Tracing Aspect-Oriented Patterns for identifying feature dependency inconsistencies in Software Product Lines

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

ABSTRACT

Features implementing the functionality in a software product line (SPL) often interact and depend on each other. It is hard to maintain the consistency between feature dependencies on the model level and in the actual implementation over time, resulting in inconsistent SPL with less ability for inclusion and exclusion of features during product derivation. We describe our initial results when working with feature dependencies implemented as aspect-oriented patterns and the related inconsistencies. Our aim is to improve feature dependency analysis for inconsistency identification between traceable modeled artefacts in SPL. In this research work we argue that traceability can facilitate identification of feature dependency inconsistencies. We report our experience of feature dependency inconsistency scenarios on different abstraction levels and our intended approach using traceability modeling to analyze feature dependency inconsistencies in the context of an existing scientific calculator product line.

1. INTRODUCTION

A software product line (SPL) is a "set of software intensive systems sharing a common, managed set of features that satisfy the specific market segment or mission and that are developed from a common set of core assets" [1]. Using an SPL approach allows companies to realize significant improvements in time-to-market, cost, productivity, and system quality [2]. The products of a SPL family differ by the features they include in Feature-Oriented Domain Analysis (FODA) [3]. It is common to find cross cutting variable features during FODA. Cross cutting features are the features whose functionality spans over several parts of an application. Cross cutting variability makes it difficult to map features to architectural design and then to implement these variabilities in source code. Features are not in general independent of each other. Changes in the implementation of one feature will cause side effects in the implementation of other features [4]. The problem is caused due to the fact that feature dependencies are embedded into feature implementations, resulting in tangled code issues.

Features are not usually independent of each other. Features in established SPL interact and depend on each other. As SPL may consists of heterogeneous artefacts with inter dependencies, it is hard to map and manage inter artefact dependencies. Especially managing consistency in SPLs becomes more challenging when feature dependencies are taken into consideration along with their mappings across heterogeneous SPL artefacts.

According to [6], in distributed and collaborative development setting, different models may be developed in parallel by different persons. Since models are developed independently so there is a possibility that inter and intra dependencies between developed models may be poorly understood. Inconsistency in SPL can arise due to, 1) incompleteness of the product line artefacts, 2) dependencies among model elements intentionally or unintentionally in order to avoid premature decisions. It is possible that different stakeholders are developing artefacts and there exists no or partial traceability knowledge between the artefacts, for instance a requirements engineer using feature models to represent the functional and non functional capabilities and dependencies among them, a developer working with project implementation and implement SPL features and dependency among them in source code. Since artefacts with different abstraction levels in SPL are created by different stakeholders (requirements engineer and developer) there may arise few complex challenges for the product line manager to deal with for instance, **1**) analysis of feature dependency for SPL consistency, **2**) identifying feature dependency inconsistencies either in code (feature dependency implementation) or in feature model (requirements) **3**) How to interact with feature dependency inconsistency to manage them in SPL? **4**) How to extract feature dependencies between SPL artefacts and manage them for feature dependency analysis in SPL over time?

Traceability enables to relate and map heterogeneous artefacts. According to [8], traceability can facilitate product line engineer to perform product configuration and derivation tasks more efficiently. Traceability is considered crucial for establishing and maintaining consistency between different software artefacts such as requirements documents, architectural design, detailed design, source code and test cases. When considering feature dependency inconsistency management traceability can play a positive role while managing (defining, recording, retrieval) feature dependency in SPLs [9]. Extracting and managing inter and intra traceability links between modelled SPL artefacts on different abstraction levels can help maintain consistency of an established product line. Traceability is defined as "*Traceability in software development is the ability to relate the different artefacts created in the development life cycle with one another*" [7]. The challenge of traceability extraction and management in product lines becomes even more challenging when feature interactions and dependencies are taken into consideration. Even though there are several solutions (traceability meta-modeling, traceability using transformation languages) available for extracting and managing traceability between different model-driven development (MDD) artefacts but still current approaches are not taking traceability as first class entity to manage feature dependency consistency in model-based product lines.

Traceability management in SPLs is itself a challenge. The *Centre of Excellence for Traceability Technical Report (COET-GCT-01-01)* [10] discusses the grand challenges in the area of traceability.
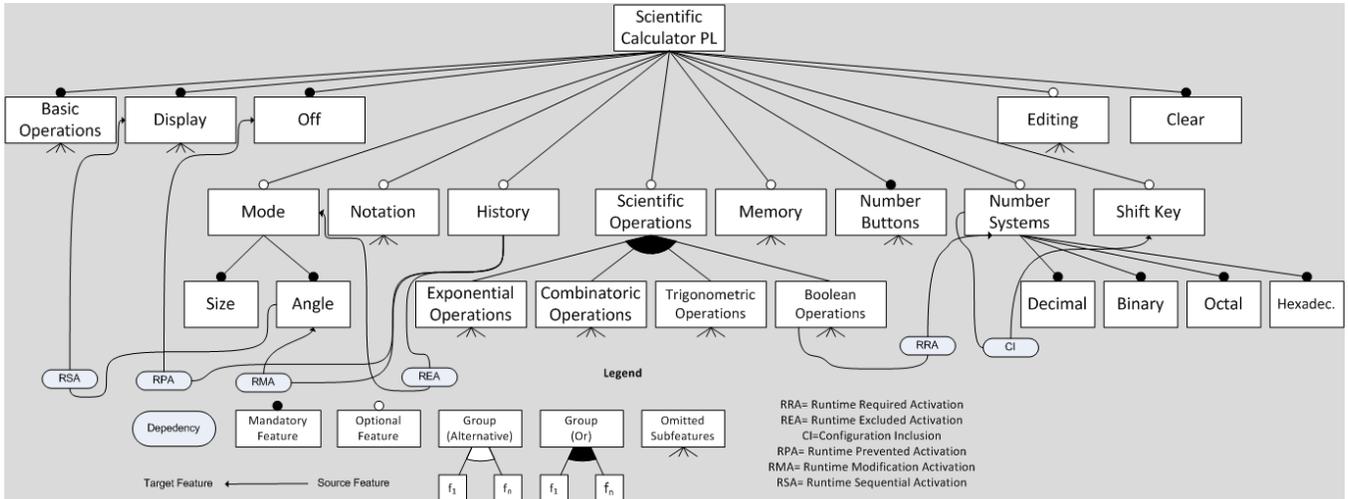
Figure 1. Scicalc-PL Feature Model and dependencies (an excerpt)

The aim of our research work is to use existing traceability modeling techniques for capturing feature dependency concepts between requirements, architecture and implementation artefacts in SPLs. The traceable modeled feature dependency concepts are than used to analyze feature dependency on different levels of abstraction facilitated by visualization techniques in SPLs.

Our work is based on the existing work done by authors in [4,5]. Authors in [4] have used aspect-oriented patterns (AO-patterns) implemented in example scientific calculator product line (Scicalc-PL) [4,5] on abstract level of requirements implemented as feature model and actual implementation project to manage feature dependencies in source code. In this paper the research work presented has two main contributions; 1) we have identified several different inconsistency scenarios when using AO-patterns [11] in Scicalc-PL, 2) we have presented our approach using existing MDD techniques/languages to analyze identified scenarios for SPL consistency management. The research questions our research work is addressing and aiming to resolve are **1**) how to map feature dependencies at different level of abstractions from requirements modeling till actual implementation in SPLs, **2**) how to trace the inconsistencies when feature dependencies are implemented as AO-patterns in SPLs?, **3**) to what extent existing MDD tools and processes facilitate feature dependency analysis in SPLs?

The remainder of the paper is organized as follows, Section 2 discusses the background on which our work is based, Section 3 discusses inconsistencies scenarios identified, Section 4 discusses our approach to solving the feature dependency inconsistencies identified, Section 5 discusses related work and Section 6 concludes the paper and provides future direction for our intended approach.

2. BACKGROUND: FEATURE DEPENDENCY

In this section we are going to briefly discuss existing background work in feature dependency management which this research work is based on. There have been several comprehensive attempts [12], [11-13] to design feature dependencies in a SPL. The work in [11], authors provide aspect-oriented patterns for managing feature dependencies. Authors have used AspectJ programming language [19] to implement the patterns in an example case study.

We are going to give a brief overview of identified feature dependencies outlined in [11] as **1**) configuration dependencies, **2**) operational dependencies and **3**) activation dependencies. Configuration dependencies are responsible for constrain selection of variable features in terms of "required" and "excluded" dependencies. Required configuration feature dependency outlines that when one feature is selected for a product other feature should also be present in the same product. Whereas excluded configuration feature dependency means that both features cannot be present in the same feature. Operational dependencies mean that "*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 usage dependency "*A usage dependency between two features means one feature (a usage client) depends on other feature (a usage supplier) for its correct functioning*". Modification dependency is another type of operational dependency where "*a feature (a modifier) may modify the behavior of other feature (a modifyee) during its activation*". Activation dependency is defined "*if an activation of one feature depends on that of other feature*". Authors in [11] have classified activation dependencies in four categories namely, **1**) *exclusive activation dependency*: where activation of features in exclusive-activation dependencies should be mutually exclusive each other while during activation, **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**) *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, **4**) *sequential activation dependency*: where subordinates of a superior may have to be active sequentially while the superior is active.

3. FEATURE DEPENDENCY INCONSISTENCY SCENARIOS

Previous section briefly discussed the background and basis of this research work. This section includes a concrete example to identify feature dependency inconsistency scenarios.

3.1 Scientific Calculator Product Line

We have taken calculator product line to identify feature dependency inconsistency scenarios. We have taken calculator

product line due to its simplicity and understandability. The product line is capable of generating different products with various functionalities that includes different types of scientific functions (e.g., sin, cos, tan etc) number systems (e.g., hexadecimal, binary, octal etc). Authors in [11] have used the same Scicalc-PL to demonstrate the effectiveness of their established aspect-oriented feature dependency patterns in Scicalc-PL implementation. So naturally it makes Scicalc-PL an initial case study to identify feature dependency inconsistency scenarios.

Figure 1 shows an excerpt of feature model from Scicalc-PL. In addition to the commonality and variability information shown in Figure 1, there are various dependencies between features. The following are few of the dependencies between the features shown in Figure 1. **1)** *Boolean Operations* requires *Number Systems* to be activated during activation, **2)** *Number Systems* requires *Shift Key* to be included during activation, **3)** *Angel* requires *Display* to be active sequentially, **4)** *History* requires *Off* to be prevented during its activation, **5)** *History* modifies *Angle* during its activation, **6)** *History* requires *Mode* to be excluded during its activation.

Work in [4] uses different naming schemes as patterns in feature model to represent the feature dependencies on higher level of abstraction. These naming schemes are implemented as aspect-oriented patterns already discussed in Section 3 as separate AspectJ programming aspects. Scicalc-PL implementation artefact contains Java classes and Aspects. The dynamic cross cutting mechanisms (i.e., Pointcut and Advice) of AspectJ are used to extend one feature's interaction part with other functional features functionality which actually implements the modification dependency. The activation dependencies are implemented using generic aspects.

3.2 Identified Feature Dependency Inconsistency Scenarios

In order to identify potential challenges for managing dependency aspects consistency, we analyzed the example Scicalc-PL. The inconsistency scenarios defined in this section are in the context of research work conducted in [5]. The extracted inconsistency scenarios are based on aspect-oriented patterns established in [11] for managing feature relations and dependency management in SPLs. Initial set of feature dependency inconsistency scenarios is established and published in [9]. These are not the only type of inconsistency scenarios that can occur but for our research we have identified and considered only the following inconsistency
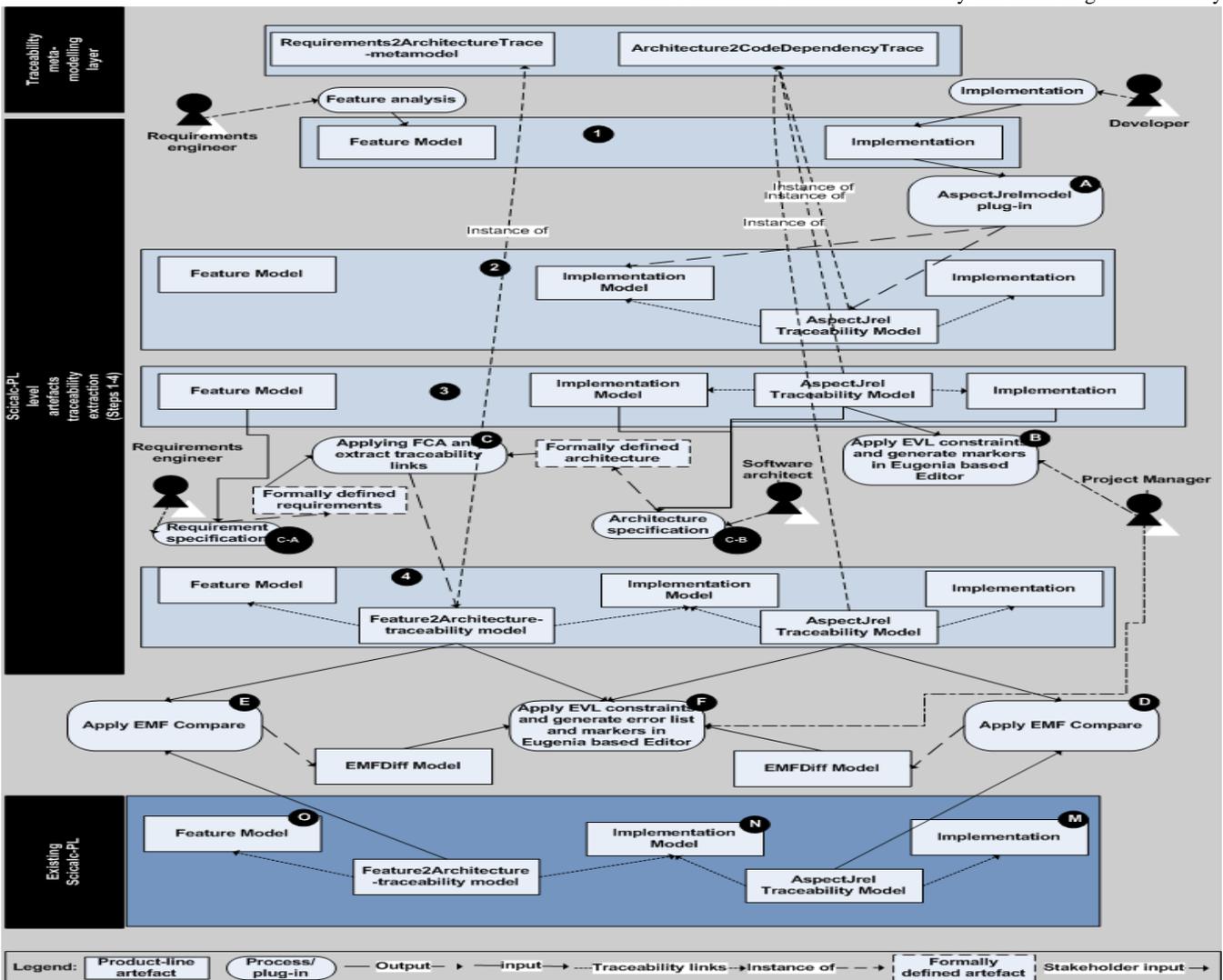

Figure 2. Intended approach for analyzing feature dependency in SPLs

3

scenarios with in Scicalc-PL.

3.2.1 Feature dependency missing in requirements model or implementation level

During our analysis of Scicalc-PL model-based product line, we observed that inconsistency occurs when feature dependencies are **1**) not implemented in source code, **2**) not included and synchronized with the implementation model (AML model) and then mapped to dependency representing it in the feature model, **3**) not implemented as feature dependency in the feature model. This type of inconsistency can also greatly affect product derivation due to the fact that it can produce product that are non functional or of less functionality. For instance, In Figure 1 feature *Boolean Operations* has required activation dependency with featur*e Number System*s. This required activation dependency is represented by a CI node (oval in Scicalc-PL feature model) in a feature model and implemented as an aspect in Scicalc-PL implementation. An inconsistency can occur when either the CI node is missing in the feature model that represents feature dependency or in the actual code an aspect that implements this dependency relationship is not implemented. This inconsistency also causes Scicalc-PL product line engineer to derive less functional or erroneous products as this situation won't let the final aspect to be included in the derived product that implements the runtime activation dependency between features *Boolean Operations* and *Number Systems*. It makes product line engineer's job more challenging how to find out if the feature model contains the missing feature dependency and not the implementation missing aspect implementing feature dependency or vice versa.

3.2.2 Implementing wrong feature dependency in requirements model or implementation level

An inconsistency can occur when dependency aspects are implementing the wrong feature dependencies. For instance, in Figure 1 feature *History* requires feature *Mode* to be excluded during its activation. An inconsistency can occur if aspect implementing runtime excluded dependency is implemented by a wrong aspect or in other words if rather than aspect implementing exclusion functionality it implements inclusion dependency in actual code. Another scenario can occur on requirements model level if requirements engineer implement wrong feature dependency in Scicalc-PL feature model using a dependency type node to represent feature dependency. As a result during product derivation it makes a hard challenge for product line engineer to analyze where did the actual problem lies? Is it on model level where requirements engineer worked or is it on the implementation level where programmer worked? This applies to all other types of feature dependency aspect-oriented patterns as well discussed in Section 3.

3.2.3 Partially implemented feature dependency in requirements model or implementation level

During analysis of Scicalc-PL we find out that partially implemented feature dependencies also contribute to raising inconsistency in product lines. For instance, feature *History* has a modification dependency with the feature *Angle* (Figure 1). This modification behavior is implemented by an aspect in the Scicalc-PL implementation. On requirements modeling level it is represented as a dependency type node in feature model. On implementation level, if there is some other method in *History*

that needs to be modified during feature interaction, which is not yet implemented, this leads to a partially implemented feature dependency aspect. In this scenario it is hard for product line engineer to identify if the dependency is fully implemented as on the requirements model level it is implemented accordingly by dependency type node. This applies to all other types of feature dependency aspect-oriented patterns discussed in Section 3.

3.2.4 Feature Sequential dependency order not maintained or implemented in requirements model or implementation level

In Scicalc-PL, there are features which need to be activated sequentially. For instance, the feature *Angle* has Sequential Activation dependency with the feature *Display* in the Scicalc-PL implementation (Figure 1) [4]. Both have to activate in sequence in order to produce a robust and fully functional product during product derivation. The aspect implementing Sequential activation dependency must take into consideration the activation of features having sequential dependency. One possible scenario can be that on feature model source and target features are swapped with each other. In consistent scenario feature *Display* has be activated before feature *Angle* (sequential activation dependency) but in inconsistent situation (when swapped source and target) feature *Angle* will be activated before feature *Display*. This inconsistency scenario makes product line engineer challenge hard to analyze where the inconsistency lies in product line, either on requirements model level of on actual implementation level. This type of feature dependency inconsistency can occur specially in operational dependency types like *subordinate activation*, *concurrent activation* and *sequential activation* dependencies. Such feature dependency inconsistency can also occur in operational feature dependency like modification dependency.

3.2.5 Ambiguous feature dependency in requirements model or implementation level

During analysis of Scicalc-PL, ambiguous feature dependency implementation on either requirements modeling level or implementation level forces product line to be in inconsistent state. For instance, feature *Boolean Operations* has runtime required activation dependency with feature *Number Systems*. An inconsistency can occur if at the same time required excluded activation dependency is introduced intentionally or un-intentionally between feature *Boolean Operations* and feature *Number Systems*. It is hard for product line engineer to identify which one is the correct dependency type. Similarly same situation can occur on the implementation level if developer implements both exclusion and required feature dependency in a form of aspects. During product derivation both aspects may get assembled in the final product hence making final product less functional. So the challenge in this scenario for the product line engineer is to trace which feature dependency is correct and where to find out the information to analyze the inconsistent state of product line.

4. ANALYZING FEATURE DEPENDENCY INCONSISTENCY USING TRACEABILITY

In the previous section, we identified some of the many inconsistency scenarios that can, **1**) force a product line into an inconsistent state, **2**) cause inconsistency issues during model-based product derivation and product configuration, **3**) produce a

faulty product with lesser or erroneous functionality. As motivated in earlier sections, our aim is to facilitate product line engineer make feature dependency inconsistencies analysis to ensure efficient product line consistency management. In this section we are going to introduce our approach. The following are the frameworks and languages we are using to build our prototype approach, **1**) The EMF[1] plug-in development environment for prototype plug-in development, **2**) EMF for traceability meta-modeling, **3**) Formal concept analysis (FCA)[2] for extracting traceability links between feature and architectural models, **4**) Eugenia[3], the Epsilon framework graphical language for developing graphical editor for Scicalc-PL model artefacts, **5**) Epsilon framework validation language (EVL) for applying constraints on models[4], **6**) EMF Compare language[5] to compare models. The following sections will discuss our approach in detail.

4.1 Traceability Meta Model Layer

In our approach we have defined feature dependency traceability Meta model layer consisting of two Meta models as, **1**) requirements to architecture feature dependency trace and **2**) architecture to code/implementation feature dependency trace. Both Meta models provide abstract view on the feature dependency traceability from either requirement to architecture or architecture to actual implementation. The advantage of developing such a traceability Meta modeling layer is twofold, **1**) Firstly, if the product line engineer wants he can customize feature dependency analysis for inconsistency management in SPL between requirements model and architecture facilitated by requirements2architecturefeaturedependencytrace (R2AFD-Trace) Meta model or between architecture2codedependencytrace (A2CFD-Trace) Meta model. If both Meta models are instantiated they can provide full feature dependency traceability from requirements to implementation level and **2**) Secondly traceability meta model layer provides flexibility to product line engineer to customize R2AFD-Trace Meta model and A2CFD-Trace Meta model according to the product line for feature dependency traceability and analysis.

4.2 Extracting Feature Dependency Relationships from Implementations

During analysis of Scicalc-PL, we identified certain inconsistencies (addressed in Section 3) that can occur in Scicalc-PL, if the implementation model (**M**) is not synchronized with actual implementation code base (**N**) (See Figure 2, existing Scicalc-PL artefacts). For this purpose we developed two EMF based plug-ins. See (**A**) in Figure 2 (due to space limitation we didn't show both in the figure). Both **Code2Aml Plug-in** and **Aspectjmapbuilder plug-in** are sub plug-ins of **AspectJrelmodel**

---

[1] http://www.eclipse.org

[2] Formal Concept Analysis (book): Mathematical Foundations, Springer-Verlag, 1999 by Ganter.et.al.

[3] http://www.eclipse.org/gmt/epsilon/doc/eugenia

[4] http://www.eclipse.org/gmt/epsilon/doc/evl

[5] http://www.eclipse.org/modeling/emft/?project=compare#compare

plug-in (A). **Code2Aml plug-in** parses the implementation and creates an implementation model without taking feature dependency aspect relationships into account. Scicalc-PL implementation project is an AspectJ project. In order to extract the feature dependency relationships, we developed another plug-in which actually works with aspects implementing feature dependencies in the Scicalc-PL implementation project. The AspectJ development project maintains an abstract syntax tree (AST) for the project. The **Aspectjrelmapbuilder plug-in** traverses the actual AspectJ AST relationship map of Scicalc-PL implementation to find out relationships between the Java classes and aspects. The plug-in then automatically generates the implemented feature dependency relationships in the already generated implementation model (AML model) using the **code2aml** plug-in.

4.3 Extracting Feature Dependency Relationships between Implementation Model and Requirements

In order to extract feature dependency relationships between feature model and requirements input from two stakeholders is required; Software architect and Requirements engineer is taken into account. FCA is used to extract traceability information between formally defined requirements (**C-A**) and formally defined architecture (**C-B**). In order to formally define architecture of Scicalc-PL implementation model the software architect takes following into consideration **1**) Scicalc-PL implementation, **2**) extracted implementation model and AspectJrel traceability model.

4.4 Analyzing Feature Dependency between Implementation and Implementation Model

During analysis of Scicalc-PL artefacts, we found that there is a need of maintaining feature dependency aspect relationship between implementation model (**N**) and actual implementation (**M**) (Figure.2, Existing Scicalc-PL artefacts). It is because feature dependency implementation relationships in the implementation need to be synchronized with the implementation model at any development stage. To solve the mentioned challenge of maintaining feature dependency relationships, we established **AspectJrelMap** meta-model [9]. The proposed meta-model also acts as a traceability model between the implementation model (AML model) and the actual implementation. It contains the AspectJ implementation concepts (i.e., Advices, DeclaresOn, etc.), which are used to implement features dependencies described in Section 2. Proposed **AspectjrelMap** meta-model can be instantiated for any project implementation developed in the AspectJ development environment. It maintains a snapshot of relationships between Aspects and Java Classes existing in the implementation.

In order to analyze feature dependency implementation between implementation model and implementation following Process **B** is carried out by the project manager. In Process **B** project manager applies EVL constraints on AspectJrel traceability model obtained as a result of process **A**. As a result of failed EVL constraints error markers are generated in the Eugenia based graphical editor.

In order to analyze feature dependency implementations over time and identification of inconsistencies Process **D** is carried out which takes existing AspectJrel Traceability Model (Figure.2

Existing Scicalc-PL artefacts) and newly obtained AspectJrel Traceability Model (Figure.2 Step 3). Process **D** then generates EMF Compare and generates EMFDiff Model. The obtained EMFDiff Model then acts as an input to Process **F**, where Project Manager applies EVL constraints on generated EMFDiff Model. As a result of failed constraints error list is populated and error markers are generated in Eugenia based graphical editor.

4.5 Analyzing Feature Dependency between Feature and Implementation Models

In Step 4 (Figure 2), we obtain the Feature2ArchitectureTraceability meta-model after applying FCA (**C**) that acts as traceability meta-model between feature model and implementation model. In order to analyze feature dependency and identify inconsistencies between Feature Model (**O**) and Implementation Model (**N**) (Figure.2 Existing Scicalc-PL artefacts) over time process (**E**) is performed, that takes existing Feature2ArchitectureTracability meta-model and newly generated Feature2ArchitectureTraceability meta-model and by applying EMF Compare generates EMFDiff Model. The generated EMFDiff Model then acts as an input to Process (**F**), that is apply EVL constraints and generate error markers as a result of failed EVL constraints in Eugenia Based editor.

5. RELATED WORK

Ivkovic et al. [14], introduces and approach to automate identification and encoding of model dependencies that can further be used for model synchronization. Vierhauser et al. [15] applies tool support for incremental consistency checking on variability models in the industrial case study. However the approach is not taking feature dependencies into consideration and works on two variability models, assets and decisions. Colyer et al. [16], presented issues for managing feature dependencies in aspect-oriented software development (AOSD).

6. CONCLUSION AND FUTURE WORK

In the position paper we have identified some of the different types of feature dependency inconsistency scenarios that can force product line to be in inconsistent state and product derivation to be inefficient and error prone. The base work for this research is described and elaborated in [4,5]. Our approach analyzes identified scenarios and its results are intended to improve analysis of feature dependency in SPLs to ensure feature dependency consistency at different levels of abstraction in a SPL and facilitate product derivation and evolutionary changes management. The approach analyzes the identified scenarios (Section 3) in the context of model-based SPLs. Preliminary results are published in [9]. In future we are planning to improve our approach by identifying more scenarios and implementing the approach. It is in our plan to complete the research prototype approach and evaluate it in different case studies.

7. ACKNOWLEDGEMENT

8. REFERENCES

[1] Clements, P. and Northrop, L. M. 2002 *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[2] Pohl, K., Böckle, G., and Linden, F. J. 2005 *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc.

[3] Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, S. *Feature Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[4] Lee, K., Botterweck, G. and Thiel, S. 2009 *Aspectual Separation of Feature Dependencies for Flexible Feature Composition*. IEEE, City, 2009.

[5] Botterweck, G., K. Lee and S. Thiel 2009. Automating Product Derivation in Software Product Line Engineering. Proceedings of Software Engineering 2009 (SE09), Kaiserslautern, Germany

[6] Mens, T. Straeten, R.V.D,.: "Incremental Resolution of Model Inconsistencies". In: Heidelberg, S.B. (ed.): Recent Trends in Algebraic Development Techniques, Vol. 4409/2007. Springer Berlin / Heidelberg (2007) 111-126

[7] Ajila ,S. A. and Kaba, A. B., "Using traceability mechanisms to support software product line evolution," in *Information Reuse and Integration, 2004. IRI 2004,* pp. 157-162.

[8] Abid, S.B. and Botterweck, G., "Resolving Product Derivation Tasks using Traceability in Software Product Lines", *5th Traceability workshop (ECMDA-TW) held in conjunction with ECMDA 2009,* Twente, The Netherlands

[9]Saadbin Abid:"Resolving Feature Dependency Implementations Inconsistencies during Product Derivation", *6th Traceability workshop (ECMFA-TW) held in conjunction with ECMFA 2010*, Paris, France

[10] Antoniol, G., Berenbach, B., Egyed, A., Ferguson, A., Maletic, S. and Zisman, A., "Centre of Excellence of traceability technical report (coet-gct-06-01-0.9)- problem statements and grand challenges", Tech. Rep COET-GCT-06-01-0.9. September 06

[11] Cho, H., Lee, K. and Kang, K.C, , "Feature Relation and Dependency Management: An Aspect-Oriented Approach", In *12th International Software Product Line Conference (SPLC'08),* 2008, pp.3-11.

[12] Filho, R.S.S, Weiss, M., Esfandairi, B. and Andrea, V.D., "Managing Feature interactions by documenting and enforcing dependencies in software product lines", Feature Interactions in Software and Communication Systems IX (ICFI'07), Grenoble, France.

[13] Ferber, H., Haag, J. and Savolainen, J., "Feature Interaction and dependencies: Modeling Features for Reengineering a Legacy Product Line", in *2nd International Software Product Line Conference (SPLC'02)*, San Diego, USA, pp. 37-60.

[14] Ivkovic, I. and Kontogiannis, K., "Using Formal Concept Analysis to Establish Model Dependencies", In *International Conference on Information Technology: Coding and Computing (ITCC'05)*, Las Vegas, USA

[15] Vierhauser, M. Dhungana,D., Heider,W. Rabiser,R. and Egyed, A., Tool support for Incremental Consistency Checking on Variability Models. In *VaMos 2010*, Linz, Austia.

[16] Colyer, A., Rashid, A., and Blair, G., "On the separation of concerns in program families", 2004.