# Assessing Modularity of Feature Concern

Sagar Sunkle

Faculty of Computer Science
University of Magdeburg, Germany
sagar.sunkle@iti.cs.uni-magdeburg.de

***Abstract*—In this paper, we put forth five observations regarding the implementation of feature concern using contemporary modularity mechanisms. Based on these observations, we propose a concern-centric approach for the assessment of feature modularity that uses syntactic and semantic reduction functions that separate the feature concern from variants containing it. We propose how the modularity of separated feature concern may be assessed using such functions.**

## I. Introduction

Software Product Lines (SPLs) is a software development paradigm that enables to create set of products, also called variants, differentiated in terms of features. The definitions of the concept of feature often side with either the configuration or the transformation view of mapping [1] between the problem space, the distinct scope that represents domain specific abstractions, and the solution space, the space of all implementation-oriented abstractions that are instantiated to obtain variants of the software system under consideration. Accordingly, features are either defined as distinguishable characteristics of a system important to any stakeholder and representing functional or non-functional requirements, or as an optional or incremental units of functionality that encapsulate a design decision pertaining to fulfilling a specific functionality [2].

Previous attempts at assessing modularity of features predominantly take the transformation view of mapping and the corresponding definition of features as units of increment in functionality into consideration [3], [4]. These approaches consist of representing a problem (expression problem in [3] and stock information problem in [4]) in terms of different modularization mechanisms such as aspects in AspectJ and Caesar, hyperslices in Hyper/J, units in Jiazzi, traits in Scala, and layered refinements in AHEAD and providing solutions that encapsulate functionalities of the defined problems as features.

Lopez-Herrejon et al. define feature modularity in terms of feature definition and composition properties evident from the representation of the expression problem [3]. Feature representations are considered modular if they can be defined in terms of units that encapsulate program deltas of various granularities in a cohesive manner and offer the possibility of separate compilation. Feature composition is considered modular if the units that define features could be composed flexibly, in any order, with static type checking support, and are closed under composition. Mezini and Ostermann [4] define feature modularity in terms of ability of different modularization mechanisms to address structural mapping between the units representing features and the base abstractions in terms of static crosscutting (when no one-to-one relation between the two exists) and dynamic crosscutting (when it is necessary to specify of how features interact with the dynamic control flow of base program).

Both these approaches essentially examine the suitability of different modularization mechanisms in representing features against a set of pre-defined characteristics of feature modularity. In the following section, we present five observations pertaining to features as modular entities. Our objective is to distinguish modularity assessment of features from traditional metrics based modularity assessment techniques.

## II. Distinguishing Feature Modularity Assessment

In the following, we attempt to throw light on feature as a concern against features represented in terms of different modularization mechanism which we believe to be only one of the several important characteristics of feature concern. We make the following observations regarding feature as a concern and the modularity mechanisms that provide support for representation and composition of feature concern:

**Basic and special purpose concerns**- Features are essentially special purpose concerns as opposed to basic concerns such as classes and objects or functions. Measuring modularity of feature representations in a system is always with respect to the representations of the basic concern (E.g., w.r.t to classes and objects in feature-oriented decomposition of object-oriented systems and w.r.t functions in feature-oriented decomposition of functional systems [5]). Modularity capabilities of the basic concern representations and good or bad design choices in basic concern representations substantially influence modularity of feature representations. It would be interesting to check if feature modularity metrics can also measure the proportion of modularity offered by the basic concern representations.

**Multiple modularity mechanisms**- Features can be implemented in myriad ways [2], [3], [4]. When implemented using other modularity mechanisms, features become secondary citizens [6]. This introduces further indirection in the modularity of features, because apart from basic concern representation such as classes, one incorporates further modularity characteristics specific to the modularity mechanism used to represent features.

**Direct and indirect dependencies**- We believe that feature modularity assessment should include the domain-specific abstractions, the implementation-oriented abstractions as well as the mapping between them. This is because using features to create SPLs implies composition specification from the domain [6]. Features contain explicit dependencies amongst themselves implied in the feature inclusion and exclusion constraints. At the level of implementation, program entities belonging to features depend on each other implicitly for error-free compilation [2], [3], [6]. Further indirect dependencies are introduced by feature interactions, e.g., features that work in isolation may depend on glue code for proper behavior when composed together [2]. An ideal feature modularity assessment technique should be able to distinguish between these dependencies.

**Degrees of presence**- Since features are predominantly implemented atop basic concern representations, the implementation of a single feature is almost always scattered across many entities representing the basic concerns (across many classes or functions [5]). In other words, concern diffusion is inherent to features. Feature modularity assessment techniques should include not only traditional metrics such as design structure matrix [7] and concern metrics such

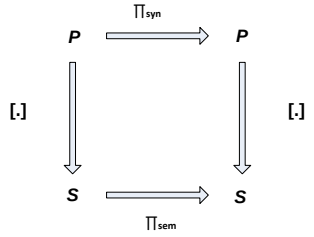


Fig. 1. Separation of concerns based on homomorphism and reduction

as concern diffusion over components but also metrics that measure degrees of presence [8].

**Evolvability/Changeability**- SPLs offer reuse in terms of features, but how to reuse features in evolving SPLs is not yet well established. Metrics for measuring changeability of modular entities may also be reinterpreted in terms of assessing how the features can withstand independent changes to the underlying program structure which is an important attribute of modularity specified by Parnas [9].

We now propose a concern-centric approach to assessing feature modularity as opposed to testing suitability of features modularization mechanisms based on problem domain specific characteristics. Our proposed approach for assessing feature modularity is based on a formal technique for measuring separation of concerns presented in [10].

## III. Concern-centric Feature Modularity Assessment

Ernst presents a technique to measure separation of concerns which has formal core [10] shown in Figure 1. Given a programming language *L*, set of all programs *P* in *L*, and all possible semantic values *S* in *L*, function $[.] : P \rightarrow S$ defines a function such that for every program p $\in$ *P* has semantic s $\in$ *S*.$\Pi_{syn} : P \rightarrow P$ defines a syntactic reduction function and $\Pi_{sem} : S \rightarrow S$ defines a semantic reduction function [10]. The concern diagram in Figure 1 is expected to commute, i.e., $[\Pi_{syn}\ (P)] = \Pi_{sem}\ ([p])$. We reinterpret the explanation of this diagram in terms of the feature concern.

The syntactic reduction function takes a program and removes the concern, in our case a feature, so that the program no longer contains the feature. The semantic reduction function maps semantic values of a program with a feature to a program without the feature. This would enable specifying degrees of separation of feature concern at both syntactic and semantic levels [10]. If the concern diagram is considered for all programs containing a concern, i.e., all variants containing a given feature, then the semantic reduction function must be defined such that $\Pi_{sem}$ will remove the meaning of the feature from these variants. With this arrangement, separation of feature concern may be defined in the following ways: (1) a $\Pi_{syn}$ would indicate a high degree of separation (DOS) if removing feature syntactically only removes code fragment(s), (2) a $\Pi_{syn}$ has low DOS if it results in altering the program behavior, (3) a $\Pi_{sem}$ has high DOS if removing a feature doesn't affect normal compilation of a variant, and (4) a $\Pi_{sem}$ has low DOS if its removal results in complex changes that affect composition of other features, as well as error-free compilation of other program entities and behavior of other features as well as the variant as a whole.

We believe that such a characterization of reduction functions for measuring DOS of features is a promising direction because, (1) it can enable us to distinguish the indirect effect of modularity characteristics of the basic concerns, (2) it can enable us to measure degrees of direct and indirect dependence, and (3) changeability can be incorporated in the $\Pi_{syn}$ and $\Pi_{sem}$ to measure DOS in evolving SPLs. We indicate in the following how this might be done.

## IV. Using Separation of Concern Measurement to Assess Feature Concern Modularity

Using the technique of measuring the degree of separation of the feature concern would involve obtaining proper syntactic and semantic reduction functions (which in their simplest form simply remove the concern syntactically and semantically respectively). In order to measure the indirect effect of incorporating modularity characteristic, it would be interesting to note how removing a feature affects the basic concern and the middle concern (such as aspect or traits when used to implement features in a system which is object-oriented to begin with). Similarly, removing a feature would reveal how strongly connected program entities in it are to programs entities in other features and base programs. To check this, some sort of type checking support would be required, either directly implemented for features themselves [2] or in terms of composition of other concern representations. Changeability characteristics of features can be investigated by tracking how the syntactic and semantic reduction functions themselves change when a changed feature is to be removed and the complexity of change in these functions could be used as indication of changeability characteristics of feature modularity.

## V. Conclusion

We believe that such a concern-centric modularity assessment technique could be better in terms of accurately determining feature concern modularity, because instead of based on problem domain specific characteristics as in [3], [4], it targets features as concern representations and thus provides broader implications.

## References

[1] K. Czarnecki, "Overview of generative software development," in *UPP*, ser. Lecture Notes in Computer Science, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds., vol. 3566. Springer, 2004, pp. 326–341.

[2] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, July/August 2009, guest Column.

[3] R. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating support for features in advanced modularization technologies," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, ser. Lecture Notes in Computer Science, A. P. Black, Ed., vol. 3586. Springer, 2005, pp. 169–194.

[4] M. Mezini and K. Ostermann, "Variability Management with Feature-Oriented Programming and Aspects," in *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM Press, 2004, pp. 127–136.

[5] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Feature (de)composition in functional programming," in *Software Composition*, ser. Lecture Notes in Computer Science, vol. 5634. Springer, 2009, pp. 9–26. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02655-3

[6] S. Sunkle, S. Günther, and G. Saake, "Representing and Composing First-class Features with FeatureJ," Department of Computer Science, Otto-von-Guericke University of Magdeburg, Germany, Tech. Rep. FIN-017-2009, Nov. 2009.

[7] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *ASE*. IEEE Computer Society, 2009, pp. 197–208.

[8] M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns," in *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*. Washington, DC, USA: IEEE Computer Society, 2007, p. 2.

[9] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *cacm*, vol. 15, no. 12, pp. 1053–8, Dec. 1972.

[10] E. Ernst, "Separation of concerns," in *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.