácio, R., Borba, P., Ferraz, C., Accioly, P.: Empirical assessment of two approaches for specifying software product line use case scenarios. Software and Systems Modeling (2015)
20. Bonifácio, R., Borba, P., Soares, S.: On the benefits of scenario variability as croscutting. In: EA-AOSD'08, pp. 1–6 (2008)
21. Braganca, A., Machado, R.J.: Automating mappings between use case diagrams and feature models for software product lines. In: SPLC'07, pp. 3–12 (2007)
22. Buhne, S., Halmans, G., Lauenroth, K., Pohl, K.: Scenario-based application requirements engineering. In: Software Product Lines. Springer (2006)
23. Buhne, S., Halmans, G., Pohl, K.: Modeling dependencies between variation points in use case diagrams. In: REFSQ'03, pp. 59–69 (2003)
24. seok Choi, W., Kang, S., Choi, H., Baik, J.: Automated generation of product use case scenarios in product line development. In: CIT'08, pp. 760–765 (2008)
25. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
26. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2001)
27. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, pp. 422–437 (2005)
28. Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau, S., Pietroszek, K.: fmp and fmp2rsm: Eclipse plug-ins for modeling features using model templates. In: OOPSLA'05, pp. 200–201 (2005)
29. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3), 621–645 (2006)
30. Dhungana, D., Grünbacher, P., Rabiser, R.: The DOPLER metatool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering **18**, 77–114 (2011)
31. Duran, A., Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortes, A.: FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. Software and Systems Modeling (2016)
32. Eriksson, M., Borstler, J., Asa, A.: Marrying features and use cases for product line requirements modeling of embedded systems. In: SERPS'04, pp. 73–82 (2004)
33. Eriksson, M., Borstler, J., Borg, K.: The pluss approach - domain modeling with features, use cases and use case realizations. In: SPLC'05, pp. 33–44 (2005)
34. Eriksson, M., Borstler, J., Borg, K.: Managing requirements specifications for product lines - an approach and industry case study. Journal of Systems and Software **82**, 435–447 (2009)
35. Fantechi, A., Gnesi, S., John, I., Lami, G., Dorr, J.: Elicitation of use cases for product lines. In: PFE'03, pp. 152–167 (2004)
36. Fantechi, A., Gnesi, S., Lami, G., Nesti, E.: A methodology for the derivation and verification of use cases for product lines. In: SPLC'04, pp. 255–265 (2004)
37. Faulk, S.R.: Product-line requirements specification (PRS): an approach and case study. In: RE'01, pp. 48–55 (2001)
38. Forbus, K.D., Kleer, J.D.: Building Problem Solvers. MIT Press (1993)
39. Goknil, A., Kurtev, I., van den Berg, K.: A metamodeling approach for reasoning about requirements. In: ECMDA-FA'08, pp. 310–325 (2008)
40. Goknil, A., Kurtev, I., van den Berg, K., Veldhuis, J.W.: Semantics of trace relations in requirements models for consistency checking and inferencing. Software and Systems Modeling **10**, 31–54 (2011)
41. Goknil, A., Kurtev, I., Millo, J.V.: A metamodeling approach for reasoning on multiple requirements models. In: EDOC'13, pp. 159–166 (2013)
42. Gomaa, H.: Object oriented analysis and modeling families of systems with uml. In: ICSR-6, pp. 89–99 (2000)
43. H. Cunningham et al.: Developing language processing components with gate version 8 (a user guide), http://gate.ac.uk/sale/tao/tao.pdf

44. Hajri, I., Goknil, A., Briand, L.C., Stephany, T.: Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach. In: MODELS'15, pp. 338–347 (2015)
45. Halmans, G., Pohl, K.: Communicating the variability of a software-product family to customers. Software and Systems Modeling **2**, 15–36 (2003)
46. John, I., Muthig, D.: Product line modeling with generic use cases. In: EMPRESS'04 (2004)
47. Kulak, D., Guiney, E.: Use Cases: Requirements in Context. Addison-Wesley (2003)
48. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall (2002)
49. Lauenroth, K., Pohl, K.: Towards automated consistency checks of product line requirements specifications. In: ASE'07, pp. 373–376 (2007)
50. Lauenroth, K., Pohl, K.: Dynamic consistency checking of domain requirements in product line engineering. In: RE'08, pp. 193–202 (2008)
51. Mendonca, M., Branco, M., Cowan, D.: S.P.L.O.T. - software product lines online tools. In: 761-762 (ed.) OOPSLA'09 (2009)
52. Moon, M., Yeom, K.: An approach to develop requirement as a core asset in product line. In: ICSR'04, pp. 23–34 (2004)
53. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. IEEE Transactions on Software Engineering **31**(7), 551–569 (2005)
54. Mussbacher, G., Araújo, J., Moreira, A., Amyot, D.: AoURN-based modeling and analysis of software product lines. Software Quality Journal **20**, 645–687 (2012)
55. Myllärniemi, V., Asikainen, T., Männistö, T., Soininen, T.: Kumbang configurator - a configuration tool for software product families. In: IJCAI-05, pp. 51–57 (2005)
56. Nie, K., Yue, T., Ali, S., Zhang, L., Fan, Z.: Constraints: The core of supporting automated product configuration of cyber-physical systems. In: MODELS'13, pp. 370–387 (2013)
57. Nöhrer, A., Biere, A., Egyed, A.: Managing SAT inconsistencies with HUMUS. In: VaMoS'12, pp. 83–91 (2012)
58. Nöhrer, A., Egyed, A.: Conflict resolution strategies during product configuration. In: VaMoS'10, pp. 107–114 (2010)
59. Nöhrer, A., Egyed, A.: C2O configurator: a tool for guided decision-making. Automated Software Engineering **20**, 265–296 (2013)
60. Oppenheim, A.N.: Questionnaire Design, Interviewing and Attitude Measurement. Continuum (2005)
61. Rabiser, R., Grünbacher, P., Dhungana, D.: Requirements for product derivation support: Results from a systematic literature review. Information and Software Technology **52**, 324–346 (2010)
62. Rosa, M.L., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Questionnaire-based variability modeling for system configuration. Software and Systems Modeling **8**, 251–274 (2009)
63. Sinnema, M., Deelstra, S.: Industrial validation of COVAMOF. Journal of Systems and Software **81**, 584–600 (2008)
64. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: COVAMOF: A framework for modeling variability in software product families. In: SPLC'04, pp. 197–213 (2004)
65. Stoiber, R.: A new approach to product line engineering in model-based requirements engineering. Ph.D. thesis, University of Zurich (2012)
66. Stoiber, R., Fricker, S., Jehle, M., Glinz, M.: Feature unweaving: Refactoring software requirements specifications into software product lines. In: RE'10, pp. 403–404 (2010)
67. Stoiber, R., Glinz, M.: Supporting stepwise, incremental product derivation in product line requirements engineering. In: VaMoS'10, pp. 77–84 (2010)
68. Sun, J., Zhang, H., Li, Y.F., Wang, H.: Formal semantics and verification for feature modeling. In: ICECCS'05, pp. 303–312 (2005)
69. Trinidad, P., Benavides, D., Duran, A., Ruiz-Cortes, A., Toro, M.: Automated error analysis for the agilization of feature modeling. Journal of Systems and Software **81**, 883–896 (2008)
70. Trinidad, P., Ruiz-Cortes, A.: Abductive reasoning and automated analysis of feature models: How are they connected? In: VaMoS'09, pp. 145–153 (2009)
71. Varela, P., Araújo, J., Brito, I., Moreira, A.: Aspect-oriented analysis for software product lines requirements engineering. In: SAC'11, pp. 667–674 (2011)
72. Wang, B., Zhang, W., Zhao, H., Jin, Z., Mei, H.: A use case based approach to feature models' construction. In: RE'09, pp. 121–130 (2009)
73. Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, M.Z.Z.: Automatic generation of system test cases from use case specifications. In: ISSTA'15, pp. 385–396 (2015)
74. Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, M.Z.Z.: UMTG: a toolset to automatically generate system test cases from use case specifications. In: ESEC/SIGSOFT FSE 2015, pp. 942–945 (2015)
75. Wang, H., Li, Y.F., Sun, J., Zhang, H., Pan, J.: A semantic web approach to feature modeling and verification. In: SWESE'05 (2005)
76. Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: SPLC'09, pp. 211–220 (2009)
77. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering. Springer (2012)
78. Yue, T., Ali, S., Briand, L.C.: Automated transition from use cases to uml state machines to support state-based testing. In: ECMFA'11, pp. 115–131 (2011)
79. Yue, T., Briand, L.C., Labiche, Y.: Facilitating the transition from use case models to analysis models: Approach and experiments. ACM Transactions on Software Engineering and Methodology **22**(1) (2013)
80. Yue, T., Briand, L.C., Labiche, Y.: aToucan: An automated framework to derive uml analysis models from use case models. ACM Transactions on Software Engineering and Methodology **24**(3) (2015)
81. Zhang, G., Yue, T., Wu, J., Ali, S.: Zen-RUCM: A tool for supporting a comprehensive and extensible use case modeling framework. In: Demos@MoDELS 2013, pp. 41–45 (2013)
82. Ziadi, T., Jezequel, J.M.: Product line engineering with the uml: Deriving products. In: Software Product Lines. Springer (2006)
83. Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: Vml* – a family of languages for variability management in software product lines. In: SLE'09, pp. 82–102 (2009)