Introducing Product Line Engineering in a Bottom-Up Approach

Nelufar Ulfat-Bunyadi, Rene Meis, Nazila Gol Mohammadi, Maritta Heisel
paluno - The Ruhr Institute for Software Technology, University of Duisburg-Essen, Duisburg, Germany
{firstname.lastname}@paluno.uni-due.de

Keywords: Product line engineering, variability, control system, embedded system, Six-Variable Model

Abstract: The optimal way for introducing a product line is to set up a completely new product line by developing a reuse infrastructure for the whole range of products right from the start. However, in practice, product line engineering is frequently introduced by a company after having developed a number of products separately (i.e. in single system engineering). The challenge then consists of defining the product line based on these existing products, i.e. to a certain extent these products have to be re-engineered. More precisely, two problems need to be solved: first, commonality and variability among the existing products needs to be identified to define a common set of core assets, and, second, the way in which future systems (i.e. products of the product line) will be developed based on this common set of assets needs to be defined. The method we suggest in this paper solves these two problems. Our method focuses on control systems, i.e. systems which monitor/control certain quantities in their environment.

1 INTRODUCTION

A (software) product line is defined as follows (Clements and Northrop, 2002): "*A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment and that are developed from a common set of core assets in a prescribed way.*" The process of developing a product line is referred to as product line engineering (Pohl et al., 2005). Companies introduce product line engineering due to diverse reasons, for example, in order to improve time-to-market, to improve product quality, to increase customer satisfaction, to enable mass-customization, etc. (Clements and Northrop, 2002). In a nutshell, substantial production economies can be achieved by developing systems in a product line instead of separately. Depending on the situation, different strategies may be used to introduce a product line (Schmid and Verlage, 2002). Ideally, no predecessor products exist and a new product line is set up right from the start by developing a reuse infrastructure for all products of the product line. Yet, in practice, companies often use a more incremental approach (Schmid and Verlage, 2002): frequently, predecessor products exist that need to be integrated into product line development. Therefore, they need to be re-engineered. In this paper, we present a method for introducing product line development in such a situation. Our method supports developers in identifying commonalities and variability in the existing products, defining core assets for the product line (restricted to requirements artefacts), and defining how future products can be derived in the product line. We focus mainly on control systems.

The paper is structured as follows. In Sect. 2, we present fundamentals which provide the basis of our method. In Sect. 3, we describe our method and the tool support we provide. In Sect. 4, we describe the application of our method to a real example. In Sect. 5, we discuss related work. Finally, in Sect. 6, we provide a conclusion and an outlook on future work.

2 FUNDAMENTALS

Context and Problem Diagrams. Context and problem diagrams have been introduced by Jackson (Jackson, 2001). He differentiates between the system, the machine, and the environment. A system is a general artefact that might have both, manual and automatic components. The machine is the computer-based artefact of the system and is the target of software development. The environment is a portion of the real world that is becoming the environment of the development project, because its current behaviour is unsatisfactory. The machine will be connected to this environment so that the behaviour of the environment becomes satisfactory. According to Jackson's

approach (Jackson, 2001), first a context diagram is created showing the machine in its environment. Then the overall software development problem is decomposed into subproblems and each subproblem is documented in a problem diagram. A context diagram usually consists of the following modelling elements: the machine domain, problem domains, and interfaces between them (see Fig. 8 for an example). The machine domain is the software-to-be. A problem domain represents a material or immaterial object in the environment (e.g. people, other systems, a physical representation of data). An interface expresses that phenomena (e.g. events, states, values) are shared between the domains it connects. At the interface, the shared phenomena are annotated and, by means of an exclamation mark, the domain controlling them is indicated. For creating problem diagrams, the same modelling elements are used. In addition, a problem diagram contains a requirement which is to be satisfied by the machine domain and the problem domains shown in the problem diagram (see Fig. 9 for an example). The requirement is connected to the problem domains by means of at least one constraining reference, and optionally a requirement reference. The latter means that the requirement refers somehow to the domain phenomena. The former means that the requirement even constrains the domain phenomena. In context and problem diagrams, so called connection domains may be modelled as well. A connection domain is "*a domain that is interposed between the machine and a problem domain*" (Jackson, 2001). Examples of connection domains are sensors and actuators. They connect the machine to the environmental domains that are monitored/controlled. According to Jackson, they can be omitted in context/problem diagrams if they are reliable.

The Six-Variable Model. We introduced the Six-Variable Model in previous work (Ulfat-Bunyadi et al., 2016). It is based on the famous Four-Variable Model for control systems defined by Parnas and Madey (Parnas and Madey, 1995). The four variables are monitored, controlled, input, and output variables. Monitored variables *m* are environmental quantities the control software monitors through input devices like sensors. Controlled variables *c* are environmental quantities the software controls through output devices like actuators. Input variables *i* are data items that the software needs as input, and output variables *o* are quantities that the software produces as output. We made the observation that in practice it is sometimes not sufficient to document only the four variables. As requirements are refined and the decision is made which sensors/actuators/other systems to use for monitoring and

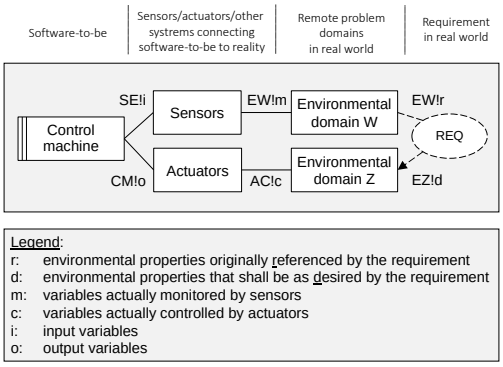

Figure 1: Six-Variable Model (Ulfat-Bunyadi et al., 2016)

controlling, the environmental quantities that have been relevant in the requirements at first (i.e. before decision making) are replaced by the environmental quantities that can actually be monitored/controlled by the selected sensors/actuators/other systems. Existing approaches (like the Four-Variable Model) only call for documenting the environmental quantities that are finally relevant and that are actually monitored/controlled. The quantities that have been relevant originally, are not documented. This results in problems when the software shall be reused in another context/environment with slightly different sensors/actuators. Then it is hard for developers to decide, which environmental quantities still need to be monitored/controlled and which ones not. Therefore, we argue that six variables should be documented: the four variables *i*, *o*, *m*, *c* and additionally *r* and *d* (see Fig. 1). *r* represents the environmental quantities that were originally *referenced* in the requirement and *d* represent the environmental quantities that shall be as *desired* by the requirement.

In Fig. 2, an example of documenting the six variables is given. We used the famous patient monitoring system as an example here. The machine is intended to notify a nurse if the patient's heartbeat stops. The machine is connected to a sensor and an actuator. The sensor detects the sound in the patient's chest. If the patient's heart has stopped beating, the sound from the patient's chest falls below a threshold for a certain time. Then, the machine raises the sound of the buzzer to inform the nurse. The referenced variable is the patient's heartbeat, the monitored variable is the sound in the patient's chest, and the input variable is the corresponding value measured by the sensor. The output variable is 'sound on/off', the controlled variable is the buzzing of the buzzer, and the desired variable is that the nurse is informed.

Essence versus Incaration. The differentiation between essence and incarnation of a system was introduced as part of Essential Systems Analsis in 1984

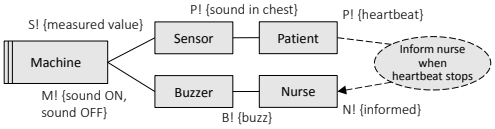

Figure 2: Exemplary documentation of six variables

(McMenamin and Palmer, 1984). The *essence* of a system comprises the capabilities it must possess to fulfil its purpose, regardless of how it is implemented. The *incarnation* comprises all implementation details. For identifying the essence of a system, the following heuristic is used: One assumes that the technology within the system is perfect, this means that processors, for example, are able to do anything constantly and containers (data stores) are able to hold an infinite amount of data. The technology outside the system is not assumed to be perfect.

3 BOTTOM-UP METHOD

3.1 Method Steps

Fig. 3 provides an overview of our method. We explain each step in the following in more detail. The main idea is that we model, for each existing system, the essence on the one hand (Steps 1 and 2) and, independent of that, the incarnation on the other hand (Step 3). Creating an incarnation model means that the corresponding system is modelled as it is. Creating an essential model is more difficult, since the information required to create the model is frequently not documented and existed only at the beginning of the system development project, in which the corresponding system was developed. Mining this knowledge is therefore more complicated. Confronting the essential model with the incarnation model helps developers to reflect on the quantities that are monitored/controlled and on the reasons why this is done, i.e. which environmental phenomena originally have been relevant in the real world.

Step 1: Create essential context diagrams. As a first step, an essential context diagram needs to be created for each existing system. To focus on the essence, we use a heuristic: we assume that the technology outside the machine is perfect. This means that we assume that all connection domains are reliable and we do not model them therefore. A major challenge in Step 1 consists in identifying the environmental domains that are actually relevant in the real world. Sometimes they are not made explicit in existing documentations. The main question to be answered is: What shall be monitored/controlled in the environment? In this step, it is not important how this

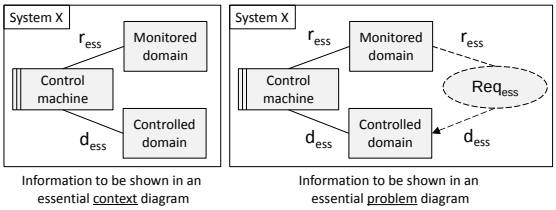

Figure 4: Information in essential diagrams

is achieved, since we focus on the essence. Therefore, in the context diagram (see Fig. 4, left hand side), only the problem domains that are relevant in the real world are shown and, at the interfaces to the machine, the r and d variables are annotated that are relevant from an essential view. We call them r_{ess} and d_{ess}.

Step 2: Create essential problem diagrams. Based on the essential context diagrams from Step 1, essential problem diagrams are created during this step. For each essential context diagram, usually several essential problem diagrams are created, since the considered machine usually has to satisfy several requirements and each requirement is modelled in a separate problem diagram. The right hand side of Fig. 4 shows the information that needs to be shown in an essential problem diagram. Note that we still abstract from connection domains and focus on the r_{ess} and d_{ess} variables. Since we assume perfect technology (i.e. perfect sensors and actuators), the requirement refers to/constrains the same phenomena as the ones shared with the machine. Note that the requirement (Req_{ess}) represents an essential requirement, i.e. it describes what shall be achieved in the real world without being biased by the technology to be used.

Step 3: Create incarnation problem diagrams. Independently of the diagrams from Step 1 and 2, incarnation problem diagrams are created during this step for each system. During this step, we take implementation details into account and, therefore, we model all connection domains. The diagram on the left-hand side of Fig. 5 shows which information needs to be shown in an incarnation problem diagram. At the interfaces between the machine, connection domains (sensors/actuators), and environmental domains (monitored/controlled domains), we annotate now the six variables from an incarnation view ($r_{inc}, m_{inc}, i_{inc}, o_{inc}, c_{inc}, d_{inc}$). Creating these incarnation diagrams independently of the essential diagrams has the benefit that we once focus on the r and d variables from an essential view and the other time we focus on them from an incarnation view. In this way, it is more likely that we find differences between r_{ess} and r_{inc} as well as d_{ess} and d_{inc} variables. Note that we welcome such differences. Essence and incarna-

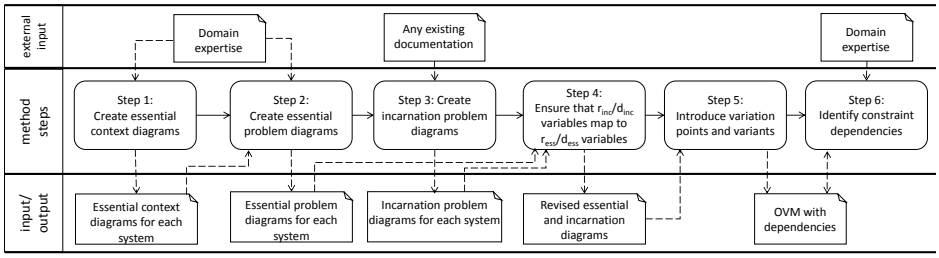

Figure 3: Overview of our method

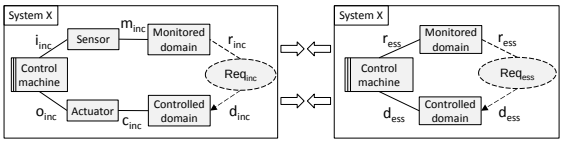

Figure 5: Mapping essence and incarnation

tion are often different. It is only important that the difference is traceable and therefore we document it now, i.e. in retrospect.

Step 4: Ensure that r_{inc}/d_{inc} variables map to r_{ess}/d_{ess} variables. During this step, we try to map the r_{inc}/d_{inc} variables to the r_{ess}/d_{ess} variables (see Fig. 5), i.e. we try to answer the question, why each r_{inc} is monitored and why each d_{inc} is effected by identifying their relations to certain r_{ess}/d_{ess} variables. The reasoning explaining, why the variables map, is documented textually. At the end of this step, for each r_{inc}/d_{inc} variable, such a documentation must exist. Thus, the term *mapping* does not mean that r_{inc} and r_{ess} as well as d_{inc} and d_{ess} variables need to be the same. Rather, it means that r_{inc} needs to reflect actually r_{ess} and that d_{inc} needs to result actually in d_{ess}. Even the environmental domains may be different to a certain extent, e.g. a domain shown in the essential diagram may be decomposed and shown as two domains in the incarnation diagram. This is allowed but should be traceable and documented in the reasoning. If the discrepancy is not traceable, one should consider revising one/both diagrams to identify the cause and correct the diagrams from Steps 1-3.

Step 5: Introduce variation points and variants. Based on the (probably revised) diagrams from Step 4, variants and variation points are introduced. Usually there are different variants for one r_{ess}/d_{ess} variable, since different sensors/actuators may be used. Therefore, we recommend to create incarnation diagram fragments for the variants as shown in Fig. 6. To depict variation points and variants (i.e. the choices that can be made), we suggest creating an OVM (Orthogonal Variability Model) (Pohl, 2010). The variation points and variants are then related to

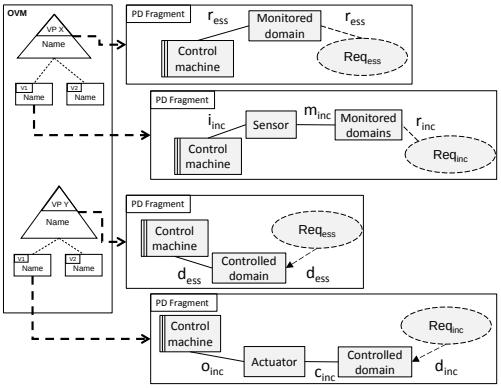

Figure 6: Diagram fragments as variants

the essential and incarnation problem diagram fragments by means of so called artefact dependencies.

Step 6: Identify constraint dependencies. Usually, the selection of variants at one variation point may affect the available choices at another variation point: a variation point may require/exclude another variation point, a variant may require/exclude another variant, and a variant may require/exclude a variation point (cf. (Pohl, 2010)). Such relationships are called constraint dependencies. Since we create the OVM based on the (essential and incarnation) problem diagrams, it is likely that not all constraint dependencies that actually exist, will directly be identified. Therefore, we introduce this step. During this step, the developers must reflect on constraint dependencies that might exist and document them.

After Step 6, the product line is defined and initiated. To derive concrete products of the product line, developers need to select variants and to compose the related essence and incarnation diagram fragments to whole context/problem diagrams for their product.

3.2 Tool Support

By means of our method, the OVM as well as core assets for the product line are created (in Steps 5 and 6). The tool support we present in this section helps developers in creating and maintaining these core assets

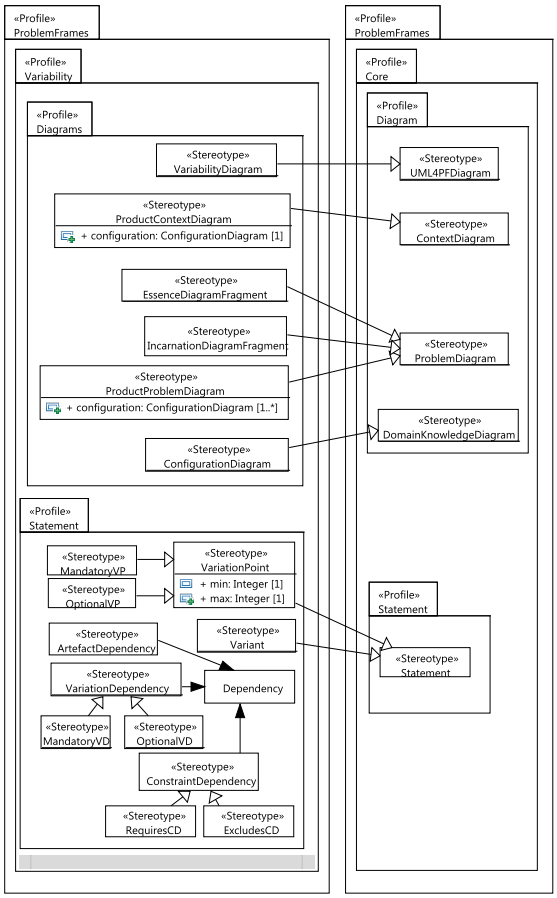

Figure 7: OVM Profile extending UML4PF

and the OVM together with its artefact and constraint dependencies. We use the UML4PF profile proposed by Hatebur and Heisel (Hatebur and Heisel, 2010) as a basis. This profile allows describing context and problem diagrams using UML class diagrams. It has already been extended by Alebrahim et al. to support variability (cf. (Alebrahim et al., 2014)). Yet, this variability profile does not support the orthogonal modelling of variability. To support orthogonal variability modelling, we adapted the variability profile and present the resulting OVM profile (see Fig. 7) in the following. Due to space limitation, we mainly explain the adaptations we made.

The variability diagram captures the variation points and variants (i.e. it represents the OVM). The essence and incarnation diagram fragments are specialisations of problem diagrams. The configuration diagram captures the selected variants for a certain product of the product line. Product context and product problem diagrams are diagrams that are created when deriving products of the product line. Therein, essence/incarnation diagram fragments are composed to build whole context/problem diagrams. An important extension that we made, which allows for modelling the variability orthogonally to the other diagrams, is that we defined artefact dependency as a specialisation of dependency. This allows for modelling artefact dependencies between the OVM and diagram fragments. Furthermore, we maintain variation points and variants exclusively in the variability diagram. They are not scattered across various diagrams.

4 APPLICATION EXAMPLE

As examples, we use ACC (Adaptive Cruise Control) systems. Real ACC systems are described in (Robert Bosch GmbH, 2003) and (Robert Bosch GmbH, 2006). The description of the three systems we use in the following is based on these documents. We assume that they are existent and, based on them, product line engineering is introduced:

System 1: Simple ACC. System 1 uses a long range radar (LRR) sensor and ESP (Electronic Stability Program) sensors. The LRR provides information about speed, distance, and lateral offset of objects ahead. ESP sensors measure the yaw rate, lateral acceleration, steering wheel angle, and wheel speed of the ACC vehicle. Based on this data, the ACC software is able to calculate the yaw rate corrected for offset which is needed to determine the projected course of the ACC vehicle. Based on the projected course of the ACC vehicle and the lateral offset of a detected object, the lane of the object can be estimated. If the detected object is on the same lane as the ACC vehicle, it is identified as a target object (for tracking). If it is not on the same lane, it is considered to be irrelevant. However, the ACC software (using these sensors) is not able to decide with sufficient certainty, whether or not an object is in the same lane as the ACC vehicle. Therefore, stationary objects are ignored by the ACC constant-gap function, i.e. only moving objects can be selected as target objects for tracking. Due to the large number of stationary objects at the roadside, the likelihood for the ACC software reacting to one of them by mistake is very high.

System 2: Advanced ACC. System 2 uses an LRR sensor as well. Instead of the ESP sensors, a stereo video sensor is used. As in System 1, the LRR provides information about speed and distance of objects ahead. Yet, we do not need the lateral offset of detected objects, because the video sensor identifies the lane of them precisely. In addition, the video sensor recognizes object dimensions and is thus able to differentiate between vehicles, people, beverage cans, etc. Due to the precise information about the lane of a

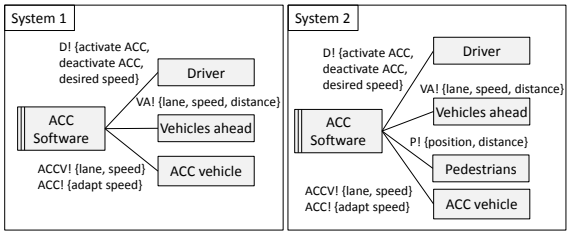

Figure 8: Essential context diagrams (Step 1)

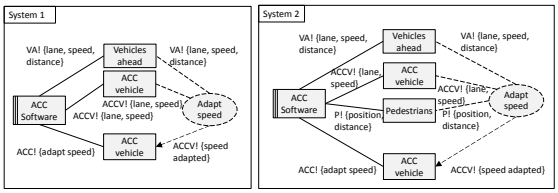

Figure 9: Essential problem diagrams (Step 2)

detected object (at the roadside and on the road), the ACC software is able to take stationary objects into account in the constant-gap function. Due to the reliable information about stationary and moving objects that is now available, the ACC software in System 2 supports also emergency braking.

System 3: Sophisticated ACC. System 3 is similar to System 2, except for the short range radar (SRR) sensors that are used in addition. The SRR sensors detect vehicles ahead that are close to the ACC vehicle (e.g. vehicles cutting in sharply). Due to this information, the ACC software supports in addition (to the capabilities that are also supported by System 2) stop-and-go in urban traffic.

We now describe how we applied our method to identify variability and commonality and to define core assets for our product line of ACC systems.

Step 1: Create essential context diagrams. We have created an essential context diagram for Systems 1, 2, and 3 by abstracting from connection domains and identifying the r_{ess} and d_{ess} variables, i.e. the environmental quantities that are actually relevant in the real world regardless of the sensors/actuators that will be used to monitor/control them. The context diagrams for System 1 and 2 are given in Fig. 8. The diagram for System 3 is the same as for System 2. This is possible because, as regards the essence, the same problem domains in the environment are relevant for System 3. For all three systems the driver, vehicles ahead, and the ACC vehicle are relevant problem domains in the environment. For System 2 and 3, in addition, pedestrians are relevant.

Step 2: Create essential problem diagrams. Based on the essential context diagrams from Step 1, we created essential problem diagrams. Note that to each essential context diagram, several essential problem diagrams were created because each machine was decomposed into submachines and each submachine was modelled in another problem diagram with the corresponding essential requirement it has to satisfy. Fig. 9 shows two of these essential problem diagrams for System 1 and System 2. The essential problem diagram for System 3 is again the same as for System 2 due to the same reason as in Step 1. The essential requirement to be satisfied is "adapt speed". In case of System 1, the ACC software needs to know the lane, speed, and distance of vehicles ahead as well as the lane and the speed of the ACC vehicle. These are the phenomena referred to by the requirement. The phenomenon constrained by the requirement is the speed of the ACC vehicle indicated by "speed adapted". In case of System 2, the same requirement has to be satisfied but there is one more problem domain involved: pedestrians. The reason is that, in contrast to System 1, the ACC software in Systems 2 and 3 is able to differentiate between pedestrians and vehicles ahead.

Step 3: Create incarnation problem diagrams. Independent of the diagrams created in Steps 1 and 2, we modelled the incarnation of the three existent ACC systems. The incarnation problem diagrams for System 1 and 2 are given in Fig. 10. We omitted the diagram for System 3 due to space limitation. The diagram for System 3 resembles the one for System 2, but contains additionally the SRR sensors and the corresponding shared phenomena. As Fig. 10 shows, the diagrams contain now the sensors and actuators they use for monitoring and controlling problem domains in the environment. Furthermore, at the interfaces, the six variables $r_{inc}, m_{inc}, i_{inc}, o_{inc}, c_{inc}, d_{inc}$ are annotated. For example, the incarnation problem diagram for System 1 shows the LRR and the ESP sensors that are used for monitoring. The LRR measures lateral offset, speed, and distance of vehicles ahead. The ESP sensors measure the yaw rate, lateral acceleration, wheel speed, and steering wheel angle of the ACC vehicle. All these input variables are needed by the ACC software in order to know the relative position, speed, and distance of vehicles ahead (referenced variables). This is different in System 2. As the incarnation problem diagram in Fig. 10 shows, different sensors and a different set of input variables are used.

Step 4: Ensure that r_{inc}/d_{inc} variables map to r_{ess}/d_{ess} variables. During this step, we contrast the incarnation of each system with the essence of the system, for example, the incarnation of System 1 shown on in Fig. 10 with its essence shown in Fig. 9. As stated above, for each r_{inc}/d_{inc} variable, a reasoning must be documented. We illustrate that exemplar-

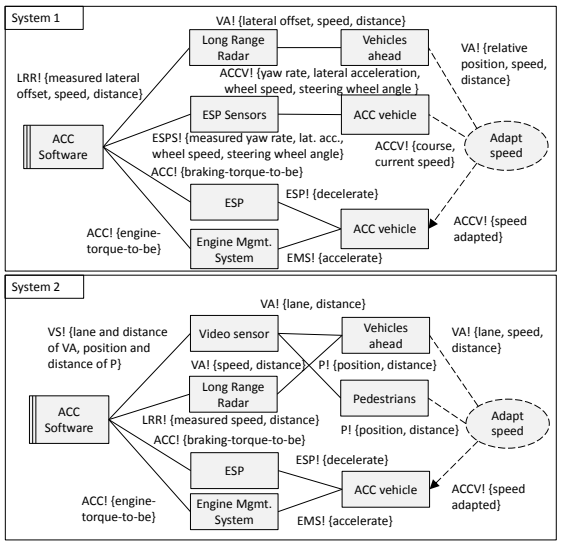

Figure 10: Incarnation problem diagrams (Step 3)

ily for one r_{ess} variable, the lane of vehicles ahead:

> *Based on the yaw rate, lateral acceleration, wheel speed, and steering wheel angle of the ACC vehicle (m_{inc} variables), the yaw rate corrected for offset can be calculated. Based on this value, the course of the ACC vehicle (r_{inc} variable) can be determined. Based on the course of the ACC vehicle (r_{inc} variable) as well as the lateral offset of vehicles ahead (m_{inc} variable), the relative position of vehicles ahead can be determined (r_{inc} variable). The relative position is an estimation of the lane of vehicles ahead (r_{ess} variable).*

Step 5: Introduce variation points and variants. During this step, we introduce variants (incarnation diagram fragments) and variation points (essence diagram fragments) and create an OVM. The OVM in Fig. 11 shows the variation point 'Identifying vehicles ahead' with the three variants V1 to V3. At least one and at most one of the variants must be selected. The variation point is related by means of an artefact dependency to the corresponding essence diagram fragment. Each variant is also related by means of an artefact dependency to the corresponding incarnation diagram fragment. For creating the diagrams shown in Fig. 11 and the artefact dependencies between them, we used our tool that is described in Sect. 3.2.

5 RELATED WORK

There are existing approaches that provide support in introducing a product line based on existing systems or legacy systems. Li and Chang (Li and Chang, 2009) describe also a bottom-up approach for initiating product line engineering but they mainly focus on process-related aspects regarding the company/organisation (e.g. building teams and institutionalizing new processes that replace traditional ones). Kang et al. (Kang et al., 2005) present a method for feature-oriented re-engineering of legacy systems into product line assets. Yet, their focus is on architectural design while we focus on requirements engineering. Ferber et al. (Ferber et al., 2002) describe a re-engineering approach for an entire product line. They define a method to investigate feature dependencies and interactions, which restrict the variants that can be derived from the legacy product line assets. Their approach is interesting, although they consider a different project situation than we do (they assume there is legacy product line). Yet, their approach for investigating feature dependencies and interactions may probably complement our method as regards the identification of constraint dependencies (Step 6). We will consider that in future work. Beyond these approaches, there is one paper of ourselves (Ulfat-Bunyadi et al., 2016) that has similarities with the work presented in this paper. Nevertheless, the main difference is that we present in this paper a bottom-up approach for introducing a product line. The method presented in (Ulfat-Bunyadi et al., 2016) describes a top-down approach. Furthermore, the latter solves a different problem: which information to document about a control software to enable later systematic reuse of the software.

6 CONCLUSION

In this paper, we presented a bottom-up approach for introducing product line engineering by re-engineering existent systems. To this end, we modelled the essence of each system independent of its incarnation and confronted the two models. We believe that reflecting on the essence of similar existing systems will help developers in identifying commonalities and variability among the systems. Our method provides support in creating the core assets of the product line and the variability model. We validated our method using real examples and provide tool support. In future work, we plan to extend our tool support. We presented here the OVM profile which enables orthogonal variability modelling. We would like to add OCL (Object Constraint Language) constraints to check validation conditions. Furthermore, we plan to analyse how difficult it is to create an essential model for an existing system. To this end, we plan experiments in student groups.

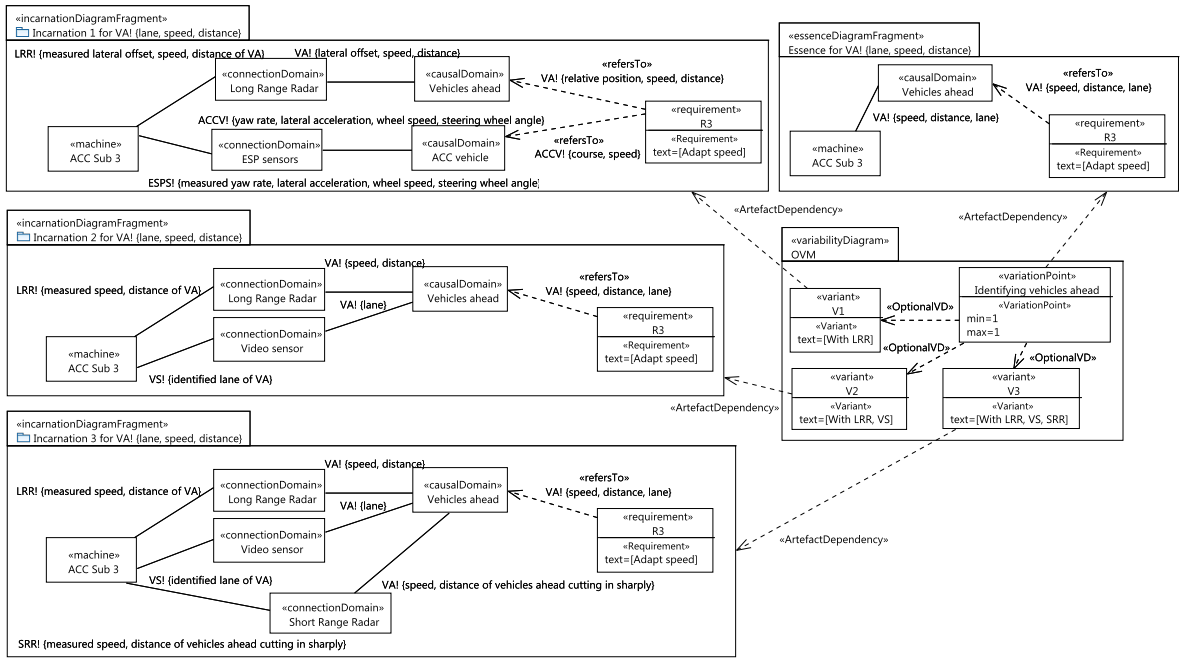

Figure 11: Exemplary variants and variation points for ACC product line (Step 5)

REFERENCES

Alebrahim, A., Fassbender, S., Filipczyk, M., Goedicke, M., Heisel, M., and Konersmann, M. (2014). Towards a computer-aided problem-oriented variability requirements engineering method. In *Proc. CAiSE 2014 Workshops*, number 178 in LNBIP, pages 136–147. Springer.

Clements, P. and Northrop, L. (2002). *Software Product Lines - Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

Ferber, S., Haag, J., and Savolainen, J. (2002). Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Proc. SPLC 2002*, number 2379 in LNCS, pages 235–256.

Hatebur, D. and Heisel, M. (2010). A uml profile for requirements analysis of dependable software. In *Proc. SAFECOMP 2010*, number 6351 in LNCS, pages 317–331. Springer.

Jackson, M. (2001). *Problem Frames - Analysing and Structuring Software Development Problems*. Addison-Wesley.

Kang, K., Kim, M., Lee, J., and Kim, B. (2005). Feature-oriented re-engineering of legacy systems into product line assets – a case study. In *Proc. SPLC 2005*, number 3714 in LNCS, pages 45–56.

Li, D. and Chang, C. K. (2009). Initiating and institutionalizing software product line engineering: from bottom-up approach to top-down practice. In *Proc. Annual IEEE Intl. Computer Software and Applications Conference*, pages 53–60. IEEE Computer Society.

McMenamin, S. M. and Palmer, J. (1984). *Essential Systems Analysis*. Prentice Hall, London.

Parnas, D. and Madey, J. (1995). Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61.

Pohl, K. (2010). *Requirements Engineering- Fundamentals, Principles, and Techniques*. Springer.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition.

Robert Bosch GmbH (2003). *ACC Adaptive Cruise Control - The Bosch Yellow Jackets*. Edition 2003 edition.

Robert Bosch GmbH (2006). *Safety, Comfort and Convenience Systems. Function, Regulation and Components*. John Wiley and Sons.

Schmid, K. and Verlage, M. (2002). The economic impact of product line adoption and evolution. *IEEE Software*, 19(4):50–57.

Ulfat-Bunyadi, N., Meis, R., and Heisel, M. (2016). The six-variable model – context modelling enabling systematic reuse. In *To be published in Proc. of ICSOFT-PT 2016*.