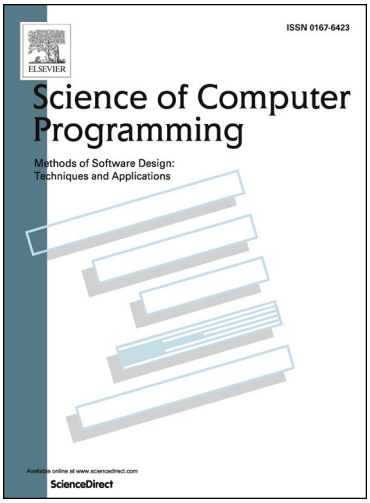
Journal Pre-proof



An industrial experience of using reference architectures for mapping features to code

Karam Ignaim, João M. Fernandes and André L. Ferreira

PII: S0167-6423(24)00010-8
DOI: https://doi.org/10.1016/j.scico.2024.103087
Reference: SCICO 103087

To appear in: *Science of Computer Programming*

Received date: 4 March 2023
Revised date: 17 January 2024
Accepted date: 18 January 2024

Please cite this article as: K. Ignaim, J.M. Fernandes and A.L. Ferreira, An industrial experience of using reference architectures for mapping features to code, *Science of Computer Programming*, 103087, doi: https://doi.org/10.1016/j.scico.2024.103087.

An industrial experience of using reference architectures for mapping features to code

Karam Ignaim[a,1], João M. Fernandes[b,2], André L. Ferreira[b,c,3]

[a]*Department of Software Engineering, Prince Abdullah bin Ghazi Faculty of Information and Communication Technology, Al-Balqa Applied University (BAU), Salt, Jordan*
[b]*Departamento de Informática, Universidade do Minho, Braga, Portugal*
[c]*Bosch Car Multimedia Portugal, Braga, Portugal*

Abstract

Context: Software Product Lines (SPLs) constitute a popular method for encouraging the methodical reuse of software artefacts. Just like any other piece of software, SPLs require management throughout their evolution, namely to preserve the consistency between requirements and the code. **Problem:** Over time, for a given SPL, many change requests are made and all of them need to be integrated in a consistent and coordinated way. The evolution of an SPL is facilitated if there exist links between its artefacts, namely between each feature and its respective pieces of implementation code. **Method:** This paper proposes FMap, a systematic feature mapping approach to be used within SPLs. FMap traces a Feature Model (FM) to other artefacts of an SPL, the reference architecture, and the code, and it establishes connections between each feature of the FM and its locations in the code-base. Additionally, we have created a tool called friendlyMapper to provide some automatic support for the approach. Using two case studies from two different companies, FMap and friendlyMapper are evaluated. **Results:** The evaluation of the case studies indicates that the FMap approach outperforms the baseline approach (i.e., the branching approach). **Contribution:** This work contributes with FMap, a novel tool-based approach that supports feature-architecture-code mappings based on reference architecture. FMap assists software engineers in adapting the evolution of the SPLs to accommodate new features and change requests as the SPLs evolve. The case studies for both companies demonstrate that the approach is applicable to real-world products and is able to support feature traceability and maintain consistency among features, architecture, and code.

Keywords: software product line, feature model, feature location, maintenance, traceability, mapping.

1. Introduction

A products family is a set of similar products that are generated from a common platform and contain unique features (functionalities) to meet specific customer requirements. Each member of a products family is referred to as a product [1]. Industry uses Software Product Lines (SPLs) to generate a family of related software products from a shared set of assets [2]. A feature model (FM) is a representation of all the products of a given SPL, in terms of features and their relationships. A feature is the fundamental building block of an FM [3, 4] and constitutes a unit of functionality that meets a set of requirements of a products family. Features describe the functional aspects, as well as the quality characteristics of the products that are part of an SPL. Each product in an SPL is characterised by a unique and valid combination of the features, as indicated by the FM. Thus, the FM serves to manage the variability within an SPL [5].

*Corresponding author
[1]karam.ignaim@bau.edu.jo; ORCiD: 0000-0001-8650-8021
[2]jmf@di.uminho.pt; ORCiD: 0000-0003-1174-1966
[3]Andre.Ferreira2@pt.bosch.com; ORCiD: 0000-0002-7028-426X

Over the past few decades, many industrial companies, like Bosch, Nokia, Philips, and Siemens, embark on an SPL Engineering (SPLE) approach [1]. They all expect to increase software customisation, to reduce the time-to-market, and to improve the quality of their products. However, there are many challenges that the software engineers face when they adopt an SPLE approach that obliges them to systematically model and manage the variations in their products. For instance, there is a need for methods to identify and exploit variabilities, appropriate modelling methods that handle thousands of variations, efficient methods to detect and resolve dependency interactions among variations at different abstraction levels, and techniques to trace variability information among development artefacts.

Feature location is an important aspect when developing an SPL. It is the process of identifying and locating the specific features within a SPL that are needed to meet the requirements of different stakeholders, such as clients, customers or users. For customisation, reusability, scalability, maintenance, evolution, and product differentiation, the location of features in SPLs is essential. It allows for effective management of a product line, enabling the creation of customised products, while maximising reuse and minimising development effort. Identifying the features in an SPL is usually tedious and error-prone, as it may involve browsing similar variants of the same products [6]. When experience fails to provide an immediate answer to locate features in the code, programmers must become more systematic in that activity.

Based on the feature location concept, we decided to develop FMap. This approach has the capacity to define and maintain traceability among artefacts involved in the software life cycle in forward and backward directions [7]. The traceability could be specified, for instance, from requirements to code (forward direction) or in the opposite direction (backward direction). A detailed explanation is presented in Section 3. In this paper, the code units that are used as a result of the mapping from a feature are called "feature-code fragments," and the relationships between each feature and its corresponding code units are called "tracing links."

In the literature, there are many works addressing feature location, such as [8–12]. Significant work on SPLE approaches, techniques, and tools was developed to manage the mapping of SPL artefacts [13–17]. They can map a feature either to a design model [18–20], to the reference architecture [13, 21], or to the code [15, 22], mostly relying on tracing links. Traceability is still a challenging issue [13, 23, 24]. Autonomously creating and maintaining tracing links across various software artefacts and development tools is a research challenge and not yet generically feasible in SPLs [14, 25]. Specifically, none of the existing approaches support feature-architecture-code mapping at the same time, as there is no architecture-implementation mapping mechanism for SPLs. Existing works in this field maintains conformance between the architecture and code-base of a single product according to a particular criterion [26]. None of them can guarantee that the variability in the reference architecture and its implementation are consistent with respect to variability conformance. Additionally, the variability of the reference architecture may be altered as a result of a change in a feature. To maintain variability conformance, none of these methods can autonomously update the code-base of an SPL, particularly programmer-created or user-defined code.

1.1. Problem statement

Implementing new product requests in a product family that was developed using an ad hoc approach (e.g., a branching, a copy-and-paste or a baseline approach) requires a substantial amount of effort. The crucial step is to determine where and how the relevant features are implemented within the code-base. The process of responding to new product requests begins with feature mapping, i.e., feature location. Our FMap approach aims to facilitate software engineers in the process of transforming existing product families into SPLs. The location and mapping of features are essential for this transition. Using a reference architecture to map features to the code-base, developers gain a clear understanding of the existing code structure and organisation. This knowledge enables them to identify the components and modules that must be modified or expanded to accommodate the new product requests. They can rapidly navigate into the code-base, locate the appropriate code sections, and make targeted changes, thereby reducing the implementation time and effort.

1.2. Contributions

This work presents FMap, a novel feature mapping approach. The work exploits and continues our previous research, where we introduced an approach to support the evolution of an SPL after its construction using the existing products family [27]. In that work, we used the requirements document of the products family as input to derive the FM that describes the SPL. Based on this contribution, this paper proposes an approach that maps features belonging

to the FM to feature-code fragments of the code, using the reference architecture (i.e., feature-related architecture elements, including subsystems and components) as a centric point. In order to fulfil the mapping process, our approach uses the traceability tree, which is an artefact that stores the tracing links between the FM and the code. A tracing link maps a given feature to the future-related-code fragments, enabling the FM and the code to be linked and to evolve consistently. This study also contributes with friendlyMapper, a tool that facilitates feature mapping. It uses the traceability tree to retain the tracing relationships between features and feature-code fragments whenever a feature is modified. Using case studies conducted at two companies (Bosch Car Multimedia and Pioneers Company), we have evaluated our approach and the associated tool. The results provide good insights on the practical applicability of the approach and the tool in industrial contexts.

The remainder of this paper is organised as follows: Section 2 introduces the relevant background on FMs and Reference Architectures. Section 3 describes the FMap approach. Next, the evaluation process is presented in Section 4. Section 5 discusses the main threats to validity, while the related works are presented in Section 6. Finally, the conclusions and future directions of the work are discussed in Section 7.

2. Background

2.1. Feature models

To adopt SPLs from the existing products family, the software engineers initially are expected to derive the FM [27–29]. Figure 1 shows an example of an FM for the "Mobile Phone" domain. The part of the tree that is at the root is called "Phone." In every valid configuration of a product, there is always only one root feature (or tree). The FM supports the "Connectivity" feature, and it can also support the "Camera" feature as an option. "Connectivity" is a feature that most phones have, and it can be either "Bluetooth" or "WI-FI."

If a feature is present in every product in a family, it is referred to as being common (or mandatory). On the other hand, an optional feature is present in certain products from a family but not all of them [18]. Additionally, features can be classified as "Or" or "or-groups" (such as "Bluetooth" and "WI-FI") or "Alternative" or "xor-groups" (i.e., alternatives relation). The model dependence relationships can be specified using terms like "requires" (selecting one feature necessitates the selection of another) and "excludes" (two features mutually exclude each other). For instance, the "Camera" needs the "Bluetooth" [3, 30, 31].

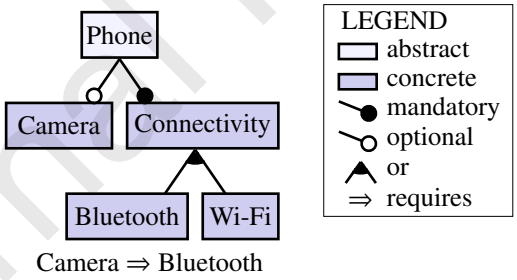


Figure (1) An FM for the "Mobile Phone" domain .

2.2. Reference architecture

A reference architecture is a fundamental architecture that captures the high-level design for SPL applications [23]. The variability of an SPL is incorporated into the framework of a reference architecture. The feature mapping approach described in this study can be used with reference architectures that follow the layered architectural style, which presents desirable characteristics for SPLs [32]. Figure 2 shows the general structure of the reference architecture, according to the development view. This is the same architecture of the final SPL that has been extracted from the products of a family in our previous work [27]. The development view is used to model the architecture structure in layers, subsystems, and components. Typically, each layer has a distinct role, function and responsibility and it includes an internal structure composed of subsystems and components.

The key purpose of decomposing the system according to the development view, which includes the high-level decomposition of the software system into subsystems, components, and their relationships, is related to the fact

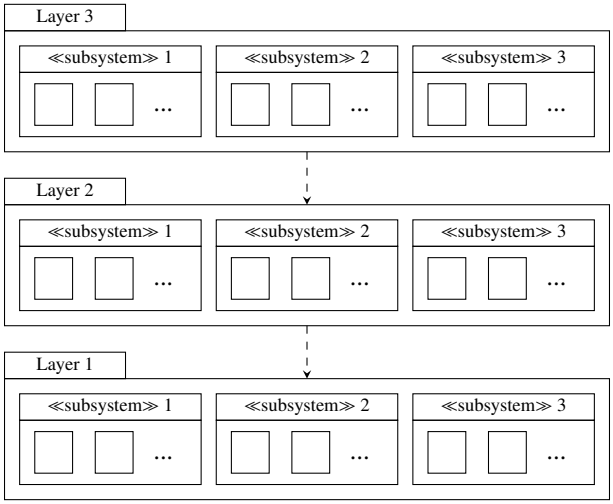


Figure (2) Reference layered architecture of the final SPL.

that it is an essential view for capturing commonality and variability. Therefore, the variable parts of the reference architecture correspond closely to the FM elements (see Figure 3).

Typically, in order to design for reuse, the architect develops subsystems such that the required variability is enclosed within the design of their internal components. This assignment is fixed for all SPL products. The structure, represented in Figure 2, consists of three layers (1–3).

Layer 1 supports the essential functionality (i.e., platform software) of the SPL as a whole. Layer 2 consists of subsystems including components for fundamental domain-specific functionality (i.e., basic software). Layer 3 contains the special requirements of a product within a product family in order to meet the needs of a particular customer [33]. It is the highest layer that documents variability among the products of an SPL; hence, only this layer is depicted in Figure 3.

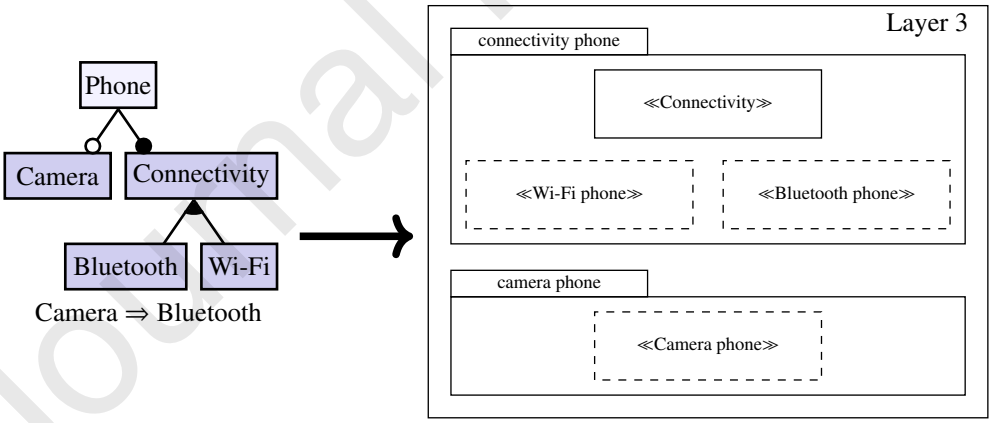


Figure (3) The FM for the Mobile Phone SPL and its mapping to layer 3 of the reference architecture.

Figure 3 represents the relationship of the FM for the Mobile Phone SPL to the layer 3 of the reference architecture. Subsystems are depicted by package (rectangle with a top-left tab) and components by a component notation (rectangle). The components with solid lines comprise the common parts of the structure. The optional alternative components are represented by dashed lines. The layer contains two subsystems: “connectivity phone” and “camera phone.” The “connectivity phone” subsystem contains three “Connectivity,” “Bluetooth phone,” and “WI-FI phone” components. The “Camera phone” component is part of the “camera phone” subsystem. The “connectivity phone” subsystem consists of the component “Connectivity” for the common “Connectivity” feature. This subsystem also

provides components for each two variations that appear in the FM as “Bluetooth” and “WI-FI.” Each one is implemented by a distinct component (“Bluetooth phone” and “WIFI phone”, respectively) within the “connectivity phone” subsystem. The optional component “Camera phone” is related to the optional “Camera” feature.

3. The feature mapping (FMap) approach

This section discusses FMap, which provides a solution for tracing features and code using the development view of the reference architecture.

3.1. General overview

FMap defines a set of tracing rules to map features of the FM to code fragments that implement those features. Our solution is based on tracing each feature to its components in the uppermost layer of the reference architecture, and then from those components to the code units that correspond to them.

As depicted in Figure 4, the FMap approach accepts as inputs the FM, the reference architecture, and the code artefacts and it generates as output a traceability tree.

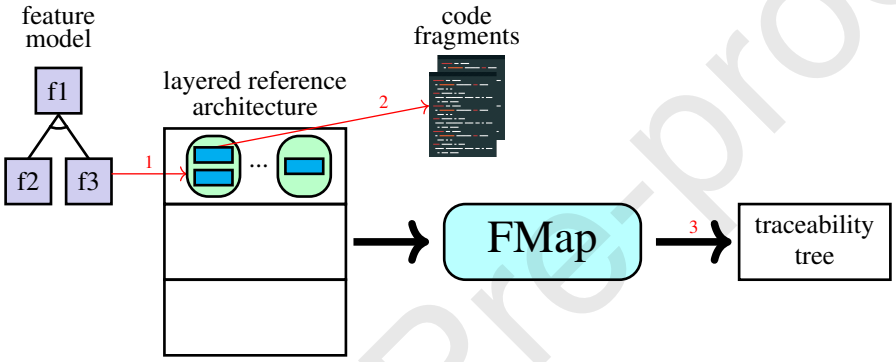


Figure (4) The proposed FMap approach. The green rounded rectangles represent components and the blue rectangles represent modules (inside the components). The red numbers indicate the three macro-steps.

The FMap approach includes three macro-steps:

1. the **feature-architecture mapping**, where each feature is traced to its corresponding parts of the uppermost layer of the reference architecture;
2. the **feature-code mapping**, in which the identified part of the reference architecture is traced to its corresponding code units; and
3. the **feature mapping conformance**, where the code pieces that implement each feature are identified and put in the traceability tree.

Notably, the last two macro-steps are automated using the friendlyMapper tool that the authors have developed to assist the approach.

The FMap approach is applicable when the software architecture of the system product is common, as expected, for the products in the same SPL. This work illustrates the architecture from a development standpoint in order to capture the similarity and diversity of the products of a given family. FMap requires that (1) each subsystem and its integral components of the reference architecture are typically implemented as a sub-package and its internal modules of implementation code. It also assumes that (2) the programmer understands the concepts of the program domain and that (3) he/she may not be aware of the location of a given features in the code. Variable components, which provide all project-specific configurations to other components (if necessary), and their associated modules, which, in our case, primarily consist of C-C++ files, represent the area that the FMap approach investigates to locate features in the implementation code. The reference architecture includes the specification of the component relationships (e.g., interfaces, dependencies, hierarchy).

FMap also assumes that the subsystems have equivalent conventional naming of packages, sub-packages, and module levels according to company standardisation and that all the architecture documentation, at different levels, can be explored and extracted.

3.2. Feature-architecture mapping macro-step

This macro-step intends to map features of the FM to the reference architecture. In particular, it links the features with the feature-related architecture elements, specifically the subsystems and components of the reference architecture. Actually, the FMap approach considers tracing a feature to the reference design to work smoothly for the reasons listed below:

1. the structure of reference architecture is predefined and agreed upon by the developers;
2. the architect designed the reference architecture to meet nearly all specifications of a product family;
3. the development includes the detailed design of the reference architecture, which is followed to write code that adheres to the architecture and incorporates the code view.

The feature-architecture mapping macro-step is comprised of steps 1–4.

Step 1: The software engineers establish a tracing link between the FM and the subsystems and components at layer 3 of the reference architecture.

Step 2: The software engineers establish the tracing links between each feature of the FM and the feature-related subsystems of the layer 3 of the reference architecture that are related to that feature.

Step 3: The software engineers provide the tracing links between each feature of the FM and the feature-related component in the subsystem that has already been identified for the feature.

Step 4: The software engineers store in a document the tracing links between each feature and its feature-related architecture elements. The feature-code mapping macro-step (Subsection 3.3) takes this document as an input.

Steps 1 and 2 define and maintain the traceability definition in a backward direction from a feature to requirements that specify this feature, by considering the detailed descriptions of each requirement statement. Then they use this traceability to realise feature-related architecture elements that document the feature, while ensuring that they satisfy the product requirements.

3.3. Feature-code mapping macro-step

The major objective of the FMap approach is satisfied by the feature-code mapping macro-step. Using the design structure of the reference architecture, it creates a sort of traceability link between each feature of the FM and its implementation code fragments. The reference architecture is investigated in this step, and its structure can be used as a guide. The feature-code mapping macro-step consists of Steps 5–8.

Step 5: The software engineers examine the code characteristics and code variable nature of the final SPL.

Step 6: The software engineers define the following tracing rules (TRs):

- **TR1.** All features of the FM can be traced directly to the package corresponding to layer 3 of the reference architecture, which comprises the implementation of a particular product requirement within the main package of one or more products.
- **TR2.** A feature can be traced to the subpackage that corresponds to the feature-related subsystem.
- **TR3.** A feature can be traced to the exact module corresponding to the feature-related component already identified for the feature within the subsystem.
- **TR4.** A feature can be traced to the source files of a module that is already identified for the feature.
- **TR5.** A feature can be traced to a set of routines, statements, and expressions associated with the source file that have already been defined for the feature.

Step 7: The software engineers examine and identify the feature positions within the code (i.e., feature location).

Step 8: The software engineers create tracing links between the feature and its code locations.

Step 9: The software engineers store the tracing links in the traceability tree.

The software engineers can repeat Steps 7 to 9 until all the features of the FM are considered. In Step 7, the software engineers perform feature location to find feature places (i.e., feature-code fragments), using the manual analysis of the code assisted by the Eclipse IDE, as follows:

1. Use Eclipse to import the packages of a product family into its local development workspace.

2. Label each feature of the FM with its related product (see "Labelling the FM" below).
3. Find the feature-code fragments throughout the code-base of the product.
4. Prepare feature-related terms, which are target words related to or with a significant relationship with features.
5. Retrieve the feature-code fragments by searching the code-base of the product for the feature-related phrases. The search procedure targets the code in the source files that corresponds to the product labeled in the FM. The search capability of Eclipse is used to find and return feature-code fragments. Using the code visualisation feature of the same IDE, the Eclipse IDE search capability presents feature-code fragments.
6. Relate the feature with feature-code fragments once a match appears, by highlighting the places in the code-base that match the feature-related terms.

Labelling the FM Labels are used to indicate the products that implement each optional/alternative feature. Figure 5 shows how the FM in Figure 1 was labelled. There are two alternatives for labelling a feature:

1. Software engineers can use the initial release product notation (i.e., "i") to label the common feature.
2. Software engineers can label a given optional/alternative feature with the reference of the products that contains that feature (e.g., "P2"). The approach assumes that a feature has the same implementation in all the products of the SPL.

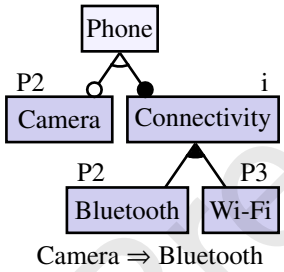


Figure (5) The FM in Figure 1 after labelling.

Traceability tree Our approach defines the traceability tree as a drop-down structure, beginning at the third level with features. It maps a feature to the routines that implement this feature by descending through each level, beginning with the SPL project, SPL package, features, and code fragments (i.e., routines). The traceability tree contains numerous tracing links. Each link connects a feature to its associated code fragments (see subwindow 1 in Figure 5). The FMap approach automates the traceability tree in the friendlyMapper tool, which enables software engineers to use the automated version of the tree to (1) trace features to the corresponding feature-code fragments and (2) retrieve the feature-code fragments for specific features.

3.4. Feature mapping conformance macro-step

The feature mapping conformance macro-step preserves the FM and code compliance whenever a feature change happens. This macro-step can semi-automatically update the tracing links of the traceability tree in order to preserve the coherence between features and code fragments. This macro-step operates on the already-established tracing links of the traceability tree.

Step 10: The software engineers execute a set of predefined activities to each case (either add or delete a feature) as follows:

- **Case 1 (add feature).** Whenever a feature change results in the addition of a new feature to the FM, the software engineers can use the tool (1) to import this feature to the traceability tree. The tool automatically (2) gives the new feature a red colour (e.g., the 4G feature shown in subwindow 1 of Figure 5), and the software engineers can use the tool (3) to create new links from the new feature to feature-code fragments.

- **Case 2 (delete feature).** Whenever a feature change results in the deletion of a feature from the FM, the software engineers can use the tool to (1) remove that feature from the traceability tree, and the tool (2) automatically eliminates all the links from this feature to feature-code fragments in the traceability tree (see subsection 3.5).

3.5. Tool support: friendlyMapper

This subsection describes friendlyMapper, which provides tool support for FMap. Specifically, it supports the feature-code mapping and the feature mapping conformance macro-steps. Using the traceability tree, the tool enables a set of actions for creating, storing, and maintaining the tracing links between features belonging to the FM and the feature-code fragments. All modifications made to the FM are reflected and saved automatically in the tree.

Figure 6 is a screenshot of the friendlyMapper tool for the "Mobile Phone" SPL example described earlier in this paper. It contains three subwindows that are related to: (1) the traceability tree; (2) the routine information; and (3) the routine list. The first subwindow (Traceability tree; '1' in Figure 6) displays the features of the FM extracted straight from the XML file (the XML file is associated with a feature tree) by selecting "Import features" from the context menu that appears when right-clicking on the "Feature" item. This subwindow also displays the tracing links between features and feature-related code fragments (e.g., from Connectivity to ConnectivityManageRoutine, IsConnectRoutine, and ConnectivityInfoRoutine).

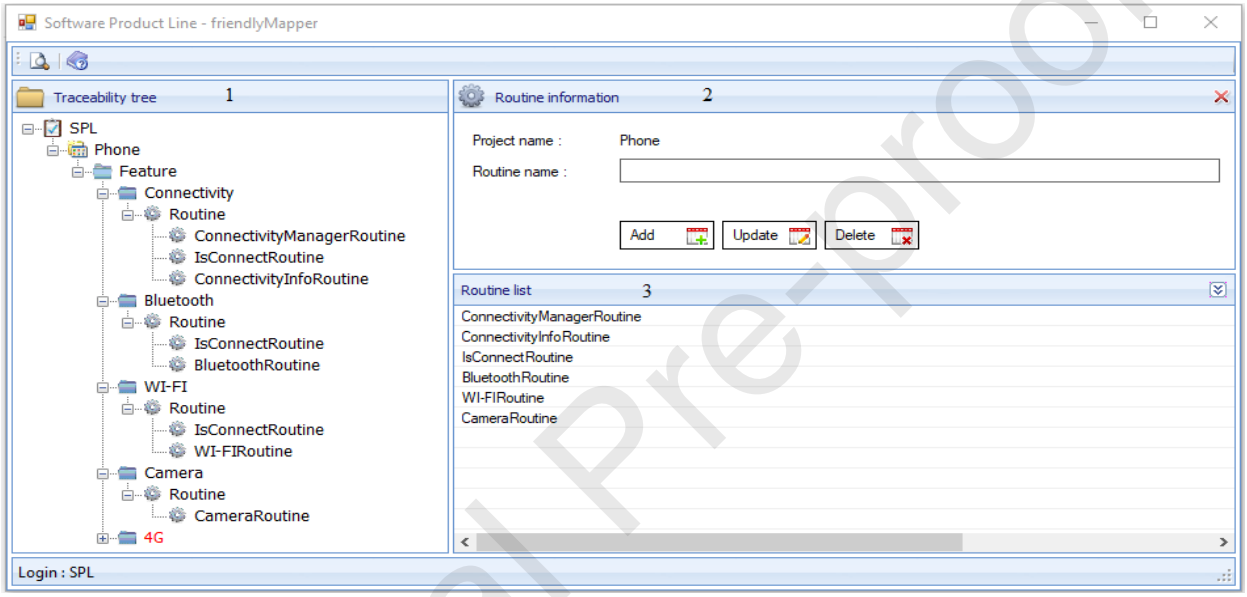


Figure (6) Screenshot for the main window of the friendlyMapper tool.

The second subwindow (Routine information; '2' in Figure 6) presents allows software engineers to enter the routine information of the SPL (e.g., code-fragments of the Mobile Phone SPL) by selecting the "Routine information" item when showing the context menu upon right-clicking on the SPL package (e.g., Phone). This subwindow allows software engineers to add, update, and delete a routine.

The third subwindow (Routine list; '3' in Figure 6) shows the list of routines in the SPL that are responsible for the code implementation of a feature.

4. Evaluation: Industrial case study

To evaluate the steps of the FMap approach and the friendlyMapper tool, we conducted two case studies at two different companies: Bosch Car Multimedia (Portugal) and Pioneers (Jordan).

4.1. Case study #1

The first case study was conducted at Bosch Car Multimedia, using the classical sensor variants family (CSVF), which follows the AUTOSAR reference architecture (AUTomotive Open Systems Architecture; www.autosar.org). The team responsible for developing CSVF is next designated as the classical sensor development team (CSDT). Initially, we employed the FM derived from a reengineering approach described in our previous work [27]. Secondly, we have established the tracing links in the traceability tree between the features and the feature-code fragments.

Thirdly, we have evolved the traceability tree by updating the tracing links to preserve coherence whenever a feature change occurs in the FM. The performance of the tool has been examined.

4.1.1. System used and participants

The CSDT at Bosch explicitly adopted an ad-hoc reuse approach (branching). The team reuses requirements documents and code from earlier projects by copying them from the artefacts of the platform (or initial release product). This is because most of the products are based on platform products [34]. The CSVF is an industrial C-based code that is widely used for designing and implementing sensors for the steering wheel braking systems of different vehicle models. The variability in the code is implemented with C pre-processor directives. For more than three years, the team has been developing and customising the CSVF to meet the needs of various customers in the automotive industry. Moreover, it has around 20 major releases and includes 50 packages. A number of features have been added and modified, while the product variants have evolved over time.

We have selected a subset of five products from the family, based on the suggestion of the project manager. The selected products cover all features of the family. Table 1 presents each product of the CSVF, to which we have assigned a different number and designated the upcoming product as 'Product New'. These products were developed according to the needs of customers in the automotive domain. The new product was scheduled to be developed upon receiving a new customer request. All the products were cloned from the platform product or the initial release product, which often evolves from the platform developed and successfully used by the first customer. They are then modified according to customer's needs.

The members of the CSDT and the project manager, who have different levels of skills and experience, participated individually in the exercise to measure the 'usefulness' of the FMap approach.

4.1.2. Evaluation objective

The objective of the evaluation is twofold: to demonstrate that (1) the steps of the FMap approach can be applied to the evolution of a realistic product family and that (2) the capabilities of the friendlyMapper tool support the traceability aspects of the SPL. We are particularly interested in whether or not both the approach and the tool are compatible with a family of products that fits the requirements and constraints of the FMap approach. Additionally, we want to assess how well the FMap approach manages the variability among the CSVF in comparison to the branching approach that the company has been using for a long time.

4.1.3. Evaluation protocol

After an analysis of the AUTOSAR architecture (Figure 7), we found that the modifications occur in the uppermost layer (PRJ). This layer corresponds to the project package of the code. This scenario makes the CSVF appropriate for our evaluation, given the objective described above. The evaluation consists of the following three phases:

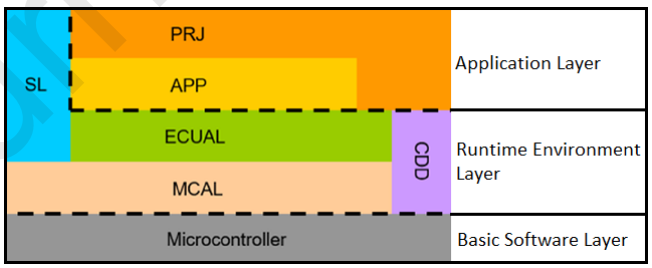


Figure (7) AUTOSAR layered architecture.

Phase 1. The initial evaluation task was to import the CSVF-related product packages, including the implementation code, into the local development workspace of the Eclipse IDE. A benefit of our method is that it does not necessitate any code modifications.

Phase 2. This phase includes the analysis of the FM,, the reference architecture, and the code, and the creation of tracing links in the traceability tree. For that, we analysed the reference architecture of the family. The CSDT employs the AUTOSAR architecture to address several forms of variation, including differences in module interfaces,

Table (1) The CSVF products.

product no.	**enabled optional features**	**package**
1	layoutT_1 layoutR_1 algorithm_1 flag_1	Product1CSVF
2	layoutT_2 layoutR_2 algorithm_2 signal_4 flag_2 signal_9	Product2CSVF
3	layoutT_2 layoutR_2 algorithm_2 flag_2 signal_9 signal_10 value_3 signal_13	Product3CSVF
4	layoutT_1 layoutR_1 algorithm_1 flag_1	Product4CSVF
New	layoutT_1 layoutR_1 algorithm_1 flag_1 signal_10 value_3	Product5CSVF

architectural levels and variations, etc. In addition, the CSDT developers readily comprehend the relationships among the variants (members). As illustrated in Figure 7, AUTOSAR comprises three layers: the Application Layer (PRJ and APP), the Runtime Environment Layer (SL, ECUAL, MCAL, and CDD), and the Basic Software Layer, which runs on a microcontroller. The application layer is a distinct component of the AUTOSAR architecture that is present in every product. Moreover, the PRJ is a specialised component of the application layer. Every single product is built with its own PRJ layer. Customer and sensor parameters are connected to the PRJ layer. Then, we mapped features of the FM to architecture elements of the PRJ layer, according to the feature-architecture mapping macro-step.

We have located the feature-code fragments that correspond to a given feature in the sub-packages, modules, and source files of the PRJ main package, including routines, statements, and expressions. The identification of features relied heavily on (i) configuration files, (ii) textual requirements, and (iii) code comments (which often contain rich documentation that helps to specify feature-code fragments). Eclipse aided in the identification of feature-code fragments through a semi-automatic examination of the code-base. We have studied the preprocessor variability of the code. Then, the Eclipse IDE was used to discover feature-code fragments based on code variability observations. For example, the C preprocessor annotations, like `#ifdef` and `#endif`, were used to annotate the code fragments and delimit the optional code associated with each feature. After identifying the features, we updated the previously generated traceability tree by either creating new tracing links or updating the existing ones. During this procedure, the software engineers could use the friendlyMapper tool to semi-automatically build and update trace links.

Phase 3. This phase employs the AUTOSAR reference architecture, the code, the FM, and the traceability tree to perform the final step of evaluation. Thus, we modified the FM designed with FeatureIDE; specifically, we made a feature modification, and we used the friendlyMapper tool to update the tracing links in the traceability tree. The tool was then run again to generate an updated version of the traceability tree.

4.1.4. Results

This section discusses the evaluation of FMap, including the friendlyMapper tool.

1. The fact that we were able to successfully complete the creation of the tracing links in the traceability tree and that all of the CSVF functionalities continued to function correctly in the finished system serves as evidence of the viability of the FMap approach. The architecture contains 56 components. There are 35 common components representing the basic software and hardware services application parts of the CSVF. As they exist in every instance, these portions are excluded from the feature tree and, consequently, the traceability tree. The application layer contains 60 features and 21 components. Inside the application layer, the PRJ layer has four components that comprise the product-specific requirements. A total of 300 feature-code relationships were created. The number of tracing links in the traceability tree was used to figure this out. On average, each feature is connected to five feature-code fragments. During the evaluation, all of them were automatically created in the traceability tree of the tool.

2. The project manager evaluated the updated CSVF for its feature-code coherence and functional correctness. Based on the tracing links in the traceability tree of the tool, the CSDT has modified the code in response to a feature change in the FM. The project manager has certified that the reference architecture, the FM, and the code continue to be consistent after modifications. We have requested that they automatically update the traceability tree after every change (e.g., adding a feature) and manually update the code to preserve its compliance with the FM. This enabled us to validate the correctness of our approach.

For instance, one feature of the CSVF states that "the sensor can write sensor commands in a message in multiple formats." We have introduced a feature to reflect the new message format (i.e., identification) and subsequently requested that the software developer of the CSDT update the code of the CSVF by getting the tracing links for the new feature from the traceability tree. In addition, "signal_4" and "interface support" features signify changes to the FM (i.e., new features); therefore, the friendlyMapper automatically displays these in red colour after re-importing the features from the XML file to the traceability tree.

3. In terms of performance, all the important activities (e.g., creation, update, and removal of tracing connections) that we have evaluated have been completed quickly (in several seconds). This is an advantage of automating the generation and maintenance of tracing links in the traceability tree.

To compare the FMap approach with the branching approach, we have defined the efficiency-dependent variable, which is calculated as the ratio between the number of correct code updates in response to feature changes and the total amount of time spent performing update scenarios. To conduct the experiment, fourteen developers of the CSDT,

Table (2) Analysis of the dependent objective variable.

	approaches	
	FMap	**branching**
mean	0.266	0.197
standard deviation	0.036	0.056

with varying levels of experience, were requested to conduct two experiments. The developers were asked (1) to add new capabilities to the FM and (2) to update the code to ensure compatibility with the new requirements.

When the project manager evaluated our approach and the friendlyMapper tool, there were several criteria to consider. These criteria help assess the effectiveness, efficiency, and overall quality of the approach. Here are eight key criteria for evaluating the approach:

1. **Accuracy:** The generated feature-code fragments should accurately reflect the desired functionality and behaviour specified by the feature. It should produce a correct and valid implementation that meets the requirements.
2. **Completeness:** The generated feature-code fragments should cover all aspects of the feature and handle various scenarios and use cases. It should not omit critical functionality or produce incomplete code-fragments.
3. **Maintainability:** The generated feature-code fragments should be readable, well-organised, and easy to understand by software engineers.
4. **Performance:** The generated feature-code fragments should be efficient and performant, avoiding unnecessary computations or resource usage. It should strive to generate optimised code that meets performance requirements.
5. **Extensibility:** The generated feature-code fragments should be designed to accommodate future changes and enhancements. It should be modular and allow for easy integration with existing code-bases. Adding new features or modifying existing ones should not require significant refactoring.
6. **User experience:** The tool should provide a positive user experience by meeting the desired user interface and interaction requirements. It should consider usability, accessibility, and responsiveness.
7. **Development time:** The approach should provide a time-efficient way to generate feature-code fragments compared to manual development. It should reduce development efforts and accelerate the implementation process.
8. **Learning curve:** The approach and the friendlyMapper tool should have a manageable learning curve for developers to adopt and utilise effectively. It should provide adequate documentation, tutorials, and support resources.

When evaluating a feature-to-code approach, it is essential to consider these criteria in the context of the specific project requirements, the capabilities of the development team, and the intended use of the generated code. To evaluate FMap, we asked every participant to adopt the branching approach to complete the tasks. Then, we asked them to use the FMap approach to accomplish the same tasks. At the conclusion of the two trials, the performance regarding the efficiency of each participant was calculated. To determine whether the *mean* difference of the dependent variable (i.e., efficiency) is considered to be statistically significant in participants performance between the two approaches, a *paired t-test* was used with the following hypotheses:

- **Ht0**: The performance of a software developer has not changed after applying FMap.
- **Ht1**: The performance of a software developer has changed after applying FMap.

Table 2 depicts the result of the *paired t-test*, which compares the average for "efficiency" of software developers using the branching approach versus the FMap approach. The output gives valuable descriptive data for the two evaluated approaches, such as the *mean* and *standard deviation*, as well as the actual findings from the *paired t-test*.

Examining the *mean* reveals that, at the end of the trial, the individuals who used our approach were more efficient than those who used the branching approach. There is a *mean* difference of 0.069 (0.266 − 0.197) between the

two trials, where df = 14. The df variable (degrees of freedom) is related to the sample size and shows how many "free" data points are available in a given test for making comparisons. The *t-value* is higher than the *t-critical value* (5.4008 > 2.2621), while the *p-value* is 0.0004. This difference is considered to be extremely statistically significant by conventional standards. As the *t-value* is higher than the *t-critical value* and *p-value* is less than the threshold (i.e., $p < 0.05$), the null hypothesis is rejected. This resulted in the final rejection of the null hypothesis **Ht0** and acceptance of the alternative hypothesis **Ht1**. In other words, after using our FMap approach, the performance of software engineers has improved. In addition, the result reveals that the *standard deviation* of FMap is less than the branching approach one (0.036 < 0.056). This indicates that there may be a greater disparity between data values for the branching approach, but the data values for our approach are close together. We may infer that our approach enables software engineers (1) to achieve convergent performance regardless of their skills, and (2) to allow the SPL to evolve smoothly without major difficulties.

An experiment with seven CSDT members, while they were trying to locate the "signal_13" feature, was conducted to compare the branching approach (originally used by the company) with FMap in terms of efficiency. Figure 8 depicts the results of that experiment. The x-axis identifies the team member (1 to 7) and the y-axis indicates the number of unsuccessful attempts to find the location of "signal_13" in the code. We have decided to follow the team members while they attempted to locate this feature. We have observed that they identify feature-code fragments of "signal_13" in a matter of seconds using our FMap approach, whereas the same members take longer to identify feature-code fragments of "signal_13" using the branching approach. Using the FMap approach as opposed to the branching approach, the CSDT has executed fewer unsuccessful attempts. With FMap, the team has completed the tasks for the empirical case study more efficiently than with the branching approach.

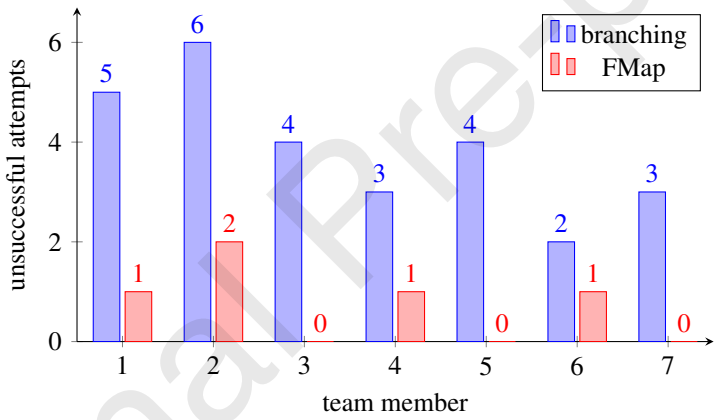


Figure (8) A comparison between the branching and the FMap approaches based on the number of unsuccessful attempts to locate the "signal_13" feature in the code for changing it.

The project manager and the CSDT members confirm that, overall, FMap benefits the company by providing consistency, scalability, re-usability, maintainability, knowledge sharing, and fostering agility and innovation. It establishes a solid foundation for building new products from the CSVF and enables efficient development and maintenance processes, leading to improved quality and a faster time to market.

4.2. Case study #2

The second case study was conducted at Pioneers, a leading software development company that focused on improving the efficiency and quality of their software development projects.

4.2.1. System used and participants

This case study involves conducting a comprehensive analysis of how change requests are managed and implemented in a variety of software products (projects), like bank information systems, COVID-19 management systems, math kids apps, and health care systems. The participants in the evaluation are software engineers who typically work in roles where they are responsible for writing, testing, and maintaining code for software applications. They have

Table (3) The efficiency values (in lines/minute) for case study #2 with the original approach (OA) and the FMap. Values in bold denote a better efficiency.

CR	change request	LOCs	efficiency	
			OA	FMap
#1	update the login page to include an additional field to enter a verification code (Bank Information System)	150	1.25	**1.50**
#2	improve the accuracy and speed of contact tracing by implementing advanced algorithms and data analytics techniques (COVID-19 Management System)	200	0.67	**0.95**
#3	introduce a new category of problem sets that focus on advanced math concepts suitable for children of higher grade levels (Math Kids Education App)	300	1.66	**2.00**
#4	integrate the healthcare system with diagnostics imaging and laboratory systems (Health Care System)	500	0.83	**1.61**

experience with various programming languages, development frameworks, and tools commonly used in the industry. Their expertise extends to both front-end and back-end development, giving them a holistic understanding of the software development process.

4.2.2. Evaluation objective

The primary objective of case study #2 is to assess and compare the effectiveness, efficiency, and impact of the steps of the FMap approach and the friendlyMapper tool. We aim to demonstrate that the tool supports traceability in different software products, spanning various domains and industries.

4.2.3. Evaluation protocol

We asked the company to adopt our FMap approach while responding to customer change requests to improve the system (the so-called perfective/adaptative maintenance). The software engineers want to modify the code-base based on the specifications of the change requests solicited by customers (namely adding new features). The main objective of the study is to determine the impact of FMap on responding to change requests, when improving a given system. For that, we formulated the following hypotheses:

- H_0: FMap has **no impact** on responding to change requests when improving a given system.
- H_1: FMap has **an impact** on responding to change requests when improving a given system.

Software engineers at the company used FMap to modify the code-base upon receiving change requests (Table 3, Column 'change request'). The company already has an estimation of the time it takes to respond to change requests using their original approach, which has been adopted for a long time (Table 3, column OA). The evaluation of the efficiency for modifications to a specific change request in the code-base is calculated based on the LOCs (abbreviation for Lines of Code) and the total time it takes to address the change request:

$$\text{Efficiency} = \frac{\text{LOCs}}{\text{total-time}}$$

The software engineers apply the evaluation activities as follows: (1) they choose a representative FM from a real-world change request at Pioneers Company (e.g., CR#1 of Table 3). The FM should have a sufficient number of features. Then (2) they apply the FMap approach that maps features of the FM to code (tracing links) based on the reference architecture. Finally, (3) they generate feature-code fragments automatically using FMap implemented for the selected FM. The software engineers could use these links as indicators of where and how the relevant modifications are implemented in the code-base. The evaluation of the efficiency for modifications with the FMap approach is shown in Column FMap; Table 3.

Table (4) The feedback from eight software engineers. A Likert scale from 1 (equivalent to "strongly agree") and 7 ("strongly disagree") is used.

role	response
Sw. Eng. 1	2
Sw. Eng. 2	1
Sw. Eng. 3	3
Sw. Eng. 4	2
Sw. Eng. 5	1
Sw. Eng. 6	3
Sw. Eng. 7	1
Sw. Eng. 8	2
Proj. Mngr	1

Once the software engineers fulfilled the implementation of the change requests, metrics could be defined and collected to assess the efficiency of the FMap approach. Since the evaluation process was conducted by comparing FMap against the original one, analysing the metrics, and collecting feedback from software engineers involved in the case study (see Table 4), the software engineers were asked the question, "Do software engineers agree on adopting the FMap approach when improving a given system?" The evaluation results presented in Tables 3 and 4 help researchers identify the strengths, weaknesses, and potential improvements of the FMap approach compared to the original one. The evaluation outcomes prove the following capabilities of FMap to address enhancements on maintenance at Pioneers Company:

- FMap demonstrates a high level of accuracy in mapping features of the feature model to code-base.
- FMap significantly reduces effort in manual modifications to the code-base, leading to increased efficiency.
- Code generated through the approach exhibits a high level of consistency, resulting in improved maintainability.
- FMap enhances traceability between the features and the code-base, enabling a better understanding of the change implementation.

The case study provides valuable insights into the effectiveness and efficiency of the FMap approach. The findings help Pioneers Company make informed decisions about adopting and refining the approach in its software maintenance processes, potentially leading to improved productivity, quality, and traceability. In conclusion, the results of the case study (presented in Tables 2 and 3) reject the null hypothesis (H_0) and prove the alternative one (H_1).

In conclusion, the evaluation shows that the FMap approach and the associated tool can offer clear traceability, enhance maintainability, promote collaboration, aid impact analysis, and align with architectural design. However, it requires initial setup effort, ongoing maintenance, and effective complexity management. With the right approach and mitigation of potential limitations, this method can significantly improve software development and maintenance processes. It is worth noting that, like any approach or tool, there are likely to be limitations. Some potential limitations of FMap and friendlyMapper may include:

1. Accuracy: The effectiveness of the pair approach+tool heavily relies on the accuracy of the software architecture representation and the mapping process itself. If the architecture representation is incomplete or inaccurate, or if there are errors during the mapping process, it may lead to incorrect feature-to-code mappings.

2. Scalability: The scalability of the pair approach+tool is an important consideration. If the code-base is very large or complex, the feature mapping process might become more challenging and time-consuming. It may be difficult to maintain an up-to-date mapping as the code-base evolves and changes over time.

3. Dependencies and Interactions: Software features often interact with each other and have dependencies on various components or modules within the code-base. Capturing and representing these complex dependencies accurately in the the FMap approach could be a challenge, potentially leading to incomplete or incorrect mappings.

4. Maintenance Overhead: Implementing the pair approach+tool may necessitate additional work and maintenance. As new features are added or modified, software engineers would have to devote time and resources to create and update mappings. This overhead should be evaluated attentively in terms of its effect on development efficiency.

5. Learnability: Adopting the pair approach+tool may require software engineers to learn new concepts and techniques. If the learning curve is steep or the tool usability is low, it may hinder adoption and effectiveness in practice.

These limitations should be taken into account when considering the adoption and practical implementation of the proposed FMap approach and the friendlyMapper tool. Further research and evaluation are necessary to assess the effectiveness and address any potential challenges or limitations.

5. Validity threats

There are some threats to the validity of the evaluation of FMap, as presented below:

- **External validity** refers to the extent to which the findings of the case studies presented in this work can be generalised to other product families, SPL domains, development teams, and enterprises. We believe that the empirical study needs to be conducted on ideal candidate FMs, which include more features. For example, a product family with more products that have a high variation However, we believe that the evaluation requires a repository containing product families from different sectors of the automotive domain, including mechanical engineering, production engineering, electrical engineering, electronics engineering, and computer engineering.

- In both case studies, the **data collection methods** are also a source of threat to validity. During data collection, we have relied mostly on a single individual (i.e., the project manager) to analyse and review the validity of our approach and record its artefacts. In addition, fourteen software developers have participated in case study #1 and eight software engineers in case study #2. Obviously, while expanding this work, we should incorporate more than one product family and more software developers with different degrees of experience.

- Regarding the **internal validity**, for the estimation of the efficacy of the FMap and the comparison with the branching approach, the software developers first completed the tasks with the branching approach and then with FMap. Repetition of a task may affect the time necessary to complete it. Additionally, the members of CSDT may embrace the same perspectives regarding FMap, and we may lose the ability to provide correct feedback if this occurs. This risk was addressed by dividing the interviews into two sessions and doing each individually. The first session was done and documented at the beginning of the month, followed by the second session at the end of the month. This may still be an issue in terms of familiarisation, since a period of 4/5 weeks may be considered as short.

6. Related work

Several approaches and techniques for SPLE have been proposed in the literature. However, few studies are focused on feature traceability, feature variability, and reference architecture, namely in the automotive domain. Heidenreich et al. [35] developed a tool named FeatureMapper. The proposed tool defines the mapping of features to model elements in Ecore-based languages. The tool supports different visualisation techniques, such as filtering and colouring, that help developers understand the complex design of SPLs. In addition, it supports automatic and manual feature mapping.

A visualisation-based technique named Colored IDE (CIDE) was proposed by Feigenspan et al. [36] to automatically visualise or hide code fragments. The proposed tool increases the understandability of the code-base and maintainability by providing different features of code views and visualising them in various colours. CIDE uses the information stored in a traceability reference to trace the feature to the code-base, and the tracing links should be updated when modifying the code-base. At the same time, the study proposed by Berger et al. [37] demonstrates the feasibility of extracting feature-to-code mappings from the build systems of two well-known operating systems, Linux and FreeBSD. The mapping process is demonstrated as a group of presence conditions relating code parts to features. The work assists in the analysis of the real-world variability case study.

Couto et al. [38] exploited the conditional compilation of C preprocessor annotations to extract a set of complex and relevant features from the code-base of a real system called ArgoUML. The feature extraction process delivered an SPL called ArgoUML-SPL. The study proposes a framework that evaluated and characterised the preprocessor-based

product lines. New metrics concerning tangling and scattering have been added to support feature characterisation based on various aspects, such as code-base size, granularity, location, and cross-cutting behaviour.

A technique that considers the granularity level of classes and methods to support the traceability information in product variants was introduced by Linsbauer et al. [39]. That technique identifies traceability based on matching code-base overlaps and feature overlaps. The individual features are the major aspect that identifies the precision of the technique. The proposed approach was evaluated in real-world software variants.

Eyal et al. [40] presented an approach to identifying the tracing links between features and feature-related code artefacts using information retrieval on a real system, ArgoUML. It was assumed that they have documented features available to use as search queries. In addition, they do not take into account the feature changes during an SPL evolution.

Zheng et al. [14] proposed an automatic approach that maps product features and their evolutionary modifications to product line architecture and code-base. The approach uses modelling and implementation techniques to address feature traceability and variability in conformance between SPL features, architecture, and code concurrently. To support the approach, a new tool named xLineMapper has been developed. A case study on the Apache Solr system is used for evaluation.

A novel feature location technique was presented by Buzaid and Hammad [17]. The strategy is designed based on employing the execution trace to map the code-base to the related functional features. The relevant feature obtained from the execution trace is queried using natural language. The suggested technique was evaluated in a case study of the ArgoUML system. Recently, a machine learning-based approach has been introduced by Airlangga et al. [12] to automatically map code-base artefacts to architectural modules. The dependency information related to syntactic and semantics obtained from the code-base and architecture descriptions is used to train the learning algorithm. A set of experiments has been conducted using eight systems well-known in this context. The results show machine learning techniques can be exploited efficiently in mapping code-base to architectural modules.

The work on SPLs is not enough to support mapping between feature architecture and code. This is because: (1) it is hard to manage the relationship and conformance between SPL artefacts (like features and the reference architecture) that can change on their own, and (2) SPLs do not have a way to map architecture to implementation. The work presented by Zheng et al. [14] provides an efficient solution to tackle the challenges presented above. The authors presented an approach and a tool that focus on mapping feature changes to the architecture and finally to code-base but not the reverse. In this case, the programmer may still introduce inconsistencies by manually altering the code-base. A programmer may, for instance, eliminate an existing annotation from a user-defined code by accident. In addition, the implementation of a component may incorrectly reference another component that is not connected to it in the architecture. The FMap approach provides a novel architecture-based approach that depends on the development view of reference architecture to support forward and backward features-architecture-code mapping, which provides a solution for the limitations of the approach presented by Zheng et al. [14] and the research challenges of this area.

It is relevant to indicate also works that address the transformation in the opposite direction, i.e., from code to architecture. A remodularisation approach, which improves the representation of features in the source code of Java programs, was presented by Olszak and Nørregaard Jørgensen [41]. Their approach supports forward and reverse restructurings through on-demand bidirectional restructuring between feature-oriented and object-oriented decompositions. The approach includes three phases: (1) to locate features, based on tracing of program execution; (2) to represent features by reallocating classes into a new package structure, and (3) to re-establish the original structure of a program from its re-modularised version.

InMap [42] is an interactive code-to-architecture mapping recommendation approach, using minimal architecture documentation to apply natural language techniques to a software code-base. InMap maps to architectural modules, low-level source code units like classes. InMap incorporates a hierarchical package mapping technique that provides recommendations for packages. It utilises InMap information retrieval capabilities to recommend mappings between a given software package and its architectural modules.

When compared to other similar approaches, namely those previously presented, the FMap approach presents several strengths. Here are a few potential benefits:

1. Clarity and organisation: By using the software architecture as a central point for feature mapping, FMap can establish a clear and organised structure for the code-base. This can help developers understand how different features relate to each other and how they fit within the overall system.

2. Modularity and reusability: Mapping features to the code-base using the software architecture promotes modularity and reusability, which helps to identify common components or modules that are shared across multiple features, making it easier to reuse the code-base and reduce duplication. This can lead to improved maintainability and faster development cycles.

3. Scalability and extensibility: Since The FMap approach is centred around the software architecture, this can support scalability and extensibility. As the system grows and evolves, the software engineers can add new features by extending existing modules or introducing new ones that align with the architecture design principles. This flexibility allows software engineers to adapt to changing requirements without significantly impacting the entire code-base.

4. Collaboration and communication: Using the software architecture as a central point for feature mapping improves collaboration and communication among team members. It provides a shared understanding of the system structure and facilitates discussions on how to implement and integrate different features. This can lead to better coordination and alignment within the development team.

5. Maintenance and troubleshooting: When features are mapped to code-base using the software architecture, maintenance and troubleshooting become more manageable. If an issue arises, developers can identify the relevant components quickly and understand their dependencies. This targeted approach simplifies debugging, testing, and maintenance tasks.

It is important to note that the success of the FMap approach depends on having a well-defined and well-communicated software architecture. Without a clear architectural design, mapping features to code may become challenging and less effective. Additionally, regular reviews and updates to the architecture will be necessary to ensure it remains aligned with the evolving needs of the system.

7. Conclusions

This paper presents FMap, a feature mapping approach that maps features to a code-base using software architecture as a centric point in order to reflect the implementation of a specific feature where it is located in the code-base. We enriched the FMap approach with friendlyMapper, a tool that (1) semi-automatically relates each feature in the FM to feature-code fragments in the code-base and (2) updates their conformance whenever a feature change occurred. FMap was evaluated at two different companies. The first one operates in the automotive industry, using an industrial-sized products family, and the second one supports different software development domains. The result reveals that FMap is applicable to dealing with real product families. In the future, we plan to improve the friendlyMapper tool to maintain software architecture, code, and FM consistency whenever feature changes occur. We also plan to extend the FMap approach to consider non-functional requirements like safety, security, and real-time constraints.

Acknowledgment

We would like to thank the classical sensor development team at Bosch Car Multimedia S.A. and particularly the project manager, Jana Seidl, for her help and support. We would also like to thank the software engineers at Pioneers Company. This work has been supported by FCT—Fundação para a Ciência e Tecnologia within the R&D Units Project Scope UIDB/00319/2020.

Data availability: Several documents and data that support this study are available at github.com/karamignaim/FMap.

Authors' contributions: Karam performed conceptualisation, methodology, software, validation, formal analysis, investigation, data curation, writing - original draft, visualisation. João was responsible for conceptualisation, methodology, writing - original draft, writing - review and editing, and visualisation. André performed resources, writing - review and editing.

References

[1] van der Linden, Frank, K. Schmid, E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer, 2007. DOI 10.1007/978-3-540-71437-8.

[2] V. Alves, N. Niu, C. Alves, G. Valença, Requirements engineering for software product lines: A systematic literature review, Information and Software Technology 52 (2010) 806–820. DOI 10.1016/j.infsof.2010.03.014.

[3] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, P. Heymans, Feature model extraction from large collections of informal product descriptions, in: 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), 2013, pp. 290–300. DOI 10.1145/2491411.2491455.

[4] S. Apel, D. Batory, C. Kstner, G. Saake, Feature-oriented software product lines: Concepts and implementation, Springer, 2013. DOI 10.1007/978-3-642-37521-7.

[5] J. Mortara, X. Tërnava, P. Collet, Mapping features to automatically identified object-oriented variability implementations: The case of ArgoUML-SPL, in: 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2020), 2020. DOI 10.1145/3377024.3377037.

[6] T. Ziadi, L. Frias, M. A. A. da Silva, M. Ziane, Feature identification from the source code of product variants, in: 16th European Conference on Software Maintenance and Reengineering (CSMR 2012), 2012, pp. 417–422. DOI 10.1109/CSMR.2012.52.

[7] O. Gotel, C. Finkelstein, An analysis of the requirements traceability problem, in: IEEE International Conference on Requirements Engineering (RE 1994), 1994, pp. 94–101. DOI 10.1109/ICRE.1994.292398.

[8] A.-D. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. Eyal-Salman, Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing, in: 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013), 2013, pp. 244–249.

[9] W. K. G. Assunção, S. R. Vergilio, Feature location for software product line migration: A mapping study, in: 18th International Software Product Line Conference (SPLC 2014), Volume 2, 2014, pp. 52–59. DOI 10.1145/2647908.2655967.

[10] R. AL-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. E. Salman, Feature location in a collection of software product variants using formal concept analysis, in: J. Favaro, M. Morisio (Eds.), 13th International Conference on Software Reuse (ICSR 2013), volume 7925 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 302–307. DOI 10.1007/978-3-642-38977-1_22.

[11] Y. Xue, Z. Xing, S. Jarzabek, Feature location in a collection of product variants, in: 19th Working Conference on Reverse Engineering (WCRE 2012), 2012, pp. 145–154. DOI 10.1109/WCRE.2012.24.

[12] T. Olsson, M. Ericsson, A. Wingkvist, To automatically map source code entities to architectural modules with naive bayes, Journal of Systems and Software 183 (2022) 111095. DOI 10.1016/j.jss.2021.111095.

[13] J. Díaz, J. Pérez, J. Garbajosa, A model for tracing variability from features to product-line architectures: a case study in smart grids, Requirements Engineering 20 (2015) 323–343. DOI 10.1007/s00766-014-0203-1.

[14] Y. Zheng, C. Cu, H. U. Asuncion, Mapping features to source code through product line architecture: Traceability and conformance, in: IEEE International Conference on Software Architecture (ICSA 2017), 2017, pp. 225–234. DOI 10.1109/ICSA.2017.13.

[15] R. Al-Msie'deen, Reverse engineering feature models from software variants to build software product lines: REVPLINE approach, Ph.D. thesis, Université de Montpellier II, France (2014). URL.

[16] Y. Hakimi, R. Baghdadi, Y. Challal, Deep learning and classical machine learning for code mapping in heterogeneous platforms, in: 5th International Conference on Networking and Advanced Systems (ICNAS 2021), 2021, pp. 1–6. DOI 10.1109/ICNAS53565.2021.9628950.

[17] F. Buzaid, M. Hammad, Mapping features to source code based on execution trace, in: 4th Smart Cities Symposium (SCS 2021), 2021, pp. 72–77. DOI 10.1049/icp.2022.0316.

[18] A. Bragança, R. J. Machado, Automating mappings between use case diagrams and feature models for software product lines, in: 11th International Software Product Line Conference (SPLC 2007), 2007, pp. 3–12. DOI 10.1109/SPLINE.2007.17.

[19] H. Gomaa, Designing software product lines with uml 2.0: From use cases to pattern-based software architectures, in: M. Morisio (Ed.), 9th International Conference on Software Reuse (ICSR 2006), Springer, 2006, pp. 440–440. DOI 10.1007/11763864_45.

[20] M. Eriksson, J. Börstler, K. Borg, The PLUSS approach–domain modeling with features, use cases and use case realizations, in: 9th International Software Product Lines Conference (SPLC 2005), volume 3714 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 33–44. DOI 10.1007/11554844_5.

[21] M. Janota, G. Botterweck, Formal approach to integrating feature and architecture models, in: 11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008), volume 4961 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 31–45. DOI 10.1007/978-3-540-78743-3_3.

[22] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. Le Traon, Bottom-up adoption of software product lines: A generic and extensible approach, in: 19th International Conference on Software Product Line (SPLC 2015), 2015, pp. 101–110. DOI 10.1145/2791060.2791086.

[23] K. Pohl, G. Böckle, F. J. van Der Linden, Software product line engineering: Foundations, principles and techniques, Springer, 2005. DOI 10.1007/3-540-28901-1.

[24] H. Abukwaik, A. Burger, B. K. Andam, T. Berger, Semi-automated feature traceability with embedded annotations, in: 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018), 2018, pp. 529–533. DOI 10.1109/ICSME.2018.00049.

[25] F. Buzaid, M. Hammad, Mapping features to source code based on execution trace, in: 4th Smart Cities Symposium (SCS 2021), volume 2021, 2021, pp. 72–77.

[26] Y. Zheng, R. N. Taylor, Enhancing architecture-implementation conformance with change management and support for behavioral mapping, in: 34th International Conference on Software Engineering (ICSE 2012), 2012, pp. 628–638. DOI 10.1109/ICSE.2012.6227154.

[27] K. Ignaim, J. M. Fernandes, An industrial case study for adopting software product lines in automotive industry: An evolution-based approach for software product lines (EVOA-SPL), in: 23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B, 2019, pp. 183–190. DOI 10.1145/3307630.3342409.

[28] J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, Y. Le Traon, Feature location benchmark for software families using Eclipse community releases, in: 15th International Conference on Software Reuse (ICSR 2016), volume 9679 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 267–283. DOI 10.1007/978-3-319-35122-3_18.

[29] F. Wanderley, D. S. da Silveira, J. Araujo, M. Lencastre, Generating feature model from creative requirements using model driven design, in: 16th International Software Product Line Conference (SPLC 2012), Volume 2, 2012, pp. 18–25. DOI 10.1145/2364412.2364416.

[30] M. Huchard, A.-D. Seriai, C. Urtado, S. Vauttier, et al., Reverse engineering feature models from software configurations using formal

concept analysis, in: 11th International Conference on Concept Lattices and Their Applications (CLA 2014), volume 1252 of *CEUR Workshop Proceedings*, 2014, pp. 95–106. URL.

[31] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, Reverse engineering feature models, in: 33rd International Conference on Software Engineering (ICSE 2011), 2011, pp. 461–470. DOI 10.1145/1985793.1985856.

[32] H. Gomaa, Software modeling and design: UML, use cases, patterns, and software architectures, Cambridge University Press, 2011.

[33] M. Broy, M. Gleirscher, S. Merenda, D. Wild, P. Kluge, W. Krenzer, Toward a holistic and standardized automotive architecture description, IEEE Computer 42 (2009) 98–101. DOI 10.1109/MC.2009.413.

[34] S. Sepúlveda, A. Cravero, C. Cachero, Requirements modeling languages for software product lines: A systematic literature review, Information and Software Technology 69 (2016) 16–36. DOI 10.1016/j.infsof.2015.08.007.

[35] F. Heidenreich, J. Kopcsek, C. Wende, FeatureMapper: Mapping features to models, in: 30th International Conference on Software Engineering (ICSE 2008), Companion Volume, 2008, pp. 943–944. DOI 10.1145/1370175.1370199.

[36] J. Feigenspan, C. Kästner, M. Frisch, R. Dachselt, S. Apel, Visual support for understanding product lines, in: 18th International Conference on Program Comprehension (ICPC 2010), 2010, pp. 34–35. DOI 10.1109/ICPC.2010.15.

[37] T. Berger, S. She, R. Lotufo, K. Czarnecki, A. Wasowski, Feature-to-code mapping in two large product lines, in: 14th International Software Product Line Conference (SPLC 2010), volume 6287 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 498–499. DOI 10.1007/978-3-642-15579-6_48.

[38] M. V. Couto, M. T. Valente, E. Figueiredo, Extracting software product lines: A case study using conditional compilation, in: 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 191–200. DOI 10.1109/CSMR.2011.25.

[39] L. Linsbauer, E. R. Lopez-Herrejon, A. Egyed, Recovering traceability between features and code in product variants, in: 17th International Software Product Line Conference (SPLC 2013), 2013, pp. 131–140. DOI 10.1145/2491627.2491630.

[40] H. Eyal-Salman, A.-D. Seriai, C. Dony, R. Almsie'deen, Identifying traceability links between product variants and their features, in: 1st International Workshop on Reverse Variability Engineering (REVE 2013), 2013, pp. 17–23.

[41] A. Olszak, B. Nørregaard Jørgensen, Remodularizing Java programs for improved locality of feature implementations in source code, Science of Computer Programming 77 (2012) 131–151. DOI 10.1016/j.scico.2010.10.007.

[42] Z. T. Sinkala, S. Herold, Hierarchical code-to-architecture mapping, in: P. Scandurra, M. Galster, R. Mirandola, D. Weyns (Eds.), 15th European Conference on Software Architecture (ECSA 2021), 2022, pp. 86–104. DOI 10.1007/978-3-031-15116-3_5.

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Fernandes reports financial support and article publishing charges were provided by Foundation for Science and Technology. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Science of Computer Programming

An industrial experience of using reference architectures for mapping features to code

Karam performed conceptualisation, methodology, software, validation, formal analysis, investigation, data curation, writing - original draft, visualisation.

João was responsible for conceptualisation, methodology, writing - original draft, writing - review and editing, and visualisation.

André performed resources, writing - review and editing.