Analysis of Runtime Feature Dependency Relationships in Product Line Assets

Saad Bin Abid
Lero-The Irish Software Engineering Centre
University of Limerick, Ireland
Saad.binabid@lero.ie

Abstract—Features in a feature-oriented software product line interact and depend on each other in unexpected ways. Runtime feature dependency relationships implement the runtime behaviour of the end product derived from a product line. Analyzing runtime feature dependency relationships can facilitate the product line engineer to predict faults upfront during the platform development of a product line. The aim of this research is to facilitate a product line engineer by providing the consistency checking of runtime feature dependency relationships in the product line assets (i.e., between runtime feature dependency relationships specification and their respective implementations using aspect pattern-based solutions). We propose a four-step technique for consistency checking of runtime feature dependency relationships between the product line artefacts. We provide a tool-support based on Eclipse framework as set of plug-ins. For validation purposes we use an existing scientific calculator product line.

I. INTRODUCTION

Software product line engineering has rapidly emerged as an important software development paradigm during the last few years. SPL engineering promises benefits such as “order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers” [1].

In a feature-oriented software product line, functionalities of a product line are observed and implemented as a set of common and variable features [2]. Features realizing the functionality of an established product line often interact with and depend on each other. This phenomenon is called feature dependency and it can be observed at different levels of abstraction in a product line [3, 4].

Although understanding feature dependencies is critical in a product line asset development, they have not been analyzed and documented explicitly [3]. Inability to manage feature dependencies between different levels of abstraction, (i.e., between feature model and implementation levels) can lead to erroneous product configuration and derivation. According to [5], "Feature interactions are situations in which the combination of features leads to emergent and possibly critical behavior are a major source of failures in software product lines”.

Runtime feature dependency relationships implementing the behaviour in products of the product line can be observed at different levels of abstraction, in different product line assets, including the feature model level and the implementation level. We propose a research process aided with a prototype to facilitate a product line engineer to detect whether certain types of runtime feature dependency relationships specified using the feature model are implemented as intended using the existing design aspectJ patterns developed in existing work in [8, 9]. This research work is an extension to already work [10, 11].

This research paper proposes a four-step technique for consistency checking of runtime feature dependency relationships between the product line assets. This process begins with modelling the implementation. We then use existing reverse engineering techniques to perform a code-to-model transformation. Pattern detection algorithms are then used to identify patterns implementing runtime feature dependency relationships in the source code. The simple detection of a pattern does not guarantee that the behaviour is implemented as intended. We then apply model level constraints to provide feature dependency relationships related feedback to a product line engineer.

The rest of this paper is structured as follows. In Section 2, we discuss the research focus by providing an overview of runtime feature dependency relationships and motivational scenario for developing the approach. In Section 3 we present our consistency checking approach in a form of a process model. In Section 4 we provide a concrete implementation of the consistency checking approach and in Section 5 we discuss the benefits and limitations. Sections 6 and 7 discuss related and future work respectively while Section 8 concludes the paper.

II. RESEARCH FOCUS

Feature dependencies have to be implemented separately in order to facilitate a reusable product line asset development process and the understanding of the product line functionalities [12-15]. Feature dependencies are embedded into feature implementations in the source code and unable to manage feature dependency relationships can result in the tangled code issue. Unable to understand and manage the embedded feature dependencies in the source code can also lead towards error prone product derivation and assembly during the AE process of SPL. Work by Lee et al.,[6] provide a modular solution to implement separately runtime feature dependencies in a product line features implementation base. This work is an extension to existing work started by Lee et al., [6] by providing a consistency checking approach between specification view of runtime feature dependency relationships and their respective aspectJ-based implementations in DE process of a product line. This section provides the research focus by providing an overview on
runtime feature dependency relationships and the motivational scenario to develop an approach for the consistency checking of the runtime feature dependency relationships in product line assets.

A. Runtime Feature Dependency Relationships

Lee and Kang [12] define operational and activation dependencies. Operational dependency relationships are defined as “directly or indirectly create relationships between features during the operation of the system in such a way that the operation of one feature is dependent on one of those of other features”.

Types of operational dependencies are as following: 1) “A usage dependency between two features means one feature (a usage client) depends on other feature (a usage supplier) for its correct functioning”, 2) Modification dependency relationship is “A feature (a modifier) may modify the behavior of other feature (a modified) during its activation” and 3) Activation dependency is defined “if an activation of one feature depends on that of other feature”.

Activation dependency relationships are type of operational dependencies. Lee, Botterweck [8] further sub-classified activation dependencies into four categories namely, 3.1) in exclusive-activation dependencies features should be mutually exclusive each during activation, 3.2) subordinate activation dependency: during activation of features there may exist a feature (a subordinate) which can only be active while other feature (a superior) is active, 3.3) concurrent activation dependency: there may exist a scenario during activation where some subordinates of a superior are active at the same time while the superior is active and 3.4) sequential activation dependency: where subordinates of a superior may have to be active sequentially while the superior is active.

B. Motivational Scenario

A problematic scenario related to runtime feature dependency relationships is when two features are specified to access a shared resource sequentially but during the runtime environment both feature’s implementations are unable to access the resource in a sequential manner. For functionally correct products runtime feature dependency relationships specified in the feature model should comply with their respective implementations in the source code. This research work is conducted in the context of the product line maintenance and evolution, where a product line engineer wants to inquire if the particular type of runtime feature dependencies is implemented as intended (i.e., using a particular design pattern) in the source code.

During the DE process the software assets are developed and maintained to address not only the software system specification but also the market needs. It is often seen that the product line assets are developed in distributed and collaborative environment [16]. Different stakeholders are involved during the product line asset development. Figure 1 depicts a scenario that motivates the need for developing runtime feature dependencies analysis approach. It is possible that different stakeholders are developing artefacts and there may exists no or partial mappings between the artefacts. For instance, a requirements engineer might be using the feature model to specify features that represent the functional and the non-functional capabilities along with the feature dependencies (e.g., structural, configuration and runtime dependencies), a developer implementing the product line features and the feature dependencies in source code.

Managing the product line artefacts is one of the product line engineer’s responsibilities in the domain engineering (DE) process of an established product line. There arise different tasks that product line engineer needs to cope with, for instance 1) variability management, 2) product line evolution and 3) consistency checking in different product line assets. In this context, the goal of this work is to facilitate the product line engineer to enable him detecting if the runtime dependencies specified on feature model are implemented as intended and consistent with in the features implementation.

In this research work, we propose a runtime dependencies management approach to help a product line engineer to detect the inconsistencies related to the behaviour specified in the feature model and their respective implementations in the source code by applying the design patterns suggested in earlier research [8, 12, 13] during the DE process. Even though the design patterns suggested by [8, 12, 13] provide a practical solution for isolating runtime feature dependencies in the source code, the challenge remains that the product line engineer managing the product line assets has to detect any inconsistencies between the specified runtime feature dependencies using the feature model and their respective pattern-based implementations. Formulation of a product line
III. CONSISTENCY CHECKING OF RUNTIME FEATURE DEPENDENCY RELATIONSHIPS

This section presents the theoretical contribution in the form of a model-driven technique that allows a product line engineer to semi-automatically detect inconsistencies caused by the contradictions between runtime feature dependency relationships specified in the feature model and their respective implementations in order to achieve a functionally correct product.

A. Inconsistency Scenarios

This section discusses some of the example inconsistency scenarios related to the runtime feature dependencies. There might be various types of inconsistency scenarios based on the simple example describe in Figure 2.

The example sketched in Figure 2 represents two features namely A and B specified in the feature model along with the particular type of runtime feature dependency between them. The particular type of runtime feature dependency specified can be of any type discussed in Sub-Section A in Section II. Features A and B along with the particular type of runtime feature dependency are implemented in the source code. The following sections briefly discuss some of example inconsistency scenarios related to the implementations of the runtime feature dependencies (RTFDs).

![Fig. 2. An example to demonstrate inconsistency scenarios in the implementation of RTFDs.](image)

B. No Runtime Feature Dependencies Implementation

An operational inconsistency may occur if for instance in the feature model a runtime modification dependency is specified between two features A and B, but there is exists no corresponding implementation of the specified runtime modification dependency in the source code. This kind of inconsistency scenario applies to all other types of runtime feature dependency relationships (i.e., modification, required activation, excluded activation, sequential activation and concurrent activation). Figure 3 represents an overview of the problematic scenario.

![Fig. 3. No implementation of specified runtime feature dependency relationship.](image)

C. Incompatibility between the Specified Type and it’s Respective Implementation.

An inconsistency may occur if the specified runtime feature dependency relationship type between two features A and B is different from what is implemented in the source code. For instance, the specified type by the requirement engineer was runtime modification dependency relationship type whereas the developer implemented the runtime required activation type. This situation causes the wrong behavior being implemented and it leads towards faulty end product. This inconsistency scenario applies to all types of runtime feature dependency relationships (i.e., Modification, Required Activation, Excluded Activation, Sequential Activation and Concurrent Activation).

D. Incompatible Specified Features Pair and it’s Respective Implementation Pair.

An inconsistency may occur if the specified runtime feature dependency relationship type between feature pair is not implemented accordingly between the respective feature pair. For instance, Runtime required activation is specified between two features A and B but the developer implements the Runtime required activation between feature A and C. This situation causes the specified behavior implemented between wrong pair of features. Hence the required behavior is not achieved leading towards faulty end product. This scenario applies to all types of runtime feature dependency relationships (i.e., Modification, Excluded Activation, Required Activation, Concurrent Activation and Sequential Activation).

E. Incomplete Implementation of the Runtime Feature Dependencies.

An inconsistency may occur if the runtime modification dependency is specified between two features A and B in the feature model, but there is an incomplete implementation of the runtime modification dependency in the source code. An incomplete implemented runtime modification dependency means that the code responsible for implementation fails to modify certain parts of either of the implementations of feature A or feature B in order for both features to operate/function together during execution of the product containing both features. This kind of inconsistency scenario
applies to all other types of runtime feature dependency relationships (i.e., Modification, Required Activation, Excluded Activation, Sequential Activation and Concurrent Activation).

**F. Implementation of Runtime Feature Dependency Relationship Not Preserving the Specified Direction.**

A behavioral inconsistency occurs if the implementation of a specified runtime feature dependency direction between the features is not preserved. For instance (see Figure 4) in the feature model runtime modification dependency between A (Source) and B (Target) is specified meaning that in order for both features A and B to work in the runtime environment the implementation of feature A (Source) needs to be modified and not the implementation of feature B (Target). The semantic of the direction between two features A and B comes from the system’s specifications. Inconsistency is said to occur when the implementation of runtime modification modifies the implementation of feature B (Target) and not the implementation of feature A (Source).

This applies to all other types of runtime feature dependency types (i.e., Required Activation, Excluded Activation, Concurrent Activation and Sequential Activation). For instance, Runtime Sequential Activation specified between two features A and B means that feature A should be sequentially active before feature B during the execution of the end product. Whereas the implementation enables feature A to be executed after feature B during the execution of the end product. This scenario leads towards faulty end product with features A and B not working as intended during execution of the end product.

For the proposed technique Eclipse Modelling Framework (EMF) [15] is used to perform the domain modelling. As a result of performing the domain modelling three interrelated meta-models namely the dependency-oriented feature model (DOFM), the implementation model (IM) and the pattern model (PM) are developed. Each of the developed domain model is discussed as following.

1) **Dependency-Oriented Feature Model (DOFM) Modelling Language.**

The DOFM modelling language specifies the product line features and the runtime feature dependencies between the specified features. The DOFM also enables the mappings between features and their respective runtime feature dependencies to their respective implementations in the implementation model (IM).

2) **Implementation Model (IM) Modelling Language.**

The implementation model modelling language provides a structured language to develop the implementation model composed of the implementation related concepts. The IM modelling concepts have mappings to the pattern model (PM) concepts. The IM modelling language enables one to model Java and AspectJ concepts at a higher level of abstraction.

3) **Pattern Model (PM) Modelling Language.**

The PM modelling language provides a structural language to specify the AO-patterns established in the work Lee, Botterweck [6]. The PM modelling language concepts are also mapped to the IM modelling concepts.

The Extraction of source code concepts can be performed automatically by performing the source code parsing and abstract syntax tree (AST) manipulation techniques to mine the source code concepts. An AST provides implementation concepts of interest at a higher level of abstraction as a hierarchal structure to be traversed. The source code of the implementation under study is first parsed then the resulting AST is traversed and used as input to instantiate the IM DSL meta-model. The output of this process is a populated instance of IM DSL meta-model.

C. Mapping Dependency-Oriented Feature Model to Implementation Model.

Mapping the dependency-oriented feature model (DOFM) concepts to their respective implementations in the implementation model (IM) is manually performed (e.g., by naming convention) by the product line engineer. However these techniques are not in the scope of the proposed technique. Development of DOFM is manually performed by the requirements engineer. The process also takes into account the mapping knowledge while performing this process of integrating DOFM and IM by the product line engineer. The output of the process is mapped DOFM and IM.

D. Detection of Inconsistencies in the Implementations of the Runtime Feature Dependencies.

Detection of inconsistencies due to contradiction between specified runtime feature dependencies and their respective implementations process enables the detection of inconsistencies in the implementations of the RTFDs using the integrated DOFM and IM. This process has two sub-processes.

1) Aspect-Oriented (AO) Pattern-based Implementation Detection.

The product line engineer executes developed AO pattern-based implementation detection algorithms on the project source code. The AO pattern detection algorithm for each type of runtime feature dependencies is developed. AO pattern-based implementation detection process takes the source code as an input and generates the populated instance of PM meta-model composed of the detected runtime feature dependency relationships. The detected runtime feature dependency relationships concepts are automatically mapped to the respective IM concepts. The output of this process is mapped DOFM, IM and PM meta-models instances.

The developed algorithms perform the static code analysis using the generated AST of the source code of the product line. The developed algorithms for AO pattern search are the directed search algorithms inspired by the work by Heuzeroth, Holl [16]. The algorithms used the generated AST concepts generated as a result of parsing the source code to identify the AO pattern parts. The static analysis algorithm computes the AO pattern relation on the source code concepts and provides the result as a set of candidates, i.e., a set of tuples of IM concepts with the appropriate static structure. This set is a conservative approximation to the actual AO pattern based implementation in the source code. The process then populates the Pattern Model (PM) with the searched AO pattern candidate concepts as a result of identified source code concepts related to the AO pattern. The output of this process is an integrated DOFM, IM and PM. The PM contains all the detected AO patterns. The detected AO patterns are linked to the implementation concepts/nodes in the IM.

This process is applying directed search algorithms for static analysis of the source code by manipulating the IM nodes and producing the candidates set. However the dynamic analysis techniques are not in the scope of the proposed technique.

2) Applying Constraints on Detected Aspect-Oriented Patterns.

Identified AO patterns in the PM are set of PM concepts tuple with an appropriate static structure. Next important step is to query if

- There exists AO pattern-based implementation
- And also if the identified AO pattern in the PM between the IM concepts
- Have complete AO pattern structure with all related concepts (e.g., classes and function calls)
- Are same as that of mapped IM concepts

In order to perform the above tasks constraints are applied on the identified PM pattern concepts tuple and structure along with the specified runtime feature dependency relationships using the FM. Constraints are applied on the PM model concepts in order to ensure that certain properties of a concept exist. Applying constraints on detected AO patterns process is carried out automatically by applying the Epsilon Validation Language (EVL) based constraints [17]. EVL constraints supports both inter and intra-model consistency checking, constraint dependency management and specifying fixes that users can invoke to repair identified inconsistencies.

Applying EVL constraints on the detected AO patterns enables the product line engineer to detect inconsistencies due to contradictions between the specified RTFDs and their respective implementations. The errors and warning messages generated based on the applied EVL constraints enables the product line engineer to make manual analysis on the AO pattern-based implementation of runtime feature dependency relationships.

However there is still an element of error involved because it is also possible that the generated error messages/warnings by the EVL are not correct and can be categorized as a false positive or a false negative since the feedback is generated based on applying EVL constraints on the detected AO patterns in the PM. And the AO pattern detection algorithms may detect incorrect AO patterns in the IM. Some of the errors and warning messages provided to the product line engineer for further analysis are:

- AO pattern-based implementation not found in IM
- AO pattern-based implementation found but not implementing the specified type of runtime feature
dependency in the DOFM (i.e., specified type is X but implemented type is Y)

- AO pattern-based implementation found but not between the specified features in the DOFM
- AO pattern-based implementation found but direction of implemented behaviour is incorrect

V. IMPLEMENTATION

The proposed technique is realized as a part manual and a part automatic. To automate the processes of the proposed technique a tool chain is implemented in Eclipse development environment [15]. The sub-sections of this section provide an implementation overview of the proposed technique Figure 6.

A. Code to Model Transformation and AO-Pattern Detection Plug-in

In order to automate the code to model transformation and AO-pattern detection processes of the proposed technique, an Eclipse-based plug-in is developed. The developed plug-in is applied on the Java/aspect-J implementation. After applying the plug-in the implementation meta-model and Pattern Model meta-model instances are created. Figure 7 represents applying the Code2ImplModel and AOP-Pattern detection transformation plug-in on an example Java/aspect-J implementation project. The developed plug-in realizes processes (1) and (3) in Figure 5. Figure 8 shows the generated instances of IM and Pattern Model meta-models.

B. Mapping Specified Concepts in DOFM to Respective Implementation in IM

This realized process (Process 2) of the proposed technique described in Figure 6 is manually performed by the product line engineer.

C. Applying EVL Constraints

After generation of the Pattern Model meta-model instance, the next step is to apply EVL constraints on the detected AO-patterns along with the DOFM and the IM in order to validate the detected AO-patterns. Error markers are generated in the Eclipse error view after the application of the EVL constraints (please consider Figure 9).
VI. MUTATION TESTING-INSPIRED APPROACH FOR VALIDATION

The challenge addressed by this work focuses on heterogeneous product line artefacts (i.e., feature model and implementation) developed by various stakeholders. In Mutation Testing (MT) the program under test is mutated. Mutation testing [18] is a practice commonly performed to validate the correctness of source code, but in the scope of the thesis research work both the runtime feature dependencies specification and its respective pattern-based implementation can change. The choice of changing the specification along with implementation is made because of the fact that the specifications and their respective implementations are dealt with by various stakeholders in a distributed environment. Specification change is called specification mutants and implementation change is called implementation mutants. The artefacts can be changed independently by the stakeholders. An example of a specification mutant can be generated by changing the specification of a particular runtime dependency relationship type with one known problem or inconsistency. On the other hand, an example of an implementation mutant can be generated by commenting out one part of the aspectJ pattern implemented in the source code of a respective runtime feature dependency relationship type. A mutant has a mutant ID and one known problem or inconsistency.

Once the mutants are created for each type of runtime feature dependency relationship the proposed technique and tools are then validated against the developed mutants. If the inconsistencies in the mutants (i.e., specification mutants or implementation mutants) are detected then the mutants are said to be killed and validation is said to be adequate. Mutation testing is assessed by mutation coverage technique. Mutation coverage is equal to number of number of mutants killed. In the context of validating the proposed technique by applying it on the scientific calculator product line (SciCalc) case study [6] a mutant is said to be killed if the proposed technique is able to identify the deliberate change made either in the specification (i.e., the feature model) or in the AspectJ-based pattern implementation. The more the proposed technique is able to detect the change/mutation the more the proposed technique is considered to be successful.

VII. DISCUSSION OF VALIDATION RESULTS

This Section discusses the validation results of the proposed technique applied on the SciCalc product line case study. Both the implementation mutants and the specification mutants are created for each type of runtime feature dependency relationships and validated against the proposed solution.

One of my initial findings are that the work on the classification, aspectual separation and modularization of runtime feature dependencies in the source code using the aspectJ-based patterns started by Lee et al. [6, 7, 10, 12]. As discussed earlier that due to lack of the available case studies only SciCalc product line case study was selected for validation purposes. In order to fulfill the need of having multiple case studies mutation testing-inspired validation technique was adopted to perform the validation of the proposed technique and provided tool support. Mutants (both specification and implementation) were created and the proposed technique was applied in order to kill the mutant. Killing a mutant means that the proposed technique when applied on the mutant is able to identify the problem.

In the case of the modification dependency which is implemented as a generic aspectJ class, 12 of the known mutants were created and killed by the proposed technique hence the coverage is 100%.

In the case of runtime required and excluded activation dependencies mutants (both specification and implementation) 22 out of 24 mutants were killed.

In the case of runtime sequential activation dependency 20 known types of mutants are generated for the validation purposes. Out of 20 known mutants 18 were successfully killed hence mutation coverage or validation is 91.6% is achieved.

In the case of runtime concurrent activation dependency again 20 known types of mutants are generated for the validation purposes. Here I would like to mention that the SciCalc product line does not contain any of the runtime concurrent implementation examples hence I created a demo example implementation between M5 and M1 features and performed that validation on the demo example. Out of 20 known mutants 18 were killed hence the mutation coverage is 91.6%.

VIII. RELATED WORK

Work started by [5] attempts to check feature interactions in software product lines between the specified features and their behavior in separate composable units. The approach tries to make sure that the features of the product work properly together.

Work by [19] provide a formal model called coloring algebra is inspired by existing work called Colored Integrated Development Environment (CIDE) [20]. The work focused on structural feature interactions. The coloring algebra gives formal meaning to the sequential, the cross-product and the interaction operation. Each colored part of the code represents a feature in the code and coloring code is also used for representing feature interactions in the feature implementation. CIDE approach lacks modularity as it represents an SPL as a single, general program rather than separate composable feature modules. The interactions colored in the
implementation is then represented a tree if interactions called a derivative tree. A general framework is defined to analyze feature interactions and compositions. Both the work by [5] and [19] focus on detecting feature interactions in the source code. Lee and Kang [10] classified the feature dependencies into runtime feature dependencies.

The work by Lee and Kang [10] differs from [5] and [21] as it provides classification and apply modular pattern-oriented solution in source code to manage feature dependency relationships in product line. Both of the approaches resembles to the proposed approach in this thesis work to some extent but differs as it is not only focusing on feature dependency relationships that are modularized using particular design patterns but also to analyze them for behavioural consistency between their specifications and respective implementations.

IX. FUTURE WORK

This section discusses the future work and possibilities of improving the proposed approach. Only Scientific calculator product line is utilized for the proposed technique validation inspired by the mutation testing technique. It is in the agenda to investigate further case studies and validate the approach. Based on the feedback I plan to improve the proposed technique. At the moment the proposed technique has structural editor views to specify the runtime feature dependencies, view the implementation and pattern models. In future I’ll try to explore the visualization techniques to improve the proposed technique efficiency. It is in the agenda to initially identify the inconsistency on the model level and then trace the inconsistency to source code part (i.e., classes, functions, etc.) having the inconsistency and vice versa. It is in the future work agenda to provide the usability studies of the proposed technique with the industrial tool users and to perform the empirical studies and analysis of the proposed technique with respect to the existing tool support for detecting the inconsistencies related to various types of feature dependencies in the software product line domain.

X. CONCLUSION

This work is an extension of the existing work started in [8, 9]. I have presented a research work supported by a prototype on consistency checking of runtime feature dependency relationships in product line assets. The proposed technique process model along with implementation is discussed in detail. Mutation testing inspired validation of the proposed technique is discussed using the existing SciCalc product line case study.

REFERENCES


