Applying Product Line Use Case Modeling in an Industrial Automotive Embedded System: Lessons Learned and a Refined Approach

Ines Hajri, Arda Goknil, Lionel C. Briand
SnT Centre for Security, Reliability and Trust
University of Luxembourg, Luxembourg
{ines.hajri, arda.goknil, lionel.briand}@uni.lu

Thierry Stephany
IEE
Contern, Luxembourg
{thierry.stephany}@iee.lu

***Abstract*—In this paper, we propose, apply, and assess Product line Use case modeling Method (PUM), an approach that supports modeling variability at different levels of granularity in use cases and domain models. Our motivation is that, in many software development environments, use case modeling drives interactions among stakeholders and, therefore, use cases and domain models are common practice for requirements elicitation and analysis. In PUM, we integrate and adapt existing product line extensions for use cases and introduce some template extensions for use case specifications. Variability is captured in use case diagrams while it is reflected at a greater level of detail in use case specifications. Variability in domain concepts is captured in domain models. PUM is supported by a tool relying on Natural Language Processing (NLP). We applied PUM to an industrial automotive embedded system and report lessons learned and results from structured interviews with experienced engineers.**

I. Introduction

The complexity of automotive embedded systems has been escalating in recent years. An automotive system is comprised of various interacting subsystems such as airbag controller, trunk controller, adaptive cruise controller, and sensing systems. In most situations, development is distributed over multiple suppliers. For example, while one supplier specializes in a sensing solution for the seat occupant classification, another supplier may develop an airbag controller using the sensing solution output to disable the airbag for children and unoccupied seats. These suppliers work with multiple customers (manufacturers) requiring different versions of the same product. Therefore, Product Line Engineering (PLE) is an inherent component of their development process, from requirements analysis to implementation.

IEE S.A. [1], is a representative supplier in the automotive domain, producing sensing systems (e.g., $BodySense^{TM}$, *Smart Trunk Opener*, and *Driver Presence Detection*) for multiple automotive manufacturers. Such systems sense the environment through physical components (e.g., electrical field sensors, pressure sensitive sensors, and force sensing resistors), derive a conclusion from sensor measurements (e.g., seat occupant classification, gesture recognition, and driver presence detection), and inform other external systems (e.g., airbag control unit, trunk controller, and engine controller). In IEE's business context, like in many others, use cases are central development artifacts and beneficial 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 the variability to customers, documenting it for software engineers, and supporting decision making during the elicitation of customer specific requirements [2]. Therefore, the need for integrating PLE concepts in use case driven development, starting with use case specifications, led us to assess the relevant state of the art and develop a product line use case modeling method supporting embedded system development, though it should be easily adaptable to other contexts. Our motivation is to rely, to the largest extent possible, on common practices, including the ones at IEE, to achieve widespread applicability.

Considerable research has been devoted to documenting variability in use cases. Many approaches [3] [4] [5] require that feature models be connected to use case specifications and diagrams. In such cases, feature modeling needs to be introduced into practice, including establishing and maintaining traces between feature models and use case specifications and diagrams, as well as other artifacts. In many development environments, such additional modeling and traceability is often perceived as an unnecessary overhead. Some other works [6] [7] [8] propose use case template extensions for textual representation of variability. As Halmans and Pohl [2] stated, textual representation has shortcomings: variation points are not explicit, variability constraints (e.g., number of variants that can be selected for a variation point) cannot be easily represented, and variants are hard to identify. There are several approaches [2] [9] [10] proposing extensions to use case diagrams to overcome these shortcomings. Nevertheless, these extensions are usually not sufficient to express all variabilities at the required level of granularity. In addition, variability in use case diagrams needs to be reflected

in use case specifications to provide consistency of diagram and specifications.

In this paper, we propose, apply, and assess a *Product line Use case modeling Method (PUM)*, which aims at enabling the analysts to document variability at different levels of granularity, both in use case diagrams and specifications. In PUM, we adopt the product line extensions of use case diagrams proposed by Halmans and Pohl [2] to overcome the shortcomings associated with textual representation of variability. Further, we augment a more structured and analysable form of use case specifications, i.e., Restricted Use Case Modeling (RUCM) [11]. RUCM is based on a template and restriction rules, reducing the imprecision and incompleteness in use cases. We chose RUCM in PUM because it reduces ambiguity and facilitates automated analysis of use cases. RUCM was previously evaluated through controlled experiments and has shown to be usable and beneficial with respect to making use cases less ambigious and more amenable to precise analysis and design. Since RUCM was not originally designed for modeling variability in embedded systems, we introduce some extensions to represent the types of variability that cannot be captured in a use case diagram.

In addition to use case modeling, common practice in many environments also includes domain modeling, which aims at capturing domain concepts shared among stakeholders, with associated variability where needed. Such domain modeling is also part of PUM as it enables the use of consistent terminology and concepts, as well as precise definitions of conditions in use case specifications and, therefore, of related variation points. PUM expects these conditions, referring to the domain model, to be defined with the Object Constraint Language (OCL) [12] since OCL is the natural choice when defining high-level constraints on class diagrams. To summarize, the contributions of this paper are:

- PUM, a use case modeling method, which integrates and builds on existing work and captures variability in product lines at a level of granularity enabling both precise communication and guided product configuration;
- a practical, industry-strength tool, relying on Natural Language Processing (NLP) to report inconsistencies between use case diagrams and use case specifications complying with the RUCM template;
- an industrial case study demonstrating the applicability of PUM, including structured interviews with experienced engineers, from which we also draw lessons learned and guidelines. This is the first time such industrial case study, on capturing variability in use case models, is reported.

This paper is structured as follows. Section II introduces the industrial context of our case study to provide the motivations behind PUM. Section III discusses the related work. In Section IV, we provide an overview of PUM. Section V focuses on use case diagrams and RUCM with their extensions while Section VI presents the tool support for PUM. In Section VII, we present our case study, involving an embedded system called Smart Trunk Opener (STO), along with interview results and lessons learned. We conclude the paper in Section VIII.

II. Motivation and Context

The context for which we developed PUM is that of automotive embedded systems, interacting with multiple external systems, and developed by a supplier for multiple automotive manufacturers. These systems are representative examples in which variability in requirements needs to be communicated among stakeholders, including customers. For instance, IEE negotiates, with each customer, how to resolve variation points, that is, the configuration of the product line.

In this paper, we use the embedded system *Smart Trunk Opener (STO)* as a case study, to motivate, illustrate, and assess our modeling method. 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 the user has the remote. This allows the trunk controller to open the trunk automatically.

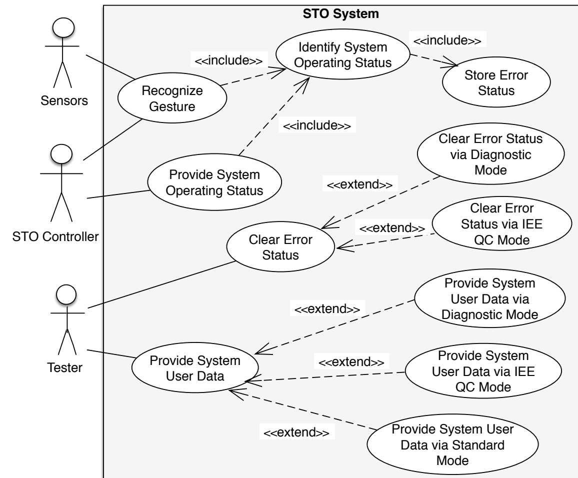

Fig. 1. Part of the UML Use Case Diagram for STO

The current use case driven development practice in IEE involves UML use case diagrams and use case specifications. Fig. 1 depicts part of the UML use case diagram for STO. *Sensors*, *STO Controller* and *Tester* are the actors of the system. The use cases describe four main functions: *recognize gesture*, *provide system operating status*, *provide system user data*, and *clear errors stored in the system*.

UML provides the *extend* and *include* relations in use case diagrams to extend the behaviour of use cases and to factor out common parts of the behaviors of two or more use cases, respectively. However, it is not possible to explicitly document variability, e.g., which use cases are mandatory and which use cases are variant. There is no way to represent variation points, i.e., location at which a variation occurs. We know from our discussions with domain experts in IEE that, in Fig. 1, there are three variation points: *Store Error Status*, *Clear Error Status* and *Provide System User Data*. *Store Error Status* and *Clear*

Error Status are optional while *Provide System User Data* is mandatory. Cardinality in a variation point, i.e., number of variants to be chosen, cannot be represented in a UML use case diagram [2]. For instance, there must be at least two ways to realize *Provide System User Data* (i.e., at least two use cases that extend the 'Provide System User Data' use case) in any STO product. Variant dependencies play a key role in the selection of variants. For example, if an STO product provides *Store Error Status*, then it must also provide *Clear Error Status*.

A use case specification contains detailed description of a use case given in a use case diagram and usually conforms to a use case template [13]. A use case template is a structure guiding the textual documentation of use cases [14]. IEE has so far been following the Cockburn's template [13] (see Table I for some simplified STO use cases).

TABLE I
SOME USE CASES FOR STO

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 user data via diagnostic mode.
26	1a1. The system *'provides system user data via Diagnostic Mode'*.
27	1b. The tester requests receiving system user data via IEE QC mode.
28	1b1. The system *'provides system user data via IEE QC Mode'*.
29	
30	**USE CASE** Provide System User Data via Standard Mode
31	**Main Success Scenario**
32	1. The system sends the calibration data to the tester.
33	2. The system sends the trace data to the tester.
34	3. The system sends the error data to the tester.
35	4. The system sends the sensor data to the tester.

Standard use case templates, such as Cockburn's, are insufficient to document variability in use case specifications. For example, variation points and variants are not visible in the STO use case specifications. Variant use cases, e.g., *Provide System User Data via Standard Mode* in lines 21-28, are not distinguishable from any other use case. In lines 18-19 and 25-28 there are two variation points modelled as extensions of the basic flow, i.e., executing the variant use cases *Store Error Status*, *Provide System User Data via Standard Mode, IEE QC Mode, and Diagnostic Mode*. However, these steps are no different from any other step describing the execution of a mandatory use case, e.g., the execution of the *Identify System Operating Status* use case in line 2. Some use case steps might be optional or their order may vary in the product line. For instance, all steps given in lines 32-35 are optional with a variant order. The analyst first needs to properly document the optional steps. Then, she can negotiate with the customer to decide which steps to select, according to which order, for the product.

Within the context of developing industrial automotive embedded systems for multiple manufacturers, we identify three challenges that need to be considered in capturing requirements' variability in use cases:

Challenge 1: **Modeling Variability with Constraints and Dependencies.** It is crucial to have variability information explicitly documented (i.e., variants, variation points, their constraints and dependencies) in order to decide with the customers which variants to include for the product and guide product configuration. Textual representation of use cases has shortcomings in explicitly representing variability information. Furthermore, it is not easy for analysts and customers to comprehend and visualize all variability information encoded in a textual representation. For instance, 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). The variability information scattered across all these use case flows in the textual description needs to be communicated with customers and used to configure a product.

Challenge 2: **Reflecting Variability in Use Case Specifications.** There are approaches that extend only use case diagrams with the notion of variation point and variant to express variability and associated constraints. IEE, as well as many similar companies, rely on detailed use case specifications to communicate with their customers. Variability should therefore be reflected in use case specifications to provide diagram-specification consistency. In addition, there are types of variability (e.g., optional steps) which cannot be expressed in use case diagrams, at the required level of granularity, to precisely guide product configuration. However, it is nevertheless crucial to also model some variability information in use case diagrams, for example to improve visualization of key variability information and provide a roadmap to look up the details in use case specifications.

Challenge 3: **Capturing Variability with Precise Conditions.** Use cases are not meant to clarify terminology and domain concepts shared among all stakeholders. In order to document domain concepts and their variability, any product line requirements modeling method would need models where the analyst can specify mandatory and optional domain entities. In addition, 'flows of events' in use cases feature associated conditions determining their occurrence, which need to be precisely specified to help communication among stakeholders. For instance, the precise definition of a valid kick (see lines 4 and 7 in Table I) is crucial for the IEE engineers

to identify the correct execution of the product in terms of 'flows of events' in the STO use cases.

In the remainder of this paper, we focus on how to best address these three challenges in a practical manner, in the context of use case driven development, while minimising the modeling overhead. Automated configuration and change impact analysis are two potential applications of product line use case modeling, which we leave out for future work. In collaboration with customers, analysts need to discuss variability documented in use cases to configure the required parts of the system design, implementation, and test cases. Customers may change their decisions during or after configuration as the product evolves. Therefore, in addition to configuration, analysts need to perform change impact analysis to identify what other decisions may be impacted and thus what artifacts must be updated.

III. Related Work

In this section, we cover related work across four categories.

Relating Feature Models and Use Cases. Some approaches propose using feature models for modeling variability information within the context of use case driven development (*Challenges 1 and 2*). Griss et al. [15] describe a manual process for constructing a feature model from a use case diagram. The main idea is to extract the structure of feature models from use case dependencies (i.e., *include* and *extend*) in use case diagrams. Braganca et al. [16] investigate the use of model transformation to automate the same idea but their approach requires that each feature be mapped to only one use case. Eriksson et al. [3] [17] [18] propose another approach relating use cases and features at a lower level of granularity, i.e., sequences of use case steps. Buhne et al. [19] use Orthogonal Variability Models (OVMs) traced to use case diagrams and specifications. An OVM documents the variable aspects of a product line by using variation points, variants and their dependencies. Following traces, analysts can identify how a given variant in the OVM is implemented in use case diagrams and specifications. Alferez et al. [5] provide a trace metamodel to capture traces between feature models and use cases. XTraQue [20] is a tool which supports semi-automatic trace generation for use cases and feature models. Despite advances in traceability research, all approaches given above bring additional modeling and traceability effort into practice. Furthermore, their correctness highly depends on the correctness and precision of traces. To use these approaches, in most cases, traces between variability models and other modeling artifacts, e.g., use cases, need to be manually established at a very low level of granularity, e.g., conditions in use case steps. Traces should be maintained for every single change in any traced artifact. Our objective is to achieve the same result by solely relying on use case and domain modeling, which are common practice.

Extending Use Case Templates. Some works propose use case templates with product line extensions to model variability in use case specifications (*Challenges 1 and 2*). Gallina and Guelfi [6] provide a product line use case template in which variants and variation points can be expressed. The template requires that variability information be encoded in use case specifications containing the fields 'selection category' and 'variation point'. These two fields do not follow any structured format to precisely define variability, in order, for example, to support product configuration. The representation of variation point cardinalities is not addressed in the template. Biddle et al. [21] provide support for customizing use cases through parametrization. Nebut et al. [7] enhance a use case template with parameters and contracts for product line system testing. These two approaches using parameters do not allow analysts to explicitly document variants and variation points. Fantechi et al. [8] [22] propose Product Line Use Cases (PLUCs), an extension of the Cockburn's 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. In most of the approaches given above, either variants and variation points cannot be documented or it is not possible to express all required types of variability constraints. It is crucial to explicitly document variability information containing variants and variation points with all their constraints and dependencies (*Challenge 1*) since analysts and customers need them to make decisions during configuration.

Extending Use Case Diagrams. Variability modeling in use cases are also addressed by approaches extending use case diagrams with new relations and stereotypes (*Challenge 1*). Maßen and Lichter [23] propose two new relations to represent alternative and optional use cases in UML use case diagrams, without any support for expressing variation points. Azevedo et al. [24] [10] explore the use of the UML 'extend' relation with the new stereotypes 'alternative', 'specialization' and 'option' to distinguish variability types. The 'alternative' and 'specialization' are applied to the 'extend' relation while the 'option' is applied to use cases that represent options. The use of the 'extend' with the stereotypes does not address variation points and their cardinalities. John and Muthig [25] introduce the stereotype 'variant' to use case diagrams. They also use the tag 'variant' for variant text fragments in use case specifications. They propose a new artifact, called decision model, to represent variation points textually but such a decision model can quickly become too complex for the analyst to comprehend. Halmans and Pohl [2] propose extensions to use case diagrams to explicitly represent variants, variation points, and associated constraints. Buhne et al. [9] enhance the extensions with some common dependency types from feature modeling. Based on our observations in practice, Halmans et al.'s extensions support a subset of our needs (*Challenge 1*) and we therefore include them in our methodology. The approaches above do not reflect variability in use case specifications (*Challenge 2*) since they do not use any template extensions. Therefore, in PUM, we introduce extensions to RUCM to reflect variability in use case specifications and also represent the types of variability, e.g., optional use case steps, that cannot be captured in use case diagrams (*Challenge 2*).

Capturing Variability in Domain Models. Variability in domain models is mostly addressed by introducing new stereotypes into UML class diagrams. Ziadi and Jezequel [26] suggest three stereotypes (i.e., *variant*, *variation*, and *optional*) to specify variability in domain models. These stereotypes are very similar to the 'kernel' and 'optional' stereotypes proposed by Gomaa [27]. In addition, Ziadi and Jezequel suggest using OCL to specify dependencies between variants in domain models, e.g., the presence of a variant requires the presence of another variant. In contrast, in PUM, we specify these dependencies in use case diagrams. We employ OCL and domain models to precisely specify conditions associated with flows of events in use case specifications (*Challenge 3*).

IV. Overview of Our Modeling Method

As depicted in Fig. 2, PUM is designed to address the challenges stated above in the use case driven development context we described, and builds upon and integrates existing work. The PUM output is *product line use case diagram*, *product line use case specifications*, *domain model*, and *OCL constraints*. Variability, and its constraints and dependencies, are captured in the use case diagram (*Challenge 1*) while it is further detailed in the use case specifications (*Challenge 2*). Variability in domain concepts is captured in the domain model (*Challenge 3*). Use case conditions are reformulated as OCL constraints referring to the domain model (*Challenge 3*).

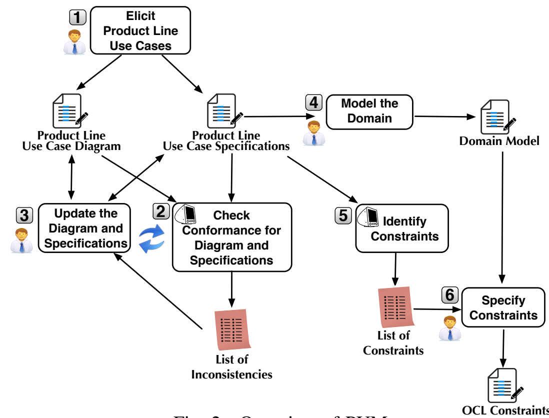

Fig. 2. Overview of PUM

The analyst elicits product line use cases with the use case diagram, the RUCM template, and their product line extensions (Step 1). PUM-C (Product line Use case Modeling - Checker), the tool we developed for PUM, automatically checks for use case diagram and specification consistency (RUCM template) and reports inconsistencies (Step 2). The tool relies on Natural Language Processing (NLP). If there is any inconsistency, the analyst updates the diagram and/or specifications (Step 3). Steps 2 and 3 are iterative: the specifications and diagram are updated until the specifications conform to the RUCM template and they are consistent with the diagram.

The domain model is manually created as a UML class diagram by the analyst (Step 4). It is important for the analyst to clarify domain concepts shared among all stakeholders. Variability in these concepts is expressed in the domain model by tagging domain entities as *variation*, *variant*, and *optional*. After the model is completed, textual descriptions of conditions in the use case specifications are automatically extracted (Step 5) to be reformulated from English to OCL by the analyst (Step 6).

The rest of the paper provides a detailed description of each step in PUM, along with detailed illustrations from STO.

V. Capturing Variability in Requirements

In this section, we provide a detailed description of the artifacts produced by PUM. We also highlight how they were extended, compared to what was proposed in existing work, to address our needs.

A. Use Case Diagram with Product Line Extensions

PUM uses the product line extensions of use case diagrams proposed by Halmans and Pohl [2]. We do not introduce any further additions into the extensions. We chose these extensions for PUM because they support explicit representation of variants, variation points, and their depedencies (*Challenge 1*). In this section, we briefly define the extensions and the reader is referred to [2] [9] for further details. Fig. 3 depicts the graphical notation of the extensions.

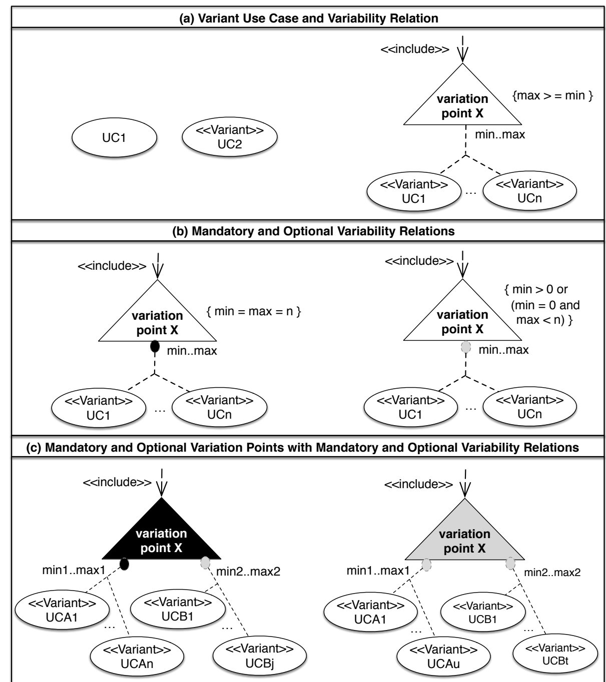

Fig. 3. Graphical Notation of Product Line Extensions for Use Case Diagrams

Variant use cases are distinguished from essential use cases, i.e., mandatory for all products in a product family, by using the 'Variant' stereotype (Fig. 3(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 [2]. Optional and mandatory relations are depicted with light-grey and black filled circles, respectively (Fig. 3(b)).

Multiple variability relations can be combined to specify the desired cardinality in a variation point [2]. A variation point is optional or mandatory based on its variability relations. A variation point is *mandatory* if ($min > 0$) in at least one variability relation for that variation point. It is *optional* if ($min = 0$) in all its variability relations. Optional and mandatory variation points are rendered as grey and black-filled triangles, respectively (Fig. 3(c)). Besides the 'include' relation, two more variant relations for variants and variation points (i.e., 'require' and 'exclusive' [9]) are used in PUM since these two relations model the consequences of decisions in product configuration. For instance, a variant might require or exclude the choice of another variant. Halmans and Pohl do not provide any metamodel or UML profile for the extensions in their paper [2]. In Fig. 4, to facilitate the tailoring of use case modeling tools, such a metamodel is depicted.

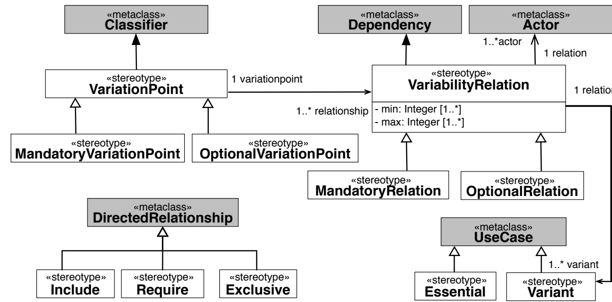

Fig. 4. Extended UML Use Case Metamodel

A use case is extended as *Essential* and *Variant*. The notation for variation points, in Fig. 3, can also be applied to specify variation points for actors [2]. Therefore, a variability relation relates a variation point either to a variant use case or to an actor (see *VariabilityRelation* in Fig. 4).

Fig. 5 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' relation relates the two optional variation points such that if the customer selects the variant use case in the 'Storing Error Status' variation point, he has to select the variant use case in the 'Clearing Error Status' variation point.

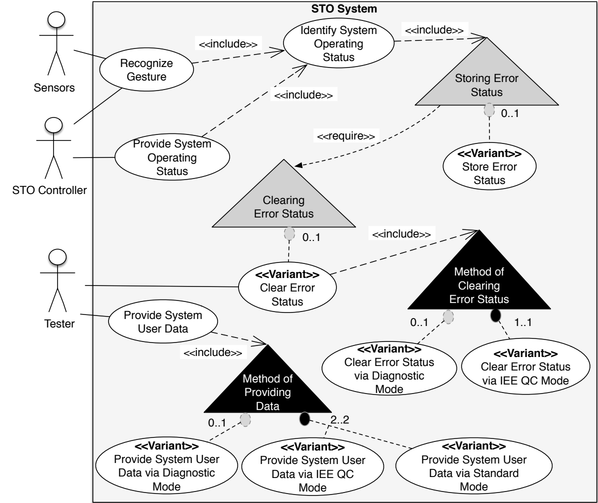

Fig. 5. Part of the Product Line Use Case Diagram for STO

In use case diagrams, we capture variants, variation points, their cardinalities and dependencies. However, some detailed information of variability cannot be captured in these diagrams. For instance, the diagram in Fig. 5 indicates that the 'Identify System Operating Status' use case includes the 'Storing Error Status' optional variation point. To find out in which flows of events the variation point is included, the analyst has to check the corresponding use case specification.

B. Restricted Use Case Modeling (RUCM) and its Extensions

This section briefly introduces the RUCM template and our extensions for variability modeling in embedded systems. RUCM provides restriction rules and specific keywords constraining the use of natural language in use case specifications [11]. We chose RUCM for PUM 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 product line modeling of embedded systems, we had to introduce extensions (*Challenge 2*).

Table II provides some 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 II, 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 39-42). 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.

TABLE II
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 Std. Mode
38	**1.1 Basic Flow**
39	V1. <OPTIONAL>The system SENDS calibration TO the tester.
40	V2. <OPTIONAL>The system SENDS trace data TO the tester.
41	V3. <OPTIONAL>The system SENDS error data TO the tester.
42	V4. <OPTIONAL>The system SENDS sensor data TO the tester.

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 II, 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 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 the keyword '*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.

PUM introduces 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 [28], 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 II (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 the notion of variation point and variant, complementary to the diagram extensions in Section V-A, into the RUCM template. 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 variant order of steps. Optional steps and optional alternative flows begin with the '*<OPTIONAL>*' keyword (Lines 8 and 39-42). 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 39-42). 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 Std. 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.

C. Domain Model and OCL Constraints

PUM uses the stereotypes (i.e., *variation*, *variant*, and *optional*) provided by Ziadi and Jezequel [26] to model variability with domain models (*Challenge 3*) since they support two common mechanisms to specify variability in UML class diagrams, i.e., *optionality* and *variation* (see Fig. 6).

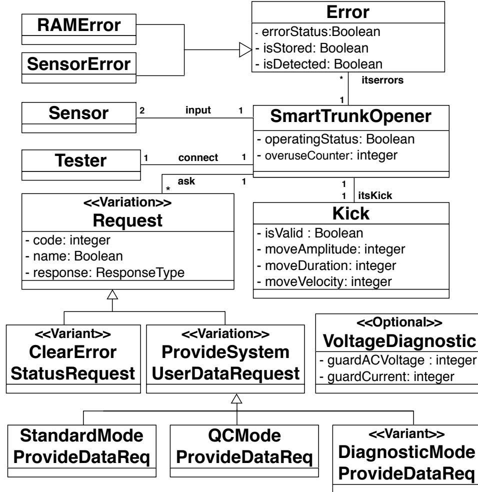

Fig. 6. Simplified Portion of the Domain Model for STO

The stereotypes 'Variant' and 'Variation' are used to explicitly specify variability associated with inheritance hierarchies in domain models. The idea is to define a variation point as an abstract class and variants as concrete subclasses [26]. Since the subclasses *ClearErrorStatusRequest* and *DiagnosticModeProvideDataReq* in Fig. 6 are not mandatory, they are stereotyped 'Variant'. The subclasses *StandardModelProvideDataReq* and *QCModeProvideDateReq* are not stereotyped, thus implying these classes are mandatory for all STO products. We do not use the stereotypes for *Error* and its subclasses since all error types are mandatory in STO. The stereotype 'Optional' is for optional entities which are not part of any inheritance hierarchy (*VoltageDiagnostic* in Fig. 6).

Table III presents some of the use case conditions in Table II with their corresponding OCL constraints referring to the domain model. Having precise definition of use case conditions is crucial to determine the correct execution of the product in terms of flows of events.

VI. Tool Support

We implemented a tool, PUM-C (Product line Use case Modeling - Checker), for checking diagram-specification and specification-template consistency in PUM. PUM-C automatically does the consistency checking and reports inconsistencies such as the diagram missing an include statement in the specification. In addition, it automatically identifies use case conditions (i.e., pre/post conditions and conditions with the keyword 'VALIDATES THAT') and asks the analyst to reformulate them as OCL constraints. To minimize the manual effort, PUM-C first locates conditions in use cases and then identifies repeating and negated ones. If use cases both feature a condition and its negation, the analyst is asked to reformulate only the condition as an OCL constraint. The OCL constraint for the negated condition is automatically derived.

PUM-C relies on NLP and is composed of three layers: *User Interface (UI) Layer*, *Application Layer*, and *Data Layer* (Fig. 7). The *UI Layer* supports creating and updating the PUM 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, IBM Rhapsody (www.ibm.com/software/products/en/ratirhapfami) for domain models, and Eclipse OCL (http://www.eclipse.org/modeling/mdt/ocl/) for writing OCL constraints. To access the *Application Layer* components through the *UI Layer*, we implemented an IBM DOORS plugin.

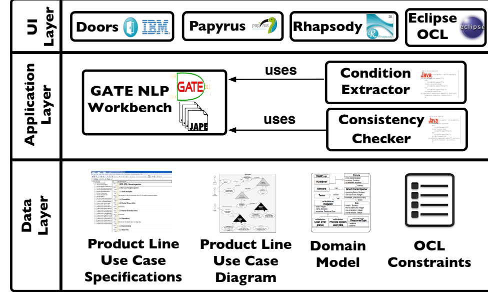

Fig. 7. Layered Architecture of PUM-C

The *Application Layer* contains the components (i.e., *Condition Extractor* and *Consistency Checker*) which we implemented as Java applications for consistency checking and condition extraction. To perform NLP in use case specifications, these two components use a regular expression engine, called JAPE [29], in the GATE workbench (http://gate.ac.uk/), an open-source Natural Language Processing (NLP) framework. JAPE enables to recognise regular expressions in annotations on documents. We implemented the extended RUCM restriction rules in JAPE. In 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 is the annotated use case steps. The *Condition Extractor* and *Consistency Checker* process the annotations and the use case diagram to generate the list of inconsistencies and conditions.

VII. Industrial Case Study and Lessons Learned

We applied PUM to the functional requirements of STO. Our goal was to assess, in an industrial context, how PUM can improve variability modeling practice and how well PUM addresses the challenges that we identified in capturing requirements variability in use cases. STO was proposed for the assessment by the fourth author of the paper since it was a relatively new project at IEE with multiple potential customers requiring different features. IEE provided their initial STO documentation, which contained a use case diagram, use case specifications, and supplementary requirements specifications describing non-functional requirements. To model the STO requirements according to PUM, we first examined the initial STO documentation and then worked with IEE engineers to build and iteratively refine our models. Tables IV and V present the size of the resulting use cases and domain model.

To evaluate the output of PUM in light of the challenges we identified earlier, we had a semi-structured interview with five

TABLE III
SOME OCL CONSTRAINTS FOR THE STO USE CASES

#	Condition in the Use Case	Corresponding OCL Constraint
1	The operating status is valid	`SmartTrunkOpener.allInstances() → forAll(sto | sto.configurationDataStatus = true AND sto.itsECU.isValid = true) AND sto.itsSTOSensors → forAll(snsr | snsr.isValid = true))`
2	Movement is a valid kick	`Kick.allInstances() → forAll(k	k.moveAmplitude > k.minKickAmplitude AND k.moveAmplitude < k.maxKickAmplitude AND k.moveDuration < 2 AND (k.backwardMovement - k.forwardMovement).abs() <= 0.2 AND k.stopMovement = false AND (k.timeDisplacementUpperAntenna > k.timeDisplacementLowerAntenna) AND (k.timeDisplacementUpperAntenna - k.timeDisplacementLowerAntenna) <= 50)`
3	There is no error detected	`Error.allInstances() → forAll(e | e.isDetected = false)`
4	Watchdog reset is valid	`SmartTrunkOpener.allInstances() → forAll(sto | sto.resetCounter < 3 AND sto.itsWatchdog.isEnabled = true)`

TABLE IV
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

participants holding various roles at IEE (i.e., process manager, software development manager, software lead engineer, system engineer, and software engineer). They all had substantial software development experience, ranging from 8 to 17 years. All participants had experience with use case driven development and modeling. The interview included a presentation illustrating the PUM steps, a tool demo, and detailed examples from STO. The presentation was interactive and included questions posed to the participants about the models for them to take a more active role and give us feedback.

TABLE V
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

To capture the perception of engineers participating in the interviews, regarding the potential benefits of PUM, and assess how it addresses the targeted challenges, we handed out a questionnaire including questions to be answered according to a Likert scale [30] (i.e., strongly agree, agree, disagree, and strongly disagree). The questionnaire was structured for the participants to assess PUM in terms of adoption effort, expressiveness, comparison with current practice, and tool support. The participants were also encouraged to provide open, written comments.

Results from the interviews showed that all participants agreed on the following positive aspects of PUM:

- The participants considered the extensions to be simple enough to enable communication between analysts and customers. They also stated that the extensions can also be used for internal communication, e.g., for test engineers to perform regression test selection.
- The participants considered the extensions to provide enough expressiveness to conveniently capture variability in their projects. In our case study, we were able to capture 7 variation points, 13 variants use cases, and 7 variant dependencies.
- The participants considered PUM to provide better assistance for capturing and analyzing variability information compared to the current, more informal practice in their projects. With PUM, 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 the method of *Clear Error Status via IEE QC Mode* is mandatory while the method of *Clear Error Status via Diagnostic Mode* is optional (see Fig. 5), which was not previously documented.
- The participants considered the effort required, to learn how to apply PUM and its tool, to be reasonable. They also stated they expect most of the effort to relate to OCL.
- The participants considered PUM-C to provide useful assistance for minimising inconsistencies in artifacts.

The participants also expressed a number of challenges regarding the application of PUM:

- *Modeling variability in non-functional requirements.* 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 such aspects as well.
- *Training customers for PUM.* Though the participants considered the effort required to learn PUM to be reasonable, training customers may be more of a challenge. The company may need customers' consent to initiate the modeling effort. Thus, the costs and benefits of PUM should be made clear to customers.
- *Imperfect variability information.* When a new project starts, requirements and their variations might be very difficult to identify. As a result, in the beginning, analysts are expected to redefine variation points and variants in requirements specifications through frequent iterations. Such changes on variability need to be managed and supported to enable analysts to converge towards consistent and complete requirements and variability information.
- *Adaptations in the tool chain.* PUM-C is currently implemented as a plugin in IBM DOORS in combination with a leading commercial modeling tool used at IEE, i.e., IBM Rhapsody, and Papyrus. PUM-C highly depends on the outputs of these tools. In time, these tools might be replaced with other tools or the newer versions of the same tools. PUM-C needs to be easily adapted for such changes in the tool chain.

Our discussion with participants resulted in the following extensions being required for PUM and PUM-C:

- *Checking consistency between use cases and domain*

model. Domain entities identified in a use case may be missing in the domain model. PUM-C can be extended to automatically evaluate the domain model completeness and correctness by checking the mapping between domain entities identified by our NLP application and entities in the domain model.

- *Automatic configuration.* A configurator can guide analysts and customers to make decisions regarding variation points, and generate product specific use cases.
- *Integrating non-functional requirements with use cases.* Additional extensions can be introduced into PUM to model non-functional requirements, e.g., response time and synchronization requirements.

Threats to validity. The main threat to the validity of our case study regards the generalizability of the conclusions and lessons learned. To mitigate the threat, we applied PUM to an industrial case study that includes nontrivial use cases in an application domain with multiple potential customers and numerous sources of variability. We selected the respondents to our questionnaire and interviews to hold various, representative roles and with substantial industry experience. To limit threats to the internal validity of the case study, we had many interviews with the IEE engineers in the STO project to verify the correctness and completeness of our models.

VIII. Conclusion

This paper presents a product line methodology centred around use case modeling, called PUM, for documenting variability in use case diagrams and specifications, and associated domain models. Our main motivation is to enable variability modeling by relying exclusively on commonly used artifacts in use-case driven development, thus avoiding unnecessary modeling overhead. PUM builds on and integrates existing work and is supported by a tool employing NLP for checking artifact consistency. The key characteristic of our method is that it captures variability in product lines at a level of granularity enabling both precise communication with various stakeholders, at different levels of details, and guided product configuration. Initial results from structured interviews with experienced engineers suggest that PUM is accurate and practical to capture variability in industrial settings.

As future work, we are considering to integrate in a single tool all aspects of our approach, which are currently supported by multiple modeling tools, i.e., Papyrus, Rhapsody, Eclipse OCL, in the current tool chain. PUM is a first step to achieve our long term objective, i.e., automated configuration and change impact analysis in use case driven development. Our plan for the next stages is to provide automated configuration that guides customers in making configuration decisions and automatically generate use case specifications and diagram for the configured product. To address contexts where products are constantly evolving, we will also support change impact analysis to help analysts properly manage change.

Acknowledgments

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

References

[1] "IEE (International Electronics & Engineering) s.a., http://www.iee.lu/."
[2] G. Halmans and K. Pohl, "Communicating the variability of a software-product family to customers," *SoSyM*, vol. 2, pp. 15–36, 2003.
[3] M. Eriksson, J. Borstler, and K. Borg, "The pluss approach - domain modeling with features, use cases and use case realizations," in *SPLC'05*, 2005, pp. 33–44.
[4] M. Eriksson, J. Borstler, and A. Asa, "Marrying features and use cases for product line requirements modeling of embedded systems," in *SERPS'04*, 2004.
[5] M. Alferez, U. Kulesza, A. Moreira, J. Araujo, and V. Amaral, "Tracing between features and use cases: A model-driven approach," in *VA-MOS'08*, 2008.
[6] B. Gallina and N. Guelfi, "A template for requirement elicitation of dependable product lines," in *REFSQ'07*, 2007, pp. 63–77.
[7] C. Nebut, Y. L. Traon, and J.-M. Jezequel, "System testing of product families: from requirements to test cases," in *Software Product Lines*. Springer, 2006.
[8] A. Fantechi, S. Gnesi, G. Lami, and E. Nesti, "A methodology for the derivation and verification of use cases for product lines," in *SPLC'04*, 2004, pp. 255–265.
[9] S. Buhne, G. Halmans, and K. Pohl, "Modeling dependencies between variation points in use case diagrams," in *REFSQ'03*, 2003, pp. 59–69.
[10] S. Azevedo, R. J. Machado, A. Braganca, and H. Ribeiro, "The UML "extend" relationship as support for software variability," in *SPLC'10*, 2010, pp. 471–475.
[11] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the transition from use case models to analysis models: Approach and experiments," *ACM TOSEM*, vol. 22, no. 1, 2013.
[12] "Object Constraint Language (OCL)," http://www.omg.org/spec/OCL/.
[13] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2001.
[14] K. Pohl, G. Bockle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
[15] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating feature modeling with the rseb," in *ICSR'98*, 1998, pp. 76–85.
[16] A. Braganca and R. J. Machado, "Automating mappings between use case diagrams and feature models for software product lines," in *SPLC'07*, 2007, pp. 3–12.
[17] M. Eriksson, H. Morast, J. Borstler, and K. Borg, "The pluss toolkit - extending telelogic doors and ibm-rational rose to support product line use case modeling," in *ASE 2005*, 2005, pp. 300–304.
[18] M. Eriksson, J. Borstler, and K. Borg, "Managing requirements specifications for product lines - an approach and industry case study," *Journal of Systems and Software*, vol. 82, pp. 435–447, 2009.
[19] S. Buhne, G. Halmans, K. Lauenroth, and K. Pohl, "Scenario-based application requirements engineering," in *Software Product Lines*. Springer, 2006.
[20] W. Jirapanthong and A. Zisman, "Xtraque: traceability for product line systems." *SoSyM*, vol. 8, no. 1, pp. 117–144, 2009.
[21] R. Biddle, J. Noble, and E. Tempero, "Supporting reusable use cases," in *ICSR'02*, 2002, pp. 210–226.
[22] A. Fantechi, S. Gnesi, I. John, G. Lami, and J. Dorr, "Elicitation of use cases for product lines," in *PFE'03*, 2004, pp. 152–167.
[23] T. von der Maßen and H. Lichter, "Modeling variability by uml use case diagrams," in *REPL'02*, 2002, pp. 19–25.
[24] S. Azevedo, R. J. Machado, A. Braganca, and H. Ribeiro, "On the refinement of use case models with variability support," *Innovations Syst Softw Eng*, vol. 8, pp. 51–64, 2012.
[25] I. John and D. Muthig, "Product line modeling with generic use cases," in *EMPRESS 04*, 2004.
[26] T. Ziadi and J.-M. Jezequel, "Product line engineering with the uml: Deriving products," in *Software Product Lines*. Springer, 2006.
[27] H. Gomaa, "Object oriented analysis and modeling families of systems with uml," in *ICSR-6*, 2000, pp. 89–99.
[28] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 2002.
[29] H. Cunningham et al, "Developing language processing components with gate version 8 (a user guide), http://gate.ac.uk/sale/tao/tao.pdf."
[30] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. Continuum, 2005.