An Approach to Detect the Origin and Distribution of Software Defects in an Evolving Cyber-Physical System

Christian Manz[1], Michael Schulze[2], and Manfred Reichert[3]

[1] Group Research & Advanced Engineering Daimler AG, Germany
christian.c.manz@daimler.com
[2] pure-systems GmbH, Germany
michael.schulze@pure-systems.com
[3] Institute of Databases and Information Systems University of Ulm, Germany
manfred.reichert@uni-ulm.de

Abstract. Cyber-Physical Systems (CPS) are usually developed by an incremental approach. A changing environment like demanding user requirements or legislation amendments lead often to multiple development paths in an evolving CPS. Hence, software variability plays an increasingly important role adapting the characteristics of such CPS to different contexts. This paper focuses on software variability realized through a Software Product Line (SPL) more specifically. Thereby, variability and evolution are usually managed in different tools. However with respect to software defects, a holistic handling of variability and evolution is necessary to ensure a reliable software defect removal. Particularly, detecting software defects in different evolution stages and derived variants is ordinary, but complex and error-prone. To close the gap between variability and evolution, this paper presents a systematic approach to combine both disciplines. In particular, we apply existing variant management techniques in combination with software configuration management methods to determine a software defect's origin and distribution in an evolving SPL. We apply our approach to a CPS from the automotive domain to show its industrial relevance and usefulness.

Keywords: Software Product Line, Evolution, Maintenance

1 Introduction

Cyber-Physical Systems (CPS), like advanced automobile systems, modern medical systems, and progressive electric power grids, connect computational entities in a collaborative manner with physical processes [1, 2]. In more detail, advanced automobile systems interact with digital networks, like the cyberspace, to realize new in-vehicle services, increase road safety, and encourage an efficient control of the growing traffic volume. CPS typically operates in different as well as partially predictable contexts and comprises a plurality of already existing embedded systems. Thereby, variability plays an increasingly important role adapting

the characteristics of such CPS. This paper focuses on software variability of a CPS more specifically. With it, Software Product Lines (SPL) are used to realize similar software variants of a CPS in an effective and manageable way. In the following, two software engineering challenges (variability and evolution) are described in more detail in the context of a CPS.

First, SPL facilitate a planned and systematic reuse of software artifacts in similar circumstances (e.g., different vehicles, markets). Benefits of a SPL include improved quality results, reduced costs, and shorter time-to-market aspects with respect to adaptable software products [3–6]. Common as well as variant specific software parts across a SPL constitute a challenge in software engineering projects. While common parts are included in every product (e.g., every vehicle has an engine), variability describes different shapes of a product at the same time (e.g., standard, sportive, classic) [4]. The common parts, enhanced by a selection of different variable objects, characterizes a software variant of a SPL.[1] In the context of CPS, software variability plays an increasingly important role adapting their characteristics to different contexts. Thereby, a comprehensive reuse of software between similar, but not identical CPS is a common objective. In the following, this paper focuses on SPLs in the context of CPSs.

Second, a continuous changing environment demands a successive adaption of a CPS in industrial projects. In more detail, an evolving CPS leads to a successive adaption of the software and results in an evolving SPL. Maintainability, traceability, and consistency get increasingly important and are affected by the additional complexity of an evolving system [7]. Such evolution depends on a changing context of a SPL, like emerging user requirements or legislation changes, and results in an incremental development process of a long running SPL [8]. Enhancements for only a subset of software variants at a specific time or maintenance of SPLs evolution stages may lead to multiple development paths. Each development path can have its individual number of evolution stages. Thereby, evolution of SPLs is more complex than in single software product development, through additional variability aspects [9]. On the one hand, software configuration management systems are well-known, established and mature tools. On the other, they do not explicitly distinguish between variability and evolution [3]. Besides that, many existing SPL approaches assume a fairly stable environment, which cannot be assumed in an industrial environment. The variability characteristics of a SPL can evolve like other software. With it, existing SPL approaches often disregard evolutionary aspects [8]. Certainly, several industrial use cases require a holistic consideration of an evolving SPL. This paper focuses on one common use case: Maintain an evolving SPL in case of

[1] As many terms in software engineering, the term *variant* is overloaded with different meanings. In this context, we understand variants as similar software products at a specific time. In contrast, *versions* are used to handle consecutively evolution stages of a software element. A version describes the characteristics of a software element at a specific time. For a better comprehension, we use the term *evolution stage* for different versions of a SPL.

a systematic removal of software defects. Therefore, a precise understanding of variability and evolution dependencies is required.

This paper describes an approach to find the origin of a detected software defect and determines potentially affected variants across all evolution stages of a SPL. Therefore, a configuration management system and a variant management system will be utilized together, supporting software experts to identify an *incorrect software object (iso)* in a specific SPL evolution stage. An *iso* represents an identifiable software object (e.g., a specific requirement, system design element, or source code object) or an object of the appropriated variability model (e.g. features, restrictions) in an evolving SPL. Further, software experts are supported in finding the origin and the distribution of a software defect in an evolving SPL. Thereby, the origin characterizes the initial faulty SPL evolution stage of the *iso*. Finally, software experts are guided to determine evolved, potentially affected, and derived variants in each SPL evolution stage of that origin.

The rest of this paper is organized as follows: Section 2 describes a small part of a CPS and their related evolving SPL to understand the evolution observations and real world scenarios. Section 3 represents our approach to face industrial challenges, using existing variant management techniques in combination with a software configuration management system. In order to guarantee the functioning of our approach preconditions are characterized. Further, initial tasks are described to ensure a reliable software defect removal. Besides, a workflow description illustrates our approach in a software expert point of view. In Section 4, we apply our approach to an application scenario to illustrate the benefits. Section 5 discusses related work. The paper is summarized in Section 6 and exhibits future research.

2 Industrial Example

To understand our industrial observations regarding CPS evolution, Figure 1 illustrates a small part of an evolving Advanced Driver Assistant System (ADAS), based on several CPS and model-driven development projects from the automotive domain [10]. This paper focuses on the software functionality of an ADAS. The software encompassed SPL architecture. The functionality was incrementally extended due to several evolution stages. In our experience, SPLs in the automotive domain typically consist of a double-digit number of evolution stages. Thereby, the ADAS comprises multiple comfort and safety functions to assist a car driver in usual traffic scenarios. More specifically, it provides basic functions (e.g., cruise control and break assistant) in the first evolution stage and advanced functions (e.g., autonomous driving and autonomous emergency braking) in later evolution stages. The complete ADAS variability is managed in a variability model [11] which comprises several hundred features and individual dependencies (e.g., autonomous driving requires signals of the high-end EE-architecture). The ADAS is offered in different vehicles classes (e.g., *mid-size* or *luxury*) and multiple markets (e.g., Europe *EU*, North American Free Trade

Agreement *NAFTA*, and China *CHN*) with multiple characteristics and different behavior. Figure 1 represents the ADAS SPL evolution stages by rectangles and software variants by ellipses.

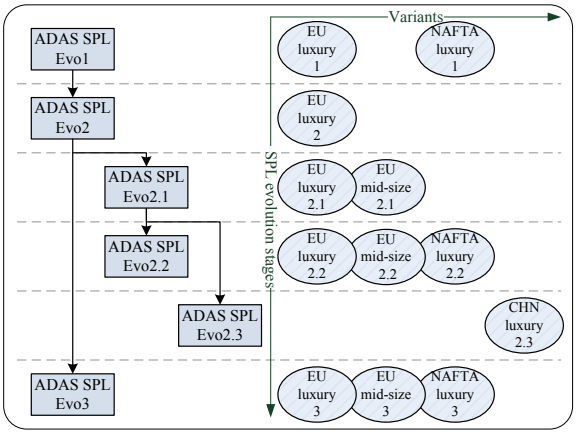


Fig. 1. The evolving Advanced Driver Assistance System (ADAS)

The *ADAS SPL Evo1* originates with the derived software variants *EU luxury 1* and *NAFTA luxury 1*. All variants are derived from a specific ADAS SPL evolution stage. A further development of already derived variants is not intended. In consequence, the development and maintenance is exclusively done at the ADAS SPL and not on already derived variants. A diversifying context of the ADAS, like emerging user requirements, results in a new *ADAS SPL Evo2*. At this point, only variant *EU luxury 2* was derived. In this regard, variant *EU luxury 1* and *EU luxury 2* can have different characteristics. Consequently, variants *EU luxury 1* and *EU luxury 2* may evolve similarly to the ADAS SPL. In the automotive domain, maintaining existing software variants is usual concerning prevalent hardware and software restrictions (e.g., software architecture or missing hardware components). Accordingly, variant *EU luxury 1* can exist in parallel with variant *EU luxury 2*. As a consequence, *ADAS SPL Evo1* has to be maintained in parallel with *ADAS SPL Evo2*. Subsequently, legislation changes for only a subset of software variants of the SPL and additional technical reasons (e.g., dependencies between systems) required a subdivision of the existing ADAS SPL in an independent development path *ADAS SPL Evo2.1*. A further separation of the development path was created due to software robustness (*ADAS SPL Evo2.2*) and additional software capability (*ADAS SPL Evo2.3*) efforts. Independently, a new *ADAS SPL Evo3* occurred. In addition, Figure 1 illustrates a variety of derived variants at different SPL evolution stages.

In the ADAS CPS, software variants are validated in long running testing procedures before start of production. For example, software variants are tested incrementally by module tests, component tests, system tests, and vehicle tests.

Meanwhile, the software development proceeds further in parallel, meaning a detected software defect during the aforementioned tests may corresponds to an earlier development stage. Consequently, the software defect has to be detected and fixed in all interim SPL evolution stages and derived variants. Additionally, derived variants cannot always be replaced by successor variants. As a result, variants of different SPL evolution stages have to be maintained in parallel. Summing up, the observed evolution patterns of the SPL are described in the following: **(1)Evolving SPL:** Legislation changes, technical reasons, or diversifying customer expectations lead to evolving SPLs in all pieces of software [9, 8, 12]. **(2)Differing derived variants:** Each evolution stage of a SPL may derive additional variants, replace existing variants, or change their characteristics. Derived variants and the SPL can evolve independently. **(3)Maintaining multiple SPL evolution stages:** Software improvements, enhanced functionality, or software defect removal for existing SPL evolution stages may lead to multiple maintenance paths with detached evolution stages (e.g., *ADAS SPL Evo2.2* in Figure 1). This is essential in the development of CPS due to existing and unchangeable hardware components (e.g., sensors, actors, processor, or memory) and associated software restrictions. A specific variant, for instance, may not receive the latest software release, if the hardware requirements are not fulfilled. **(4)Quality aspects versus enhanced capability:** Through different time-to-market schedules of differing variants, a conflict between quality aspects (e.g., software defect management, stability) and enhanced software capability (e.g., new features) may lead to a further subdivision of the SPL development.

3 Approach

In this section, we describe our approach to face the existing challenge of detecting the origin of a described software defect and its distribution in evolving SPLs in the context of variable CPSs. First, Section 3.1 characterizes preconditions in order to guarantee the functioning of our approach. Second, Section 3.2 describes tasks to ensure a later software defect removal in more detail. Third, Section 3.3 describes the workflow using the given preconditions and tasks for a software expert.

3.1 Preconditions

Precondition 1 - Use of Explicit Variant Management The SPL needs to be under explicit variant management control. Current SPLs are developed and maintained using techniques like feature oriented modelling ([13, 14, 11]), orthogonal variability models (OVM) [4], or decision-based approaches [15] to manage variability in an explicit manner. Variable product characteristics have to be described in the problem space. For example, different variation types are utilized to express optional, mandatory, or alternative variability characteristics. Constraints can be used to express require and exclude relationships between

variability characteristics. Additionally, the concrete variability realization (solution space) has to be explicitly described. Relations between the problem and solution space are assumed.

Precondition 2 - Use of Configuration Management Regardless of the used SPL approach, the described industrial application context of the SPL requires treating maintenance and evolution aspects. However, SPLs can evolve in different speed in their solution space (e.g., emerging requirements) as well as in their problem space (e.g., emerging features) [8, 16].

A unified and mature configuration management has to provide a comprehensible evolution process and specific SPL evolution stages. Each SPL evolution stage has to correlate with a particular stage of the utilized evolving variability model. Hence, the adequate management of evolving variability models is necessary. However, an evolving variability model can be considered as a usual artifact. As such, an evolving variability model is manageable in the same way as evolving architecture models, implementation artifacts, or test cases. Consequently, all artifacts representing an evolution stage need to be under control of a configuration management system. Since each evolving variability model allows dedicated derived variants, these have to be managed in a configuration management system, too. This ensures that software variants refer to their corresponding SPL evolution stage. Due to a variety of parallel evolution stages and multiple development paths of a SPL, an assignment of a specific variant to its corresponding SPL evolution stage is given by the explicit variant management. In the ADAS CPS we used *baselines* to define specific SPL evolution stages. Based on the definition of CMMI, a baseline represents "[..] a set of specifications or work products that has been formally reviewed and agreed on, which thereafter serves as the basis for further development [..]" [17].

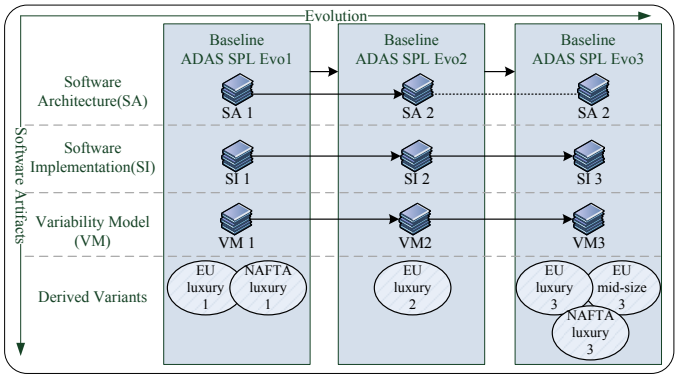


Fig. 2. Variability model in the ADAS configuration management system.

Figure 2 illustrates a part of the ADAS configuration management system with different software artifacts (vertical axis) and their individual evolution (horizontal axis). For the sake of simplicity, Figure 2 illustrates evolution without subdivided development paths. As a general remark, it should be noted that each artifact can evolve independently. For example, the software architecture does not evolve between *ADAS SPL Evo2* and *ADAS SPL Evo3*. As a result, evolution of ADAS, their corresponding variability model, and the derived variants is comprehensible in a configuration management system.

Precondition 3 - Use of Software Defect Management Emerging software defects have to be characterized in a software defect management system. At least, the software defect description has to characterize the faulty behavior, describes the expected behavior, and refers to the variant, which itself refers to the SPL evolution stage, where the software defect was detected.

3.2 Tasks to be Carried Out

Supposing the aforementioned preconditions, software experts using a variant management system and a configuration management system are able to determine the *iso*, its origin and distribution. To ensure a later reliable software defect removal in the context of an evolving SPL with multiple development paths and multiple derived variants, the following tasks have to be carried out.

Task 1: Determine the *ISO* in a Specific SPL Evolution Stage A software defect may affect the common or the variable functionality of a SPL. In more detail, a software defect of a SPL can affect an individual software variant, multiple software variants, or each software variant at a specific evolution stage. Investigating the entire SPL functionality can become complex due to the high amount of existing parts.

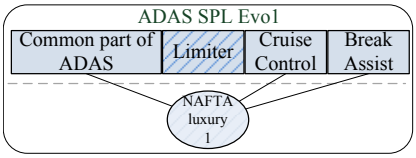


Fig. 3. *NAFTA luxury 1* functionality in contrast to provided *ADAS SPL Evo1* functionality.

As the software defect is detected in a specific software variant, software experts have to look only on those software artifacts that are relevant for that variant. Neglecting the residual SPL functionality reduces the effort and allow software experts to determine the *iso* in less time. For instance, the *Limiter*

functionality is not contained in the *NAFTA luxury 1* variant (see Figure 3) and thus need not to be considered. As a result, variant management helps software experts to determine the relevant software artifacts of the to be investigated variant. Finding which part of that set of artifacts is actually the *iso* needs a deeper and thoroughly look into the parts in combination with the defect description.

Task 2 - Determine Affected SPL Evolution Stages Determining the origin and distribution of a software defect can become complex in an evolving SPL. Multiple development paths can complicate this even more. Several development activities can be carried out between software defect introduction and software defect detection. For example, a software defect may be introduced in *ADAS SPL Evo2* (see Figure 1) but the software defect may be initially detected in *ADAS SPL Evo3*. Especially, the branching of development paths (e.g., *ADAS SPL Evo2.1*) complicates such an intention. Regarding software maintenance, the knowledge of all affected SPL evolution stages supports management decisions and is necessary in industrial projects. Stakeholders have to be informed about affected SPL evolution stages and derived affected variants. Consequently, stakeholders may dispose variant specific maintenance effort. For example, variants containing the *iso*, but currently in a deactivated state, may not be maintained until the *iso* gets activated. In contrast, variants that potentially show the faulty behavior should be maintained immediately. For such decisions, all affected variants in an evolving SPL need to be known, too (see $Task3$).

Based on the determined *iso* ($Task1$), the following algorithms determine its origin and a list of all SPL evolution stages that contains it and thus are potentially affected. We define an $evolvingSPL$ $(eSPL) = (V, E, r)$ as a directed tree with connected *edges*, *vertices*, and a defined *root* vertex. Each *vertex* $v = (p; c; A)$ represents a specific SPL evolution stage in a configuration management system. p is the parent vertex (previous SPL evolution stage) and c are [0..n] child vertices (successor SPL evolution stages) of v. A comprises development artifacts like requirements, architecture models, and variability models. Each directed *edge* $e = (s; d)$ represents a relationship and is associated with an ordered pair of $vertices(s; d)$; s is the source and d is the destination of e. r represents a defined root vertex of an $evolvingSPL$. Based on this, we call $eSPL = (V, E, r)$ an $evolvingSPL$ if:

- For every vertex $v \in V \setminus \{r\}$, there exist exactly one $p \in V$ with $e = (p, v)$. In other words, any vertex that is not the root vertex has exactly one incoming edge.
- There exists no $v \in V$ with $e = (v, r)$. In other words, the root vertex has no incoming edge.
- For every vertex $v \in V$, there exist $v_1, v_2, \ldots, v_n \in V$ with $e_1 = (r, v_1), \ldots, e_n = (v_n, v)$. In other words, for every *vertex* $v \in V$, there exists a path that starts at r and ends at v.
- As a consequence, $eSPL$ has no circles, no self loops, and no incoming edges.

Determining the software defect's origin is essential to understand its further distribution and to support stakeholder's maintenance decisions. A software defect can be introduced either in v or in an ancestor of v. Based on the determined iso and the corresponding SPL evolution stage $vDetected$, an $eSPL$ can be examined to determine the origin SPL evolution stage $vOrigin$ of the software defect. Algorithm 1 calculates that origin SPL evolution stage $vOrigin$.

Algorithm 1: Software defect origin

```
input  : vDetected ∈ V, iso ∈ A
output: vOrigin ∈ V
vOrigin = vDetected;
while (vOrigin has parent AND
vOrigin.parent contains iso) do
  | vOrigin = vOrigin.parent;
end
```

Algorithm 2: Software defect distribution

```
input  : vOrigin ∈ V, iso ∈ A
output: vAffected ⊆ V
Vector vAffected; Stack stack;
stack.push(vOrigin);
while stack is not empty do
  | Vertex v = stack.pop();
  | if v contains iso then
  |   | vAffected.add(v);
  |   | for vertex
  |   | child:v.getAllChildren() do
  |   |   | stack.push(child);
  |   | end
  | end
end
```

Algorithm 1 assumes one common $vOrigin$. Well defined and approved configuration management processes ensure such an assumption. Therefore, assuming one common $vOrigin$ is no restriction for industrial software projects. Likewise, an evolving SPL in a configuration management system may not fulfill the aforementioned demands of an $evolvingSPL$. For example, merging individual development paths, result in two incoming edges of a vertex and violates subsequently the first demand of an $evolvingSPL$. In this cases, we propose a preprocessing conversion that remove merge edges to fulfill the demands.

After determining the software defect's origin and its distribution the SPL needs to be examined in more detail. Algorithm 2 calculates a list of potentially affected SPL evolution stages. Based on the determined SPL evolution stage $vOrigin$, all children are investigated regarding the iso. This will be iteratively executed until no child contains the determined iso.

Currently, the two described algorithms work as long as the iso is identifiable in an evolving SPL. However, renaming, substitution, division, or merging activities are currently not handled, but as long as such activity information is accessible in a predefined manner, for instance in the configuration management system, the algorithms may be extended, easily. Alternatively, the algorithms can be executed iteratively with a new identified iso and a determined SPL evolution stage. In the considered ADAS, such activities are not common and are handled manually by software experts.

The result of the aforementioned algorithms can be concluded in more detail. If an *iso* is unchanged between two SPL evolution stages, these two stages are probably affected by the detected software defect. If an *iso* is changed between two SPL evolution stages, these two stages have to be investigated further by software experts. Changes (e.g., a new or reengineered implementation) of an *iso* are indicators for further investigations, because the faulty behavior could be fixed already or exists furthermore in a same or different manner.

Task 3 - Determine Affected Variants Based on the determined *iso*, an explicit variant management is beneficial to determine the derived and potentially affected variants in each SPL evolution stage. An existing set of all potentially affected SPL evolution stages (determined in *Task 2* described above) can be analyzed in more detail. Expert knowledge is required to evaluate a correct or incorrect classification of a calculated SPL evolution stage. If a derived variant of that calculated SPL evolution stage contains the *iso*, it is potentially affected. Supplementary expert knowledge is needed to evaluate the correct or incorrect classification of a determined variant. Beneficially, only relevant variants, which contains the *iso*, have to be investigated. Even though, the variation type (e.g., mandatory, optional, OR, XOR, AND) of *iso* is negligible. As a result, all affected SPL evolution stages and in turn derived affected variants are known.

3.3 Workflow

In the following, a holistic view will be illustrated and described in more detail. The aforementioned tasks are utilized in a predefined manner to determine the affected SPL evolution stages and their corresponding derived variants. This result will be used for further management decisions regarding maintenance effort.

Figure 4 illustrates our approach from the point of view of a software expert. The *Artifact layer* comprises several software artifacts which are managed by the *Tool layer*. Specified tools, like the software defect management system, comprise their related artifacts. The configuration management system comprises several evolution stages with individual software artifacts, variability models and its derived variants.

Figure 4 describes eight activities in the upper part, whereby activity four, five, and seven are realized through *algorithms*. The workflow is initiated by a *software expert* in $A1$ to analyse a specific bug report. Thereafter, a software expert configures and derives the indicated software variant utilizing the variant management system ($A2$). Within this software variant, the *iso* is determined in $A3$. Thereby, relations between the derived software variant and correlated software development artifacts permits to ignore irrelevant software artifacts ($Task1$). Nevertheless, expert knowledge is required to determine the *iso*. $A4$ expects the *iso* as an input parameter and determines the defect origin. Therefore, Algorithm 1 is executed. Next, Algorithm 2 is initiated in $A5$ to investigate the distribution of the software defect. As a result, a list of all potentially affected SPL evolution stages ($vAffected$) is created. In $A6$, the $vAffected$ is

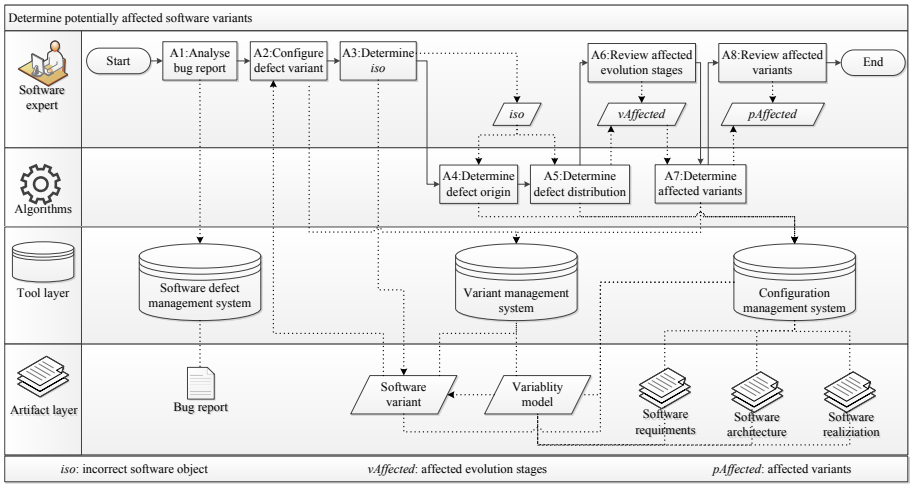


Fig. 4. Determine potentially affected software variants.

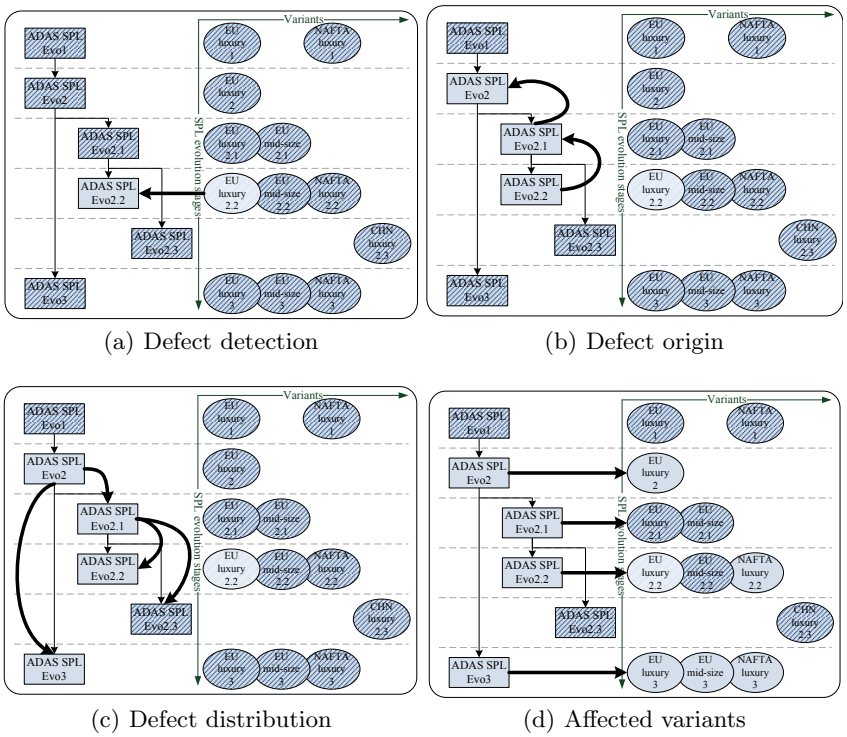
reviewed and updated by the software expert. For example, a determined SPL evolution stage will be removed from $vAffected$, if it still contains the iso but has a correct behavior or is not maintained any more. Afterwards, $A7$ creates a list of all potentially affected variants ($pAffected$). Also, $pAffected$ is reviewed and updated by the software expert to remove variants with a correct behavior ($A8$). $pAffected$ can be used for further maintenance decisions regarding software defect removal. Active or maintained variants can consequently be updated in the determined SPL evolution stages $vAffected$.

4 Application Scenario

We applied our approach to a CPS software development project in the automotive industry [10]. The application scenario refers to the ADAS CPS of Section 2 and is inspired by an industrial software defect. Furthermore, we fulfilled the aforementioned preconditions: First, the SPL is under explicit variant management control. In the ADAS CPS we use a feature model [11] to manage the variability (for more details we refer to Section 2). Second, a mature configuration management system provides a comprehensible evolution process and specifies SPL evolution stages through baselines. Third, a software defect management system comprises all emerged software defects.

Figure 5 illustrates our iterative approach to determine all potentially affected SPL evolution stages and derived variants. It shows how we use the information of the variant management system and the configuration management system in concert to expose potentially affected software variants. First, the initial state of a detected software defect is illustrated in Figure 5(a), where the software defect is initially detected in software variant *EU luxury 2.2* and subsequently in the

corresponding *ADAS SPL Evo2.2*. Through workflow activities $A2 - A3$ irrelevant software artifacts can be ignored and a software expert is able to determine the *iso* in less time than investigating the whole SPL evolution stage. Second, the software origin can be determined through Algorithm 1 in $A4$. As Figure 5(b) illustrates, the software defect originates from *ADAS SPL Evo2*. Third, activity $A5$ can be initiated in *ADAS SPL Evo2* to investigate the distribution of the software defect using Algorithm 2. Figure 5(c) illustrates the software defect distribution. The SPL evolution stages *ADAS SPL Evo2*, *ADAS SPL Evo2.1*, *ADAS SPL Evo2.2*, and *ADAS SPL Evo3* are exposed as potentially affected. Conversely, *ADAS SPL Evo2.3* is not concerned, because the *iso* was removed in this evolution stage. Fourth, all derived variants can be examined through activity $A7$. Figure 5(d) illustrates the evolving SPL and all potentially affected variants. *ADAS SPL Evo2.1* is potentially affected but no variant contains the *iso*. However, *ADAS SPL Evo2.1* may be maintained, because future derived variants will be affected. Furthermore, the *iso* became mandatory in *ADAS SPL Evo3* and all derived variants are potentially affected. Figure 5(d) illustrates the result of carrying out the workflow. All affected SPL evolution stages and derived variants are listed in a clear manner. This result, can be used for further management decisions.



(a) Defect detection (b) Defect origin

(c) Defect distribution (d) Affected variants

Fig. 5. Software defect illustration.

5 Related Work

Despite its importance, related work investigating an evolving SPL in general as well as in the context of a CPS is very limited [12]. Most of these work focus on modelling techniques or analyzes variability evolution and neglects practical application aspects. Thereby, authors address the evolution of particular types of variability models and focus on special challenges in the context of an evolving SPL [12, 18–21]. However, the integration of new techniques in running industrial projects and their existing infrastructure is not considered sufficiently. Systematic approaches that utilize existing techniques are rarely available.

In particular, Svahnberg and Bosch discusses in [9] their observations of two long running SPLs' in Swedish organizations. They recognized similar evolution scenarios as in our industrial example. Also, they described partial existing infrastructure and software dependencies. Thereby, they focus more detailed on software artifact changes, like requirements or architecture evolution and describe a set of seven guidelines to facilitate an evolving or new established SPL. Contrary, an approach to investigate an evolving SPL to determine software defects distribution is missing.

Dhungana et al. motivates in [8] the need to treat evolution of a SPL as a normal case instead of an exception. To reduce complexity, they describe a model fragment based approach. Composing several model fragments into a variability model and recording merge decisions at a given time, supports maintenance and evolution. Further, they validated their approach in an industrial model-based SPL and remark to consider engineer and tool demands. In contrast, they focus on forward evolution aspects and disregard an investigating approach.

Elsner et al. provides in [22] an overview of existing approaches that deal with product line variability and evolution aspects. They recognized a different usage of variability and evolution in literature. Therefore, Elsner et al. generalized and used three types of variability over time: Maintenance/evolution, configuration management, and product derivation. They identified tasks and aspects for a detailed evolution research agenda and applied the three types of variability on two product lines. They considered product derivation and configuration management separately. In contrast, our approach combines all types of variabilities.

Passos et al. hypothesize in [23], that changes in an evolving SPL can be handled more effectively at the level of features. They motivate the need for traces between features and their realizations. Also they motivate evolution traces for features, artifacts and relations. Following, Passos et al. listed existing work and postulated several research questions regarding traceability and evolution at the level of features. Again, an approach to investigate an evolving SPL is missing.

Seidel et al. describes in [16] a conceptual basis for the evolution of model based SPLs. The concept maintains consistency of artifacts and features regarding an evolving SPL. Further, they describe evolving feature operators (e.g., move, copy, remove, split, or merged) which may improve the given algorithms of this paper.

Heider et al. presents in [24] a meta-model for tracking model-based product line evolution. They developed EvoKing to automatically maintaining a devel-

opment history. In contrast, Heider et al. consider evolution in a technical point of view and disregard an investigating approach in case of software defects.

6 Conclusion

The work presented in this paper provides an approach to investigate software defects in an evolving CPS. Furthermore, we focus on software variability which is realized through a SPL. For this purpose, evolution observations of a SPL development projects from the automotive domain were explained. Our approach supports software experts in case of software defects in an evolving SPL. As basis, well established configuration management methods and variability modelling techniques are used. Furthermore, the approach particularly describes tasks to be performed, to detect a software defect's origin and distribution in an evolving SPL. As a result, all affected SPL evolution stages and derived variants are listed in a clear manner. This can be used for further management and maintenance decisions. An application scenario to expose potentially affected software variants was illustrated to show the relevance and usefulness of the topic in real industrial scenarios. The approach investigates the complex field of an evolving SPL and presents a holistic view for a software expert. Nevertheless, we are aware of the current limitations of the two algorithms. In this field, we will conduct further research and enhance our approach.

Evolving SPLs require further investigations to combine variability and evolution in industrial approaches. Especially, guidelines and best practices using established methods and tools have to be described in more detail. Replacement rules or maintaining information for derived variants in evolving variability models may enhance accuracy of the aforementioned approach.

Acknowledgments This work is part of the "Software Platform Embedded Systems 2020 XT" (SPES XT) project. SPES XT is supported by the German Federal Ministry for Education and Research (BMBF) by 01IS12005.

References

1. Lee, E.: Cyber physical systems: Design challenges. Technical report, Berkeley: University of California (2008)
2. Rajkumar, R., Lee, I., Sha, L., Stankovic, J.A.: Cyber-physical systems: the next computing revolution. In: Proc 47th Design Automation Conf (DAC), ACM (2010) 731–736
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-oriented Software Product Lines: Concepts and Implementation. Springer, New York Inc (2013)
4. Pohl, K., Böckle, G., Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
5. Thiel, S., Hein, A.: Modeling and using product line variability in automotive systems. IEEE Software **19**(4) (2002) 66–72

6. van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Proc 2th Working IEEE/IFIP Conf on Soft Architecture (WICSA), IEEE Computer Science (2001) 45–55
7. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proc of the IEEE **68**(9) (1980) 1060–1076
8. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. Journal of Systems and Software **83**(7) (2010) 1108–1122
9. Svahnberg, M., Bosch, J.: Evolution in software product lines: Two cases. Journal of Software Maintenance: Research and Practice **11**(6) (1999) 391–422
10. Manz, C., Stupperich, M., Reichert, M.: Towards integrated variant management in global software engineering: An experience report. In: IEEE 8th Int Conf on Global Soft Eng (ICGSE). (2013) 168–172
11. Beuche, D., Papajewski, H., Schröder-Preikschat, W.: Variability management with feature models. Science of Computer Programming **53**(3) (2004) 333–352
12. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: State of the art and perspectives. Int Journal on Software Tools for Technology Transfer (1) (2012) 1–19
13. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) feasibility study. Volume CMU/SEI-90-TR-21 of Technical report. Carnegie Mellon University. Software Engineering Institute., Pittsburgh (1990)
14. Czarnecki, K., Eisenecker, U.: Generative programming: Methods, techniques, and applications. Addison-Wesley, Harlow (2000)
15. Dhungana, D., Grünbacher, P., Rabiser, R.: Decisionking: A flexible and extensible tool for integrated variability modeling. In: Proc 1th Int Workshop on Variability Modelling of Software-Intensive Systems (VaMoS). (2007) 119–127
16. Seidl, C., Heidenreich, F., Aßmann, U.: Co-evolution of models and feature mapping in software product lines. In: 16th Int Soft Product Line Conf SPLC, ACM (2012) 76–85
17. Carnegie Mellon Software Engineering Institute: Cmmi for development: Version 1.3: Improving processes for developing better products and services. (2010)
18. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley (2000)
19. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: A case study. Journal of Systems and Software **74**(2) (2005) 173–194
20. Deng, G., Schmidt, D.C., Gokhale, A., Gray, J., Lin, Y., Lenz, G.: Evolution in model-driven software product-line architectures. Designing Software-Intensive Systems: Methods and Principles (2008)
21. Mende, T., Beckwermert, F., Koschke, R., Meier, G.: Supporting the grow-and-prune model in software product lines evolution using clone detection. In: Europ Conf on Soft Maintenance and Reengineering CSMR. (2008) 163–172
22. Elsner, C., Botterweck, G., Lohmann, D., Schröder-Preikschat, W.: Variability in time - product line variability and evolution revisited. In: Proc 4th Int Workshop on Variability Modelling of Software-Intensive Systems (VaMoS). (2010) 131–137
23. Passos, L., Czarnecki, K., Apel, S., Wasowski, A., Kästner, C., Guo, J.: Feature-oriented software evolution. In: Proc 7th Int Workshop on Variability Modelling of Software-Intensive Systems (VaMoS). (2013) 17–25
24. Heider, W., Rabiser, R., Dhungana, D., Grünbacher, P.: Tracking evolution in model-based product lines. MAPLE Software Engineering Institute, Carnegie Mellon (2009) 59–63