Resolving Feature Dependency Implementations Inconsistencies during Product Derivation

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 the actual implementation over time, resulting in inconsistency during product derivation. We describe our initial results when working with feature dependency implementations and the related inconsistencies in actual code. Our aim is to improve consistency checking during product derivation. We have provided tool support for maintaining consistency between feature dependency implementations on both model and code levels in a product line. The tool chain supports the consistency checking on both the domain engineering and the application levels between actual code and models. We report our experience of managing feature dependency consistency in the context of an existing scientific calculator product line.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Management, Performance, Experimentation, Languages, Verification.

Keywords

Software product lines, aspect-oriented product line, AspectJ programming, feature implementation dependencies, variability models, product derivation, consistency checking, tool support.

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]. SPL engineering consists of two main interlaced activities, domain engineering and application engineering. During domain engineering, the commonalities and the variation points of the product family are analyzed and maintained. Variability points indicate where there will be variations in the same product family. During application engineering the variation points identified during domain engineering are realized based on a given configuration and a concrete product is derived (Product derivation). One major difficulty with SPL engineering is to deal with thousands of variation points in an industrial size product line [25]. These variation points need special attention as they add complexity during product configuration (PC) and product derivation (PD).

Variability management can greatly impact the complexity that is involved when producing a new product from existing product line assets [3]. The products of a SPL family differ by the features they include in Feature-Oriented Domain Analysis (FODA) [4,5]. SPL should have the capability to allow configuration of features or addition of individual features to the system. 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. Software product line models are inherently complex in nature due to embedded variability. Maintaining the consistency of product line artefacts which is necessary for derivation of correct products is a challenge for product line engineers in practice.

Aspect-oriented programming (AOP) [6] is a paradigm which allows developers to capture cross cutting structure and cross cutting concerns in a modular fashion. It helps to modularize feature implementations in source code. The common features are implemented in a base structure and variable features are implemented as aspects. During aspect oriented product line engineering (AOPLE) [17] product derivation, an aspect weaver creates a product by weaving variable feature implementation (aspect) into the base structure . Several approaches [7,8,9] using AOP have been conducted to implement cross cutting features in a modular fashion using separated components called aspects.

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 [10]. The problem is caused due to the fact that feature dependencies are embedded into feature implementations, resulting in tangled code issues. In feature implementations it is possible that feature dependencies are, 1) not correctly implemented, 2) missing, 3) wrongly implemented. Such issues can cause problems during product derivation when an aspect weaver creates a product by weaving aspects into the base modular structure.

In this paper, we provide a tool suite based on the Eclipse Modeling Framework (EMF) [11] and Epsilon [12] languages to 1) automatically extract feature implementation dependencies, 2)

synchronize feature implementations (code base containing Java classes and aspects) with a pre-existing implementation model (abstract representation of feature implementation), 3) check for inconsistencies between a feature implementation model and a code base, 4) check for inconsistencies in a configured implementation model, and 5) interactively resolve the inconsistencies. This research work is built on top of an existing example case study of a scientific calculator product line (Scicalc-PL) [10,13].

The remainder of the paper is organized as follows, Section 2 provides background to the problem, Section 3 discusses the type of inconsistencies identified, Section 4 discusses our approach to solving the feature dependency inconsistencies identified in Section 2, Section 5 is discussion, Section 6 summarizes related work and Section 7 concludes the paper.

2. BACKGROUND

Variability management is a key to success for any SPL. Many SPL research approaches focus on single development artefacts consisting of isolated models (e.g. feature-oriented, component-based product derivation etc.) [14,15]. While viewing SPL as a collection of artefacts has many advantages, there are disadvantages as well including the problem of describing cross cutting features, mapping cross cutting features to architectural design and then implementing them in the source code. In order to exploit the real benefits of a product line we need to connect these isolated artefacts (i.e., models, implementation code, and documentation).

The approach represented in this paper is illustrated using the example calculator product line (Scicalc-PL) artefacts presented in [10,13]. In this paper we are focusing on the implementation model (AML model) and the implementation code base artefacts of the Scicalc-PL. The AML model is an abstract view on the actual implementation consisting of packages, Java classes, Aspects and dependency relationships among them. The dependencies between the cross cutting features are implemented as Aspects in the source code of Scicalc-PL. The process of how the variabilities and dependencies are analyzed and utilized for structuring the product line implementation is discussed in [13].

The work in [13] focuses on two different kinds of dependency aspects which implement functional feature dependencies at run-time, namely Modification and Activation Dependencies [16]. For the clarification of the reader we discuss each briefly with an example *dependency* aspect from feature F1 to feature F2. *Modification dependency* from F1 to F2 implies that functional behavior related to F1 can be divided into two parts: the functional core and the interaction. The functional core represents the main functionality of the feature F1. The interaction part represents the modification behavior from F1 to F2. We can implement the interaction part (i.e., the modification dependency) of functional feature F1 using an AspectJ aspect (for more detail see [13]). *Activation dependencies* affect the activation of functional features. Activation dependencies can be implemented as separate generic aspects. Activation dependencies aspects can be classified into four different categories namely, 1) *Excluded Activation dependency aspects:* aspects that implement the functionality between functional features describing that one function excludes the functionality of the other, 2) *Required Activation dependency aspects*: aspects that implement the functionality that during execution the functionality of one functional feature can only be activated if the functionality of the other feature is active (in other words functionality of one requires the other), 3) *Sequential Activation dependency aspects*: aspects that implement the requirement that both the features must be active sequentially, 4) *Concurrent Activation Dependency aspects*: aspects that implements the requirement that features must be active together concurrently.

In Scicalc-PL the aspectual 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.

The focus of this work is to 1) automatically generate and visualize cross cutting feature dependencies in an existing implementation model (AML model), 2) check for consistency of model-to-code dependency aspects and interactively resolve any inconsistency in between, 3) check the consistency of a given configured AML model dependency aspects with respect to the corresponding implementation code base dependency aspects.

3. INCONSISTENCY BETWEEN DEPENDENCY ASPECTS DURING PRODUCT DERIVATION

In order to identify potential challenges for managing dependency aspects consistency, we analyzed the example Scicalc-PL model-based product derivation. The inconsistency scenarios defined in this section are in the context of research work conducted in [13].

We believe that the following inconsistency scenarios have to be dealt with in order to manage consistency in any AOPLE [17]. These are not the only type of inconsistency scenarios that can occur but for this particular stage of our research we have considered only the following inconsistency scenarios with scientific calculator example (Fig.1). As a running example we will demonstrate the inconsistency scenarios using dependency aspect examples taken from Scicalc-PL implementation model (Fig.2).

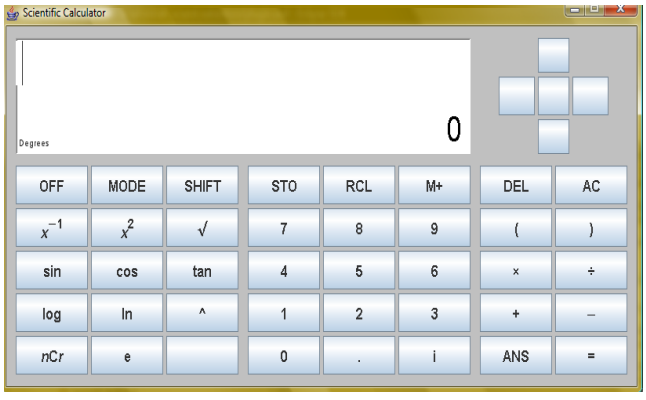

Figure 1. Calculator Application [10]

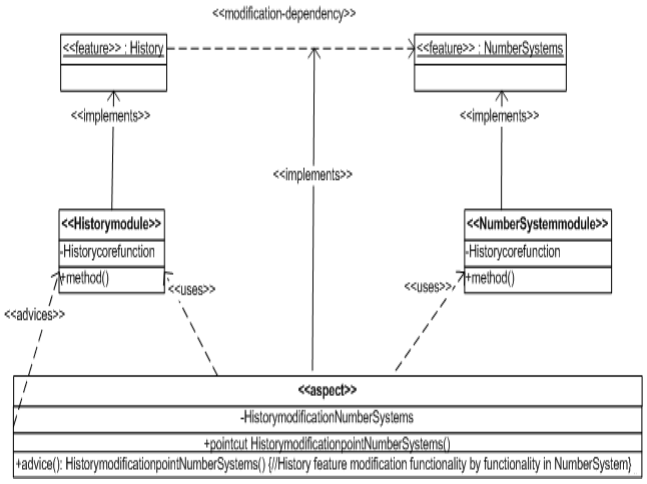

Figure 2. Modification dependency aspect (example from Scicalc-PL [13])

In Figure 2, modification dependency is implemented by an aspect. <<Feature>> *NumberSystems* modifies the functionality of <<Feature>> *History* during feature interaction. <<Historymodule>> and <<NumberSystemmodule>> implement the functionality of the History and NumberSystems features respectively. Both the modules consist of core functionality and interaction functionality.

The Aspect *HistoryModificationNumberSystem* uses the methods in <<Historymodule>> and <<NumberSystemmodule>>. The following sections shall describe not only modification dependency inconsistency but also the other types of dependency aspects discussed in Section 2.

3.1 Dependency Aspects Missing

During our analysis of Scicalc-PL model-based product derivation, we observed that inconsistency occurs when feature dependencies are 1) not implemented in source code, 2) not included and synchronized with the implementation model (abstract view on code), 3) not configured during product configuration. An inconsistency can occur when for instance, 1) the Aspect *HistorymodificationNumberSystems* is not implemented in the scicalc-PL code, 2) if it is not implemented in AML model on domain engineering, 3) not included in configured implementation model during product derivation. This leads to an inconsistency which won't allow the features *History* and *NumberSystems* to work in the final executable product. The same applies to other types of activation dependencies briefly discussed in Section 2, as all the other types of feature dependencies are implemented using aspects.

3.2 Implementing the Wrong Feature Dependency Aspects

An inconsistency can occur when dependency aspects are implementing the wrong feature dependencies. For instance, in the above mentioned example if the aspect *HistorymodificationNumberSystems* advices the method in <<NumberSystemsmodule>> and not the method in <<HistoryModule>>.

Another example scenario can be if the feature *History* has *Excluded Activation dependency* with the feature *NumberSystems*(*History—Excludes-Activation-dependency→NumberSystems*).Which means that the *History* functionality should be excluded when *NumberSystems* feature is turned on in the final product. If the situation is like *NumberSystems—Excludes-Activation-dependency→History*, then it leads to inconsistency in the final executable product. The mentioned inconsistency situation needs to be resolved during product derivation Otherwise the final product may not be robust and fully functional. This applies to other types of feature dependency aspects as discussed in Section 2.

3.3 Partially Implemented Feature Dependency Aspects

We analyzed Scicalc-PL and found that partially implemented feature dependencies also contribute to raising inconsistency during product derivation. An example scenario can be when the feature *History* has a modification dependency with the feature *NumberSystem* in the interaction part of the History Class. 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. During model-based product derivation, it is possible that such modification is not implemented at all or not included in the implementation model (AML model). In both cases this leads to an inconsistency during product derivation.

3.4 Order not Maintained or Implemented in Feature Dependency Aspects

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 [10]. 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. The implementation model must also implement this functionality so that when the implementation model is configured for a particular product this feature dependency gets included. This feature dependency aspect order inconsistency may or may not apply to other types of feature dependency aspects.

3.5 Inclusion of Redundant Feature Dependency Aspects

Introduction of redundant feature dependency aspects in the final list of components/configured implementation model may decrease the efficiency of product derivation and can increase the product derivation time. Implementation of redundant feature dependency aspects may cause inconsistencies in large scale product lines where there are thousands of cross cutting variation points.

4. ASPECTUAL CONSISTENCY MANAGEMENT APPROACH

In the previous section, we identified some of the many inconsistency scenarios which 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.

Figure 3 represents how our work is situated in the context of the overall Scicalc-PL case study [10]. We developed a plug-in tool suite chain along with the graphical representation of models for checking inconsistencies in feature dependency aspects. We have used the following frameworks and languages for developing our tool suite

- The EMF incremental plug-in development environment for tool suite plug-in development
- Eugenia , the Epsilon framework graphical language for developing graphical editor for Scicalc-PL model artefacts.
- Epsilon framework validation language (EVL) for applying constraints on models.
- EMF Compare language [18] to compare models.

The following sections will discuss our approach in detail.

4.1 Extracting Feature Dependency Relationships from Implementations

During analysis of Scicalc-PL, we identified certain inconsistencies (addressed in Section 2) that can occur while deriving a product if the implementation model (**A**) is not synchronized with actual implementation code base (**B**). See Figure 3.

For this purpose we developed two EMF based plug-ins. See (**1**) in Figure 3. **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 [19]. 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 (1b)** traverses the actual AspectJ AST relationship map of Scicalc-PL implementation to find out relationships between the Java classes and aspects. The plug-in (1b) than automatically generates the implemented feature dependency relationships in the already generated implementation model (AML model) using the **code2aml** plug-in. Both the plug-ins **(1)** are resource change sensitive. It means that whenever the Scicalc-PL AspectJ project changes (addition/deletion/update of Java classes and aspects implementing feature dependencies), both plug-ins get activated in the background.

4.2 Managing Feature Dependency Aspects

During analysis of Scicalc-PL artefacts, we found that there is a need of maintaining feature dependency aspect relationships. It is because feature dependency implementation relationships in the implementation need to be synchronized with the implementation

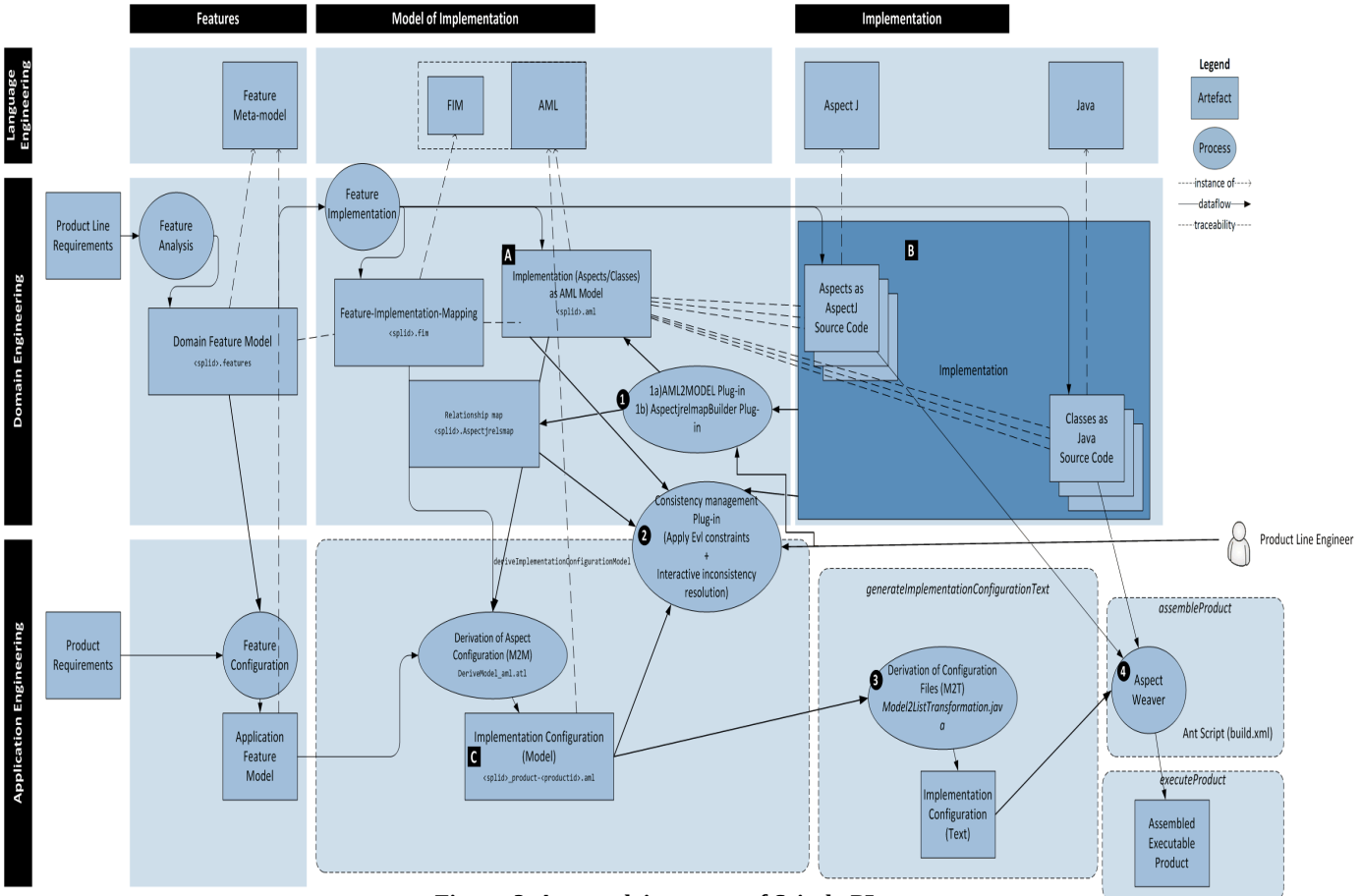

Figure 3. Approach in context of Scicalc-PL

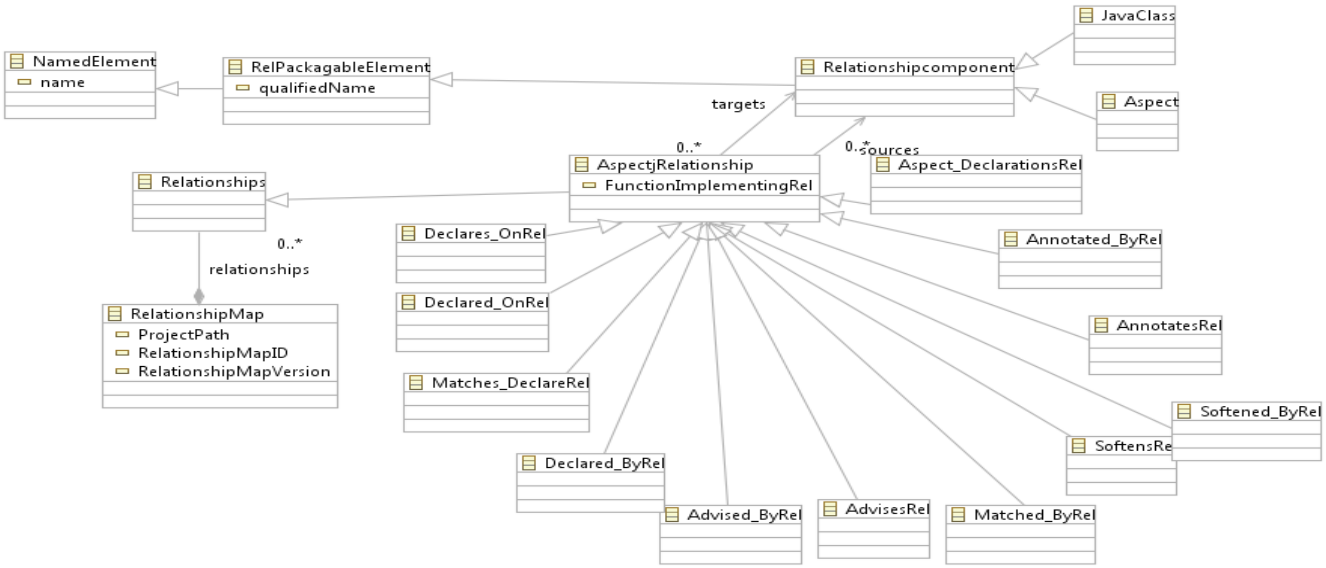

Figure 4. AspectJ relationship map meta-model

model at any development stage. To solve the mentioned challenge of maintaining feature dependency relationships, we established **AspectJrelMap** meta-model represented in Figure 4. 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-models 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.

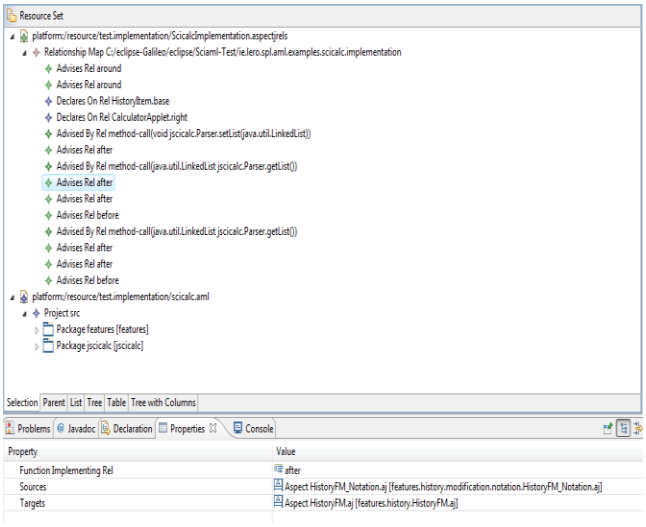

Figure 5. Example instance of Aspectjrelmap meta-model

Figure 5 presents an example initiated **AspectjrelMap** meta-model in the EMF model editor. It captures the sources and targets along with the function implementing relationship.

4.3 Visualizing Feature Dependencies Implementation Relationships

The graphical editor is generated using the Eugenia graphical language. Visual representation of feature dependency aspects helps us to visually identify inconsistencies in the implementation. It helps us to detect inconsistent evolutionary changes and resolve them interactively in the product line. It also helps us to identify and trace inconsistency (e.g., missing aspects, Java classes, dependency sources and targets) in the configured implementation model (**C**). Visualizing and resolving inconsistencies will be discussed in the following sections.

4.4 Applying Constraints on Feature Dependency Aspects

We have used EVL constraints to evaluate consistency during the domain engineering and application engineering phases (Figure 2). During domain engineering, when process (**1**) is executed we obtain an implementation model, which acts as a model artefact for the implementation. During domain engineering, after time **T** evolutionary changes result in a new version of the implementation model (**A**). We use EMF Compare language to identify potential changes in the new version of the implementation model with respect to the old implementation model **(A)**. EVL constraints **(2)** are then applied to generated the diff model. Based on the failed constraints, the error markers are generated in the new version of the implementation model **(A)**. During application engineering, the product line engineer wants to find out if the given configuration is consistent and includes all the required feature dependencies. In order to check the consistency of the configured implementation model, process **(2)** is again executed and inconsistencies are marked with error markers in the editor.

We describe our consistency rules into two levels, namely completeness constraints and dependency implementation constraints.

4.4.1 Completeness constraints

These consistency checking rules check for the completeness of the generated implementation model (AML model). After the introduction of evolutionary changes, the new version of the implementation model is checked for completeness consistency. Table 1 shows a few of the completeness constraint types applied on the generated AML model. These constraints also apply to the situation when a configured implementation model needs to be checked for completeness.

4.4.2 Dependency implementation constraints

These consistency checking rules are applicable to generated feature dependency relationships. These constraints check for generated dependency relationships attributes like name, sources

and targets of the aspect implementing feature dependency. Table 1 shows a few of the dependency implementation constraints.

Completeness constraints
Name and qualified name of model element (Java class/Aspect/ Package/Project) defined in extracted implementation model
Evolutionary change (addition/deletion/update) implemented in extracted implementation model
Dependency implementation constraints
Name and qualified name of aspect implementing the feature dependency in implementation synchronized with model element representing the aspect in extracted implementation model
New dependency implementation in actual implementation is implemented in extracted implementation model
Function implementing feature dependency is present in implementation model (aspect in model) and in synchronous with actual implementation (aspect actual implementation)
Sources and targets of aspects implementing feature dependency is in synchronous with sources and targets in extracted implementation model

Table 1. Completeness and dependency implementation constraints

4.5 Feedback to Product Engineer during Product Derivation

After applying the constraints using EVL the product engineer gets feedback in the form of error markers. These error markers help the product engineer to identify the potential inconsistencies to be resolved during product derivation. The inconsistency resolution is dealt with by suggesting the fixes in the "Quick fixes" section. The resolution choices are provided purely on the basis of implemented feature dependency aspects in actual implementation. Fig. 6 represents a scenario where target of an aspect is changed and on the basis of the pre-existing implementation model we provide choices (not shown in the Fig. 6) to the product engineer to resolve the inconsistency. Figure 6 **(1)** represents the Eugenia based graphical editor and **(2)** represents the error view.

4.6 Interactive Resolution of Feature Dependency Implementation Inconsistencies

The graphical editor developed in Eugenia is helpful in interactively resolving the inconsistencies. For example when an inconsistency arises, an error marker is generated on the respective Aspects, Java Classes or Packages. Double clicking the error detail in error view of the editor takes the product engineer to the problematic graphical element (Package, Java Class or Aspect).

5. DISCUSSION

In this paper we have presented our initial results obtained when working with feature dependency implementations in the context of product derivation in a product line. We used a case study product line to analyse the overall problem of inconsistency and to evaluate our approach. In this paper, we introduced our experimental approach for extracting and maintaining feature

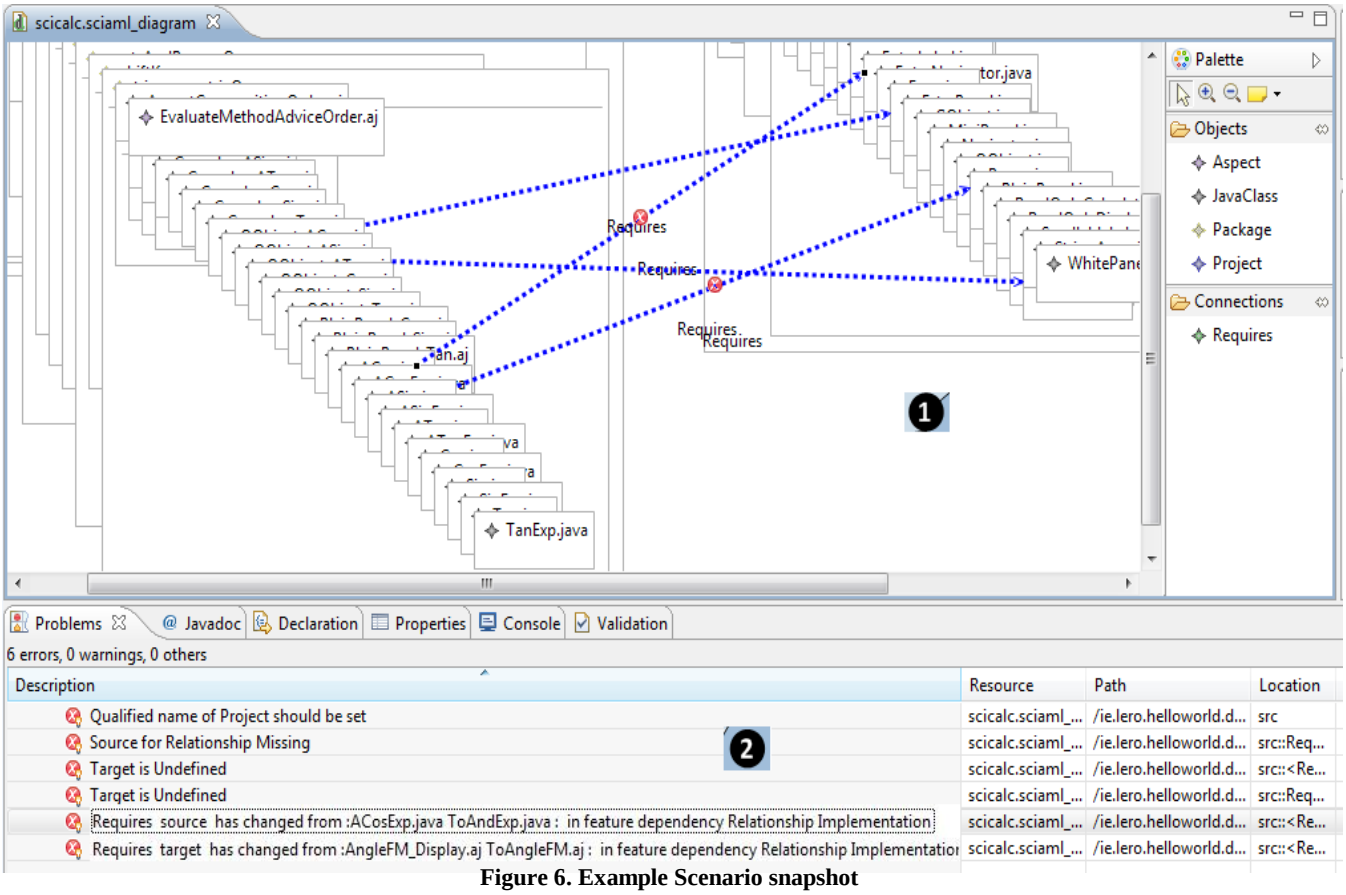

Figure 6. Example Scenario snapshot

dependency implementations.

We began by developing an incremental plug-in which actually reverse engineers the existing implementation and generates an implementation model (AML model). In order to generate the feature dependency implementation relationships, we obtain the AspectJ abstract syntax tree (AST) and work with the relationship map to find out relationships between implementation elements (e.g., Java Classes, Aspects). This information is then used to generate the relationships between extracted elements of the implementation model. We have also proposed a meta-model for capturing the AspectJ project relationship map information. The meta-model also acts as a traceability model between the generated implementation model and the actual code in the way that it links the relationship map to the AspectJ project implementation. In order to synchronize the generated implementation model and actual implementation, we use EVL constraints which we apply on the generated model for checking completeness and dependency implementation consistency.. During product derivation when product engineer wants to check if the configured implementation model (AML model) contains all necessary feature dependency implementations, we input the configured model along with implementation model representing the actual implementation to **(2)** (Fig.3). As an output the product engineer may or may not get a list of errors which he can work on to resolve them interactively.

To facilitate the product engineer to identify inconsistencies we developed a graphical editor. The inconsistencies are shown as graphical markers in the Eugenia based editor which can help the product engineer to visually see the inconsistencies. In order to resolve the inconsistencies, we extended the quick fix capability of the EMF validation framework using EVL. The developed prototype tool suite has some limitations. For this particular stage of the research work, we are assuming that all the feature dependencies are present in the implementation. When a new dependency is generated in the implementation model it represents the feature dependency in the feature model in domain engineering (Fig 3). The actual Scicalc-PL implementation consists of 348 different relationships. We obtained this information by analyzing the abstract syntax tree maintained by AspectJ development environment for Scicalc-PL implementation project. We obtained all the relationships and generated them in the implementation model successfully. The feature dependencies in Scicalc-PL were implemented with four main AspectJ programming functions (i.e., Advises, Advised by, Declared on and Aspect declarations). During application engineering, we are not taking into consideration how the configuration process is performed, rather we are checking consistency with respect to a given product configuration/AML model. We are currently completing our research prototype in order to include the feature model so that we can actually check the inconsistencies from domain engineering to the application engineering process.

6. RELATED WORK

Automated tool support is highly desirable for managing the complexity and variability inherent in software product lines.

Work by Lienhard et al.[20] analyzes the problem of runtime dependencies between features in an object-oriented system. It provides the detection strategy based on meta-models which capture the references. It also provides a visualization of feature runtime dependencies.

Kothari et al. [21] proposed an approach to system comprehension that considers features as the primary unit of analysis. The work provides a mechanism to define a relationship between features based on comparing the feature implementation.

Work in [22] presents a technique for semantic conflict detection between aspects at shared join points. The approach is based on abstracting the behavior of advice to a resource-operation model, and detecting conflicts patterns within the sequence of operations that various aspects apply to each resource.

Recent work by Vierhauser et al. [23] 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.

Our approach differs from the above mentioned approaches in that we are allowing the product engineer to interact and resolve the inconsistencies. We also provide tool support to identify inconsistencies and interactively resolve them during domain engineering (maintaining and synchronizing implementation model and actual implementation) and in application engineering (consistency checking the configured implementation model with respect to actual implementation).

7. CONCLUSION

Our primary goal is to make product derivation less error prone and more efficient. To achieve efficient product derivation we need to manage inconsistencies at all development stages in SPL particularly during product derivation. In this paper we have identified some of the different types of inconsistencies affecting feature dependency implementation that can cause product derivation to be inefficient and error prone. The base work for this research is described and elaborated in [10,13]. To support the product engineers in handling inconsistencies in large SPLs we have provided a prototype tool suite and an approach for managing feature dependency implementation inconsistencies. The tool suite has the capability to 1) extract the implementation model and generate dependencies from an actual implementation, 2) facilitate the product engineer to synchronize the implementation and the implementation model by applying EVL constraints, 3) detect and resolve the inconsistencies in the given configured implementation model, 4) provide the visual identification and interactive resolution of inconsistencies in the implementation model representing actual implementation and the configured implementation model for a particular product. In future we are planning to improve our approach by identifying and implementing the remaining feature dependency inconsistency scenarios. We are also planning to include the remaining Scicalc-PL modeled artefacts shown in Fig 3 (i.e., Feature and FIM models) in our approach. It is also planned to improve the inconsistency checking by applying incremental inconsistency checking [24] and better visualization to resolve inconsistencies during product derivation.

8. ACKNOWLEDGEMENT

This work was supported, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero- the Irish Software Engineering Research Centre (www.lero.ie). I would also like to acknowledge Dr.Norah Power and Dr.Goetz Botterweck for their feedback and support while preparing this research work.

9. 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] Sinnema, M. and Deelstra, S. 2007. Classifying variability modeling techniques. *Inf. Softw. Technol.* 49, 7 (Jul. 2007), 717-739. DOI= http://dx.doi.org/10.1016/j.infsof.2006.08.001

[4] Kang, K. C., Lee, K., Lee, J. and Kim, S. 2002 *Feature Oriented Product Line Software Engineering: Principles and Guidelines*. Gordon Breach Science Publishers, City, 2002.

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

[6] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C. and Mendhekar, A. *Aspect-Oriented Programming* 1997

[7] Godil, I. and Jacobsen, H. 2005. Horizontal decomposition of Prevayler. In *Proceedings of the CASCON 2005* Conference. IBM Press, 83-100. 2005

[8] Kaestner, C., Apel, S., and Batory, D. 2007. A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th international Software Product Line Conference (SPLC)* (September 10 - 14, 2007). Washington, DC, 223-232. DOI= http://dx.doi.org/10.1109/SPLC.2007.5

[9] Liu, J., Batory, D., and Lengauer, C. 2006. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 112-121. DOI= http://doi.acm.org/10.1145/1134285.1134303

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

[11] Eclipse Modeling Project, website: www.eclipse.org

[12] Epsilon Framework, 2009 website: http://www.eclipse.org/gmt/epsilon/

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

[14] Chae, W. and Blume, M. 2009. Language support for feature-oriented product line engineering. In *Proceedings of the First international Workshop on Feature-Oriented Software Development* (Denver, Colorado, October 06 - 06, 2009). S. Apel, W. R. Cook, K. Czarnecki, C. Kastner, N. Loughran, and O. Nierstrasz, Eds. FOSD '09. ACM, New York, NY, 3-10.

DOI= http://doi.acm.org/10.1145/1629716.1629720

[15] Atkinson, C., Bayer, J., and Muthig, D. 2000. Component-based product line development: the KobrA approach. In *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions* (Denver, Colorado, United States). P. Donohoe, Ed. Kluwer Academic Publishers, Norwell, MA, 289-309.

[16] Lee, K. and Kang, K. C. *Feature dependency analysis for product line component design*. Springer, City, 2004

[17] Griss, M. L. 2000. Implementing product-line features by composing aspects. In *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions* (Denver, Colorado, United States). P. Donohoe, Ed. Kluwer Academic Publishers, Norwell, MA, 271-288.

[18] Eclipse Modeling Framework technology Compare project, website: http://www.eclipse.org/modeling/emft/?project=compare#compare

[19] AspectJ project, website: http://www.eclipse.org/aspectj/

[20] Lienhard, A., Greevy, O., and Nierstrasz, O. 2007. Tracking Objects to Detect Feature Dependencies. In *Proceedings of the 15th IEEE international Conference on Program Comprehension* (June 26 - 29, 2007). ICPC. IEEE Computer Society, Washington, DC, 59-68. DOI= http://dx.doi.org/10.1109/ICPC.2007.38

[21] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A. 2006. On Computing the Canonical Features of Software Systems. In *Proceedings of the 13th Working Conference on Reverse Engineering* (October 23 - 27, 2006). WCRE. IEEE Computer Society, Washington, DC, 93-102. DOI= http://dx.doi.org/10.1109/WCRE.2006.39

[22] Durr, P., Bergmans, L. and Aksit, M. 2006 Reasoning About Semantic Conflicts Between Aspects. In *Aspects, Dependencies and Interactions Workshop* (held at ECOOP 2006).

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

[24] Egyed, A. 2006. Instant Consistency Checking for the UML. In *28th International Conference on Software Engineering (ICSE06)*, Shanghai, China

[25] Botterweck, G., Thiel, S., Nestor, D., Abid, S. b., and Cawley, C. 2008. Visual Tool Support for Configuring and Understanding Software Product Lines. In *Proceedings of the 2008 12th International Software Product Line Conference* (September 08 - 12, 2008). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 77-86. DOI= http://dx.doi.org/10.1109/SPLC.2008.32