Synopsis

Models, modeling and model-transformations form the basis for a set of software development approaches that are known as model driven development (MDD). This approach primarily focuses on models as opposed to source code. They can be refined, evolved into a new version and can be used to generate executable code. The ultimate goal is to raise the level of abstraction and to develop complex software systems by manipulating models only. The manipulation of models is achieved by means of model transformations, known as refinement and refactoring. Refinements are performed during the software development life-cycle. Refactoring is performed during the maintenance of software. The transformation of a model into different model at the same relative abstraction level, without changing the modeled external behavior, is known as model-refactoring.

At the level of models, research on refactoring is still in its infancy. Very few tools provide integrated support for model-refactoring as for a lack of proper understanding of model-refactoring. Some formalism has been proposed to understand and explore model-refactoring. Van Der Straeten and D’Hondt used a forward chaining logic reasoning engine to support composite refactoring [1]. Biermann, Mens used graph transformation theory as an underlying foundation for specifying model-refactoring [2, 3]. Bruno Harbulot attempts to build a link between scientific programming and software engineering using aspect-oriented programming. Most of these approaches suggest for expressing model-refactoring in a declarative way. In the contrary Don Batory [4] sketches the idea of architectural meta-programming: programming and design is a computation, where programs are values and functions (a.k.a transformations) map program to program. In [5-6], the author explores from the architectural meta-programming perspective, the underlying connections among refactoring, program synthesis and MDD. These areas can converge and outline a theory that may unify them.
FOP aims at large-scale compositional programming and feature modularity in product-lines [8]. AOP focuses on crosscut modularity in complex software [6, 9-10]. Several recent studies have observed that both paradigms have limitations [7, 11-13]. The weakness of one maps roughly to the strength of the other. Hence, the two paradigms are not competitive and can profit from each other. Recent work has suggested that both paradigms be combined to exploit their potential [7, 11-12, 14].

Different approaches have been proposed to unify FOP and AOP by introducing aspectual-feature modules, aspect-refinements and aspectual-mixin layers [7]. The success of all these approaches decreases as the program size increases. Hence it is required to explore further their relationship at the design and language level.

In near future, the advanced Integrated Development Environments (IDEs) will apply refactoring for updates. The IDEs will use architectural-meta-expressions to perform these updates [4]. Finally refactoring will not be limited to the restructuring of the source code; they need to be applied to models and features as well [15, 16]. This will enhance the power of meta-programming approach in software design and maintenance.

On the basis of the above discussion we have done the following in order to enhance the Model driven development techniques:

1. Analyzing AOP at design and language level by proposing a model in AOP
2. Analyzing FOP at design and language level by proposing a model in FOP
3. Unification of AOP and FOP in model level
4. Proposing techniques for impure refactoring detection
5. Exploring modular refactoring by manipulation of models through model refactoring
6. Modular design of proposed SPL
7. Meta-programming, architectural meta-programming, meta-expressions
generations and conversion of meta-expressions to source programs through the
proposed SPL

The research work focuses on to analyze the AOP and FOP methodologies in design level
and language level in order to propose a module for AOP implementation and developing a
model which implements FOP with some existing approaches. The designed AOP module
will be later on integrated to the feature model not only to justify the unification of AOP
and FOP but also for justifying the module of AOP can be applied for those instances
where other traditional procedure or object oriented cannot cope up. The modular designing
approach is first implemented and later on refactored to obtain metaexpressions which
further leads to source code generation. Our metaprogramming approach uses conventional
programming languages along with the architectural metaprogramming to present a novel
methodology of metaprogramming. We further justified the significance of architectural
metaprogramming in our SPL.

Analyzing AOP at design and language level by proposing a model in AOP

Crosscutting concerns [17] are inevitable in software development. All concerns cannot be
encapsulated properly in a functional decomposition. As a result, they are scattered [18],
tangled [19] and replicated [24] which are known as code clones as they are duplicated in
sequence. These concerns are encapsulated in a special unit called aspect using Aspect-
Oriented Programming [17]. There are few typical situations like logging, authentication,
authorization etc. where crosscutting concerns generate. We have designed a login module
with access, log and message communication into a single unit of aspect through class
diagrams and state-chart diagrams, in model level. In language level we designed an AOP
code for message communication. This login module can be integrated to any software as a
security component.
Analyzing FOP at design and language level by proposing a model in FOP

Feature-Oriented Programming (FOP) [5] assists in designing and implementing the modular system product lines. It is a tool for development of SPL [20-21]. The key idea is of functionalities to arrange the design and implementation as a layered stack where layers consist of different programs. Thus, implemented layers can be reused in multiple programs. The layers are represented by features. A single module (class, package) may contain fragments from multiple features. To achieve such encapsulation of fragmented features many mechanisms are available like mixin layers [5, 22], lifters [23] etc. We have suggested the steps required for FOP handling using lifters and also designed an array handling model for demonstrating the steps. The proposed model will be helpful in designing a small-scale software product line. It has been observed that FOP can be implemented in language level as well as in architectural domain model. In language level we identified that lifter mechanism is lacking.

Unification of AOP and FOP in model level

Several works have been carried out to combine AOP and FOP both on design and language level. All studies [7, 11] have observed that both paradigms have limitations and the weaknesses of one map roughly to the strength of other. The three popular ways have been suggested for combining AOP and FOP by introducing AOP concepts in FOP are Multi Mixins, Aspectual Mixin Layers, and Aspectual Mixins [7]. We have proposed a model to embed the security module with the feature module, responsible for basic array operations through AML, which justifies the unification of AOP. We also demonstrated the role of Feature model, which is required for structural mapping in Software Product Line (SPL) and commonly used for expressing and binding variability.

Proposing techniques for impure refactoring detection

Several techniques and tools have been developed to assist refactoring either in the designing phase or in maintenance phase. But the current tool support is not adequate as the
refactoring tools are facing several challenges such as composition of refactoring, searching refactoring, planning refactoring sequences, computation of conflicts and dependencies of refactoring etc. These challenges would greatly benefit from a formal support provided by techniques which can detect impure refactoring. Early detection of impure refactoring can cope up with the mentioned challenges. Several approaches have been developed to detect impure refactoring including the graph search methods proposed by Shinpei and by Perez and Crespo [25-26]. On the other hand Emerson proposed a model which depicts the conventional refactoring process [27]. This model shows the sequence of the refactoring processes but has no role in early detection of impure refactoring. If Emerson’s model can be revised with Graph search methods then it can be used as a tool for early detection of impure refactoring. This motivated us to develop a similar tool, called Refactoring Flow Diagram (RFD) that not only helps in early detection of impure refactoring but also shows the flow of refactoring processes. This also enables one to synchronize and sequence the processes which give a holistic view of refactoring process.

**Exploring modular refactoring by manipulation of models through model refactoring**

We have refactored our proposed security model in design level and code level. In order to refactor the proposed security model we used UML [28] and also consulted the impact of modular refactoring on OCL [29]. As proposed by several researchers, in modular refactoring UML are preconditions and OCL/UML are post-conditions which are to be checked for behaviour preserving. We checked this for our proposed model and justified that to be behaviour preserving.

**Modular design of proposed SPL**

We applied the modular designing in a software product line (SPL) which provides some basic operations in array. This proposed SPL is known as Array Product Line (APL). Our proposed APL provides all the basic operations of array which is displayed through a GUI environment. This APL allows the user to select few features though a GUI, generates a
Meta-programming, architectural meta-programming, meta-expressions generations and conversion of meta-expressions to source programs through the proposed SPL

The meta-programming approach is a very powerful concept in MDD in context of software automation. As meta-programming approach of software development is a major contribution towards MDD so we introduce the concepts, terminology and implementation of meta-programming. Hence we design and implement a SPL through meta-programming. Our proposed elementary software product line, known as Array Product Line (APL), is a meta-programming approach to implement SPL. The main purpose is to illustrate architectural meta-programming using software product line. Architectural meta-programming [4] is one of the major and novel contributions towards Model Driven Development. We use architectural meta-programming by first mentioning the principles and then justify that our APL is designed on the basis of architectural meta-programming. Our proposed APL provides all the basic operations of array which is displayed through a GUI environment. We present a solution to APL problem using the GenVoca [30-31] design methodology to generate a meta-expression. We first generate different meta-expressions by selecting different features of the APL and then obtained different array operational programs from the generated meta-expressions. The GUI has been designed in such a way that it handles the APL for not only the generation of meta-expression but also the source code generation through meta-programming.

SPL involves automatic software generation. We claim that SPL has been represented in APL which is a very common data handling method for linear data structure. Using such a product line will enable any end user to process linear data structure (restricted to array
only) without the detail knowledge of working of array operations. Basic knowledge of array is required for processing data arranged in a linear form. We further claim that SPL is not only for reusability, customizability etc. for large scale software but also for small-scale software which is often required by different category of users ranging from students till researchers. This can be used as a tool for data processing. As a future work it is recommended that lifter concepts may be incorporated in high level language like FeatureC++. This will allow the easy and flexible composition of feature in SPL which helps in automated software generation which is a major need in MDD. We also recommended using graph transformation in RFD as a tool to make an early detection of impure refactoring. This will make refactoring popular in both code and model level.

**Bibliography**


