Chapter 9

Metaprogramming for Array Product Line

In this chapter the metaprogramming approach has been introduced in terms of concepts, terminology and implementation. The metaprogramming approach of software development is a major contribution towards MDD. Hence we design and implement a SPL through metaprogramming. The proposed elementary software product line is known as Array Product Line (APL) which is a metaprogramming approach to implement SPL. The main purpose of this is to illustrate architectural metaprogramming using software product line. Architectural metaprogramming is one of the major contributions towards MDD. Architectural metaprogramming is used by first mentioning the principles and then it is justified that the APL is designed on the basis of architectural metaprogramming. The proposed APL provides all the basic operations of array which are displayed through a GUI environment. A solution is presented to the APL problem using the GenVoca design methodology to generate a metaexpression. Different metaexpressions are generated by selecting different features of APL and then obtained different array operational programs from the generated metaexpressions. The GUI has been designed in such a way that it handles the APL not only for the generation of meta-expression but also for the source code generation through metaprogramming.

9.1 Introduction to Metaprogramming

Metaprogramming is a method of writing programs that represent and manipulate other programs or themselves (reflection). So, code generating programs are called metaprograms and the process of writing such programs is called metaprogramming.
Code reuse is a key feature which should be offered by programming languages in order to automate and standardize a process. The two different methods that can be adopted to achieve code reuse are composition languages [96] and meta-programming [102]. In the former approach, programmers can write fragments of code (classes in the case of Object oriented programming) which are not self-contained, but those which depend on other fragments. Such dependencies can be later resolved by combining fragments via composition operators, to obtain different behaviors. These operators form a composition language. Inheritance (single and multiple), mixins and lifters are all approaches that allow one to combine classes and hence they define a composition language in the sense that is mentioned above. The limitation of this approach is that the users are provided with a fixed set of composition mechanisms and they cannot define their own operations, as it happens, e.g., with function/method definitions.

In metaprogramming, programmers write (meta) code that can be used to generate code for solving particular instances of a generic problem. In the context of object-oriented languages, the most widely used metaprogramming facility is template metaprogramming, which is available in C++. The templates are parametric functions or classes and can be defined and later instantiated to obtain highly-optimized specialized versions. The instantiation mechanism requires the compiler to generate a temporary (specialized) source code, which is compiled along with the rest of the program [41].

Moreover, template specialization allows encoding of recursive computations that can be thought of as compile-time executions. This technique is very powerful, yet can be very difficult to understand since its syntax and idioms are more complicated compared to conventional programming. For the same reasons, maintaining and evolving code which exploits template metaprogramming is rather complex. Hence, in recent years the software product line approach has emerged as an innovative way to improve software productivity and quality.

9.1.1 Metaprogramming Terminologies

Generation of the source code for program is the key process of building any software. In order to enhance these processes, to ease the task of the programmer and to relieve
the programmer from all repetitive and conceptually redundant operations, one may focus on the software automation. This means manipulation of code must be automated as much as possible. Automate programming is by definition metaprogramming. Thus, metaprogramming the art of programming programs that read, transform, or write other programs, appears naturally in the chain of software development, where it plays an essential role, be it “only” under the form of compilers, interpreters, debuggers etc. [38]. However, metaprogramming is almost never thought to be integrated in the processes of development. Thus, gaining awareness of its role is precisely the first step towards progress in this domain. Hence, few terms of metaprogramming are introduced as proposed by Czarnecki [78]. They are the following.

- **Metaprogramming**: “writing programs that represent and manipulate other programs (e.g. compilers, program generators, interpreters) or themselves (reflection)"
- **Metaprograms**: “represent and manipulate other programs or themselves”
- **Metalanguage**: “the language in which metaprograms are written”
- **Reflection**: “when a programming language is able to be its own metalanguage then it is called reflection. There are two aspects of such manipulation: introspection and intercession”
- **Introspection**: “the ability of a program to observe and reason about its own state”
- **Intercession**: “the ability of a program to modify its own execution state or alter its own interpretation or meaning”
- **Metaobject**: “it represents methods, execution stacks, the processor, and nearly all elements of the language and its execution environment”

The above terminologies are helpful in order to handle the metaprogramming techniques. Though metaprogramming is achieved through different ways, but it is desired to present a very comprehensive way of metaprogramming which handles the SPL known as Array Product Line, in the next section.
9.1.2 Proposed Metaprogramming Approach

As a part of this innovative method to improve software productivity and quality, an attempt has been made to couple disciplined metaprogramming feature with a compositional language like C/C++/Java. In the present approach, metaexpressions from a set of features have been generated, which appear as tokens having specific semantics. In other way, it can be said that, few features can be composed in a proper order which yield an expression known as metaexpression. The metaexpression has further been treated to generate a source code, in a high level language like C/C++/Java.

The varying composition of features not only generates different expressions but they also suggest development of reusable product line assets. Many technologies, such as GenVoca [12], Template Metaprogramming [101] etc., have been proposed to develop reusable product line assets. Designing of Array Product Line (APL) based on GenVoca model is proposed which is less complicated than the existing ones like template metaprogramming etc. APL has been chosen in the present work as Lopez et al. [77] used Graph Product Line (GPL) to evaluate the product-line methodologies. They have justified GPL as a non trivial design problem. Since GPL is a non trivial problem so APL has been chosen as the problem which will be used for metaexpression generation. APL caters to linear data structure whereas GPL caters to non-linear data structure.

9.1.3 GenVoca Model

GenVoca is a model of product-lines that is based on step-wise refinement [71, 74, 76]. It says that programs are values. The following constants that represent programs with individual features has been considered.

\[
f \quad // \text{a program that implements feature } f \\
g \quad // \text{a program that implements feature } g
\]

A refinement is a function that takes a program as an input and produces a refined (or feature-augmented) program as an output.

\[
i(x) \quad // \text{adds feature ‘}i\text{’ to program } x
\]
j(x)  // adds feature ‘j’ to program x

It follows that a multi-featured application is an equation, and that different equations define a set of related applications, i.e., a *product-line*, such as the following.

\[
\text{a1} = i(f); \quad \text{// application a1 has features ‘i’ and ‘f’}
\]
\[
\text{a2} = j(g); \quad \text{// application a2 has features ‘j’ and ‘g’}
\]
\[
\text{a3} = i(j(f)); \quad \text{// application a3 has features ‘i’, ‘j’, and ‘f’}
\]

Thus one can determine features of an application by inspecting its corresponding equation.

### 9.2 Architectural Metaprogramming

A new concept has been developed in context of Model Driven Development which is known as architectural meta-programming [18]. It suggests that programming and design are computations, where programs are values and that, functions (a.k.a. *transformations*) map programs to programs. The author has further explored the underlying connections between program refactoring, feature-based and aspect-oriented software synthesis and model-driven development in context of architectural metaprogramming.

#### 9.2.1 Principles of Architectural Metaprogramming

The principles of architectural programming has now been framed which has been discussed by the author [18] and is based on the statement that “*programs are values and functions transform one program to another*”.

1. Values (i.e. programs) can be added or subtracted
2. Distributive Transformations are functions that distribute over + or -
3. Distributive Transformations of non-null values yield non-null values
4. If a value (i.e. program) is to be deleted, subtraction should be used
5. Distributive Transformations are functions and function composition is not commutative but associative
6. Expressions that are formed by adding, subtracting and transforming programs are architectural metaexpressions or metaexpressions
### 9.2.2 Application of the Principles in our Proposed APL

In order to justify the principles, tests are conducted on some code fragments of the proposed APL in which we intended to be generate the source code. The following different code segments (here it is treated as values) are considered which will be used to verify that it abides with the principles of metaprogramming or not.

<table>
<thead>
<tr>
<th>class array1 { int x[10]; void store( ); };</th>
<th>class array2 { void sort( ); };</th>
<th>class array1 { int x[10]; void store( ); void sort( ); };</th>
<th>class array3 { void search( ); };</th>
<th>class array1 { int x[10]; void store( ); void sort( ); void search( ); };</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value A</td>
<td>Value B</td>
<td>Value C = (A+B)</td>
<td>Value D</td>
<td>Value E = (C+D)</td>
</tr>
</tbody>
</table>

Few values have been listed which will be used to verify that to what extent APL follows the principles.

#### Principle I

In a traditional Object oriented program array2 can inherit the members of array1 (i.e. member data and member function). However, as per Principle 1 of the architectural metaprogramming we can say C = A + B. This activity in AOP is called *introduction*, which is an architectural metaprogramming.

The *Summation* (or simply “sum”) is disjoint set union [79] with the properties.

- Sum identity 0 is the *null program* or *null value*. For any program P: P + 0 = P
- Sum is commutative: G + P = P + G
- Sum is associative: (P+Q) + R = P + (Q+R)

#### Principle II

Values can be subtracted. If a program is formed by A + B and then if is subtracted (remove) B then it is A only i.e., (A + B) – B = A

The properties of Subtraction are the following.

- 0 is the identity: P – 0 = P and P – P = 0
- Subtraction is left associative: P – Q - R = ((P-Q) - R)
- Subtraction is not commutative: P - Q ≠ Q - P

<table>
<thead>
<tr>
<th>class array1 { int x[10]; void store( ); };</th>
<th>class array2 { void sort( ); };</th>
<th>class array1 { int x[10]; void store( ); void sort( ); };</th>
<th>class array3 { void search( ); };</th>
<th>class array1 { int x[10]; void store( ); void sort( ); void search( ); };</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value A</td>
<td>Value B</td>
<td>Value C = (A+B)</td>
<td>Value D</td>
<td>Value E = (C+D)</td>
</tr>
</tbody>
</table>
Principle III
A third operation is Distributive Transformation which represents several activities. The different activities which come under Distributive Transformations are the following.

a) **Rename Operation**
Renaming a variable in a program is a type of refactoring. As Distributive Transformation is a function (i.e. activity) so when used for renaming then it is represented as: \( \text{rename} (A, x, z) \).

The outcome of the function is that all variables named ‘x’ in program (value) A will be changed to ‘z’. The sample code segment is given below:

```java
class array1
{
    int z[10];
    void store();
}
```

b) **Distributive Transformations over + and -**
As per the name of Distributive Transformation, it distributes (like set theory distribution) over + and -. Say ‘f’ is the Distributive Transformation. So in the example

\[
\begin{align*}
    f(A + B) &= f(A) + f(B) \\
    f(C - B) &= f(C) - f(B)
\end{align*}
\]

The Distributive Transformation over + can also be applied by using the rename operation. The activity of Distributive Transformation can be representative using

\[
\text{rename}(A + B, x, z) = \text{rename}(A, x, z) + \text{rename}(B, x, z) = F + B \quad \text{[since rename (B, x, z) has no effect]}
\]

c) **Transformations of non-null values:**
Transformation of non-null values yield non-null values. Say,

\[
f(x) \neq 0 \quad \text{when} \ x \neq 0
\]
This implies that application of a Distributive Transformation to a non-null value will not null the value (i.e. will not delete program).

d) Distributive Transformations are functions and can be composed as functions.
   If f1 and f2 are two DTs then f1.f2 denotes their composition.
   
   \[ f1.f2 \neq f2.f1 \]
   Hence, Distributive Transformation is not commutative.
   In our example \[ C \cdot D \neq D \cdot C \]

The above expressions which are formed by adding, subtracting and transformation of programs are known as architectural metaexpressions or metaexpressions. Metaexpressions are very useful for automated software. This is the main objective of future software generation.

### 9.3 Metaexpression Generation in Proposed APL

The proposed SPL named as Array Product Line (APL) has been developed to give a new methodology for software automation. The reason behind the development of such software is to integrate architectural metaprogramming with conventional languages. The present APL provides some guidance towards the new methodology of product line development. As different approaches exist for product line development, GenVoca model [12] has been used to implement architectural metaprogramming. This further justifies AOP, FOP, MDD and refactoring are all related to each other from a metaprogramming point of view.

The proposed APL can deal with standard array operations like traversal, insertion, deletion, searching, sorting, etc. This is a product line because none of the applications have the same set of features. Moreover, the applications are modeled as sentences of a language. Table 9.1 shows the sentences where tokens are names of features.
Table 9.1: Tokens and metaexpressions

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>APL</td>
<td>Al.At.Dim.Opr</td>
</tr>
<tr>
<td>Al</td>
<td>C</td>
</tr>
<tr>
<td>At</td>
<td>integer</td>
</tr>
<tr>
<td>Dim</td>
<td>OneDim</td>
</tr>
<tr>
<td>Opr</td>
<td>Traversal</td>
</tr>
</tbody>
</table>

9.3.1 Semantics of the APL Features

The token Al defines the language of the source code. An array type (At) is determined by the elements it stores i.e. either integer or character or float elements. The dimension of an array is indicated by token Dim. The different operations that can be performed are traversal, insertion, deletion, sorting, searching, and multiplication over the arrays. Based on the array type, operations performed and language opted, a source code in the opted language will be generated from the meta-expression.

The format of meta-expressions is shown below.

APL = Al.At.Dim.Opr;

Here, APL comprises four items that are source code language (Al), Array type (At), Array dimension(Dim) and Array operations (Opr).

The expressions are for APL metaexpression which has its own grammar and semantics. Some of the probable meta-expressions are the following.

APL1 = C.Integer.TwoDim.Insertion.Traversal;
APL2 = Java.float.OneDim.Insertion;

The expression APL1 allows one to visit each element in a two dimensional array and to obtain a source code in C language. Similarly, APL2 is for insertion of integer elements in a one dimensional array with a source code generation in java.

This approach is known as Feature-Oriented Programming (FOP) where model of a product-line is algebra: constants represent base programs (e.g. OneDim Array) and
functions add features (Traversal, C language) to base programs. On the other hand, it is an architectural metaprogramming using summation property.

A fundamental characteristic of product-lines is that not all features are compatible. This implies that the selection of one feature may disable (or enable) the selection of others. As an example, the selection of one dimensional array will disable the multiplication option.

### 9.3.2 Development of Proposed APL

The following steps are required to design and develop the APL model.

1) Designing a GUI based environment (Figure 9.1)

![GUI representation of APL and Metaexpression](image)

**Figure 9.1:** GUI representation of APL and Metaexpression

2) Obtaining a meta-expression of APL features (Figure 9.1).

In order to obtain a specific metaexpression, say insertion in one dimensional array, then it will resemble like the metaexpression window in Figure 9.1.

Here AOP is introduced. It adds a new member to an existing class or adds a new class or package to the base program (e.g. in the base program One Dim. Array, added insertion feature has been added and number data type has been
introduced). The base program contains the declaration of the one/two dimensional array. Say \( F(x) \) is a feature which modifies the base program \( d_f \) by adding a function \( t_f \) (insertion option) and introducing \( n_f \) (Number option as elements of array). A general form of all FOP features \( F \) is the following.

\[
F(x) = d_f + t_f + n_f
\]

3) A program code (in the desired language) is to be synthesized by evaluation of this meta-expression.

As the selection of one feature may disable (or enable) the selection of others, the set of constraints that govern the APL features are now provided in Table 9.2.

<table>
<thead>
<tr>
<th>Preferred language for Source code</th>
<th>Operational Source Code</th>
<th>Required Dimension</th>
<th>Required Array Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, C++, Java</td>
<td>Insertion</td>
<td>One</td>
<td>Integer, Character, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Deletion</td>
<td>One</td>
<td>Integer, Character, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Traversal</td>
<td>One, Two</td>
<td>Integer, Character, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Sorting</td>
<td>One</td>
<td>Integer, Character, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Addition</td>
<td>Two</td>
<td>Integer, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Multiplication</td>
<td>Two</td>
<td>Integer, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Minimum value</td>
<td>One, Two</td>
<td>Integer, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Maximum value</td>
<td>One, Two</td>
<td>Integer, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Average value</td>
<td>One, Two</td>
<td>Integer, Float</td>
</tr>
<tr>
<td>C, C++, Java</td>
<td>Transpose</td>
<td>Two</td>
<td>Integer, Float</td>
</tr>
</tbody>
</table>

The above constraints decide the probable combinations. The major array related activities are taken care of in the APL.


### 9.3.3 GenVoca Model of APL

Using the concept of GenVoca (referred in Section 9.1.3) the GenVoca model of APL which is a set of constants and functions defined in Table 9.3 are now provided.

<table>
<thead>
<tr>
<th>Constants</th>
<th>Meaning</th>
<th>Functions or Operations</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C Language</td>
<td>Insert(x)</td>
<td>Insertion</td>
</tr>
<tr>
<td>C++</td>
<td>C++ Language</td>
<td>Delete(x)</td>
<td>Deletion</td>
</tr>
<tr>
<td>Java</td>
<td>Java Language</td>
<td>Traversal(x)</td>
<td>Traversing</td>
</tr>
<tr>
<td>OneDim</td>
<td>One Dimension</td>
<td>Sorting(x)</td>
<td>Sorting</td>
</tr>
<tr>
<td>TwoDim</td>
<td>Two Dimension</td>
<td>Searching(x)</td>
<td>Searching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addition(x,y)</td>
<td>Addition of two matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiplication(x,y)</td>
<td>Multiplying two matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transpose(x)</td>
<td>Transpose of a matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minimum(x)</td>
<td>Minimum value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maximum(x)</td>
<td>Maximum value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prog(x)</td>
<td>Main Program</td>
</tr>
</tbody>
</table>

Table 9.3 reflects that certain array operations need one/two parameters. The array operations are the function names (like traversal, insertion, multiplication etc.) that need a single array (one/two dimension) as input parameter. Traversal operation will have only one parameter whereas the array operation like multiplication will need two 2D arrays as parameters.

All the functions are distinct as per the name except Prog(x). Prog operation creates the objects that are required to represent an array and calls the required operations of the family member on this APL. Refinements cannot be composed in arbitrary orders. The legal compositions of refinements that are shown in Table 9.3 are defined by simple constraints called *design rules* that have been mentioned in [71]. In the present work the GUI specification tool translates a sentence in the grammar of Figure 9.2 (in addition to checking for illegal combinations of features) into an
equation. This translation is straightforward because features are in 1-to-1 correspondence with refinements. For example, an APL application “app” that implements traversal in a one dimensional array consisting integers with code generation for the expression in C language is the following equation.

\[
app = \text{Prog}(\text{C.Integer.Traversal(OneDim)})
\]

Similarly, the equation for multiplying two 2D arrays consisting integers with C++ language code generation is the following.

\[
app = \text{Prog}(\text{C++.Integer.Multiplication(TwoDim, TwoDim)})
\]

### 9.3.4 Conversion of Metaexpression to Source Code

The metaexpressions that are obtained can further be used for synthesizing source code. A model of a product-line is an algebra where constants represent base programs (e.g., OneDim), and functions add features (Traversal, Integer, etc.) to programs. Each domain has its own algebra, and different meta-expressions synthesize different programs of that domain (product-line). Moreover, the generation of program can be in any standard programming language like C/C++/Java as per user’s choice.

The design and development of the proposed APL software is carried out using in the following steps.

(i) Design user interface
(ii) Develop codes for the designed interface
(iii) Develop codes for activating/implementing the user interface
(iv) Generate the metaexpression
(v) Check the validity of the options chosen (present in the expression)
(vi) Fetch the appropriate coded text file
(vii) Generate the appropriate source code as per the metaexpression

The sample code generated by APL software for insertion and traversal in two dimensional array using C language is shown in Figure 9.2.
Array Product Line has been developed in order to design a software product line (SPL) using a metaprogramming technique. Architectural metaprogramming technique, which is a new approach of MDD has also been demonstrated along with its principle in the present APL. It has been demonstrated that APL uses AOP, FOP and refactoring in context of architectural metaprogramming. Automatic software generation has been achieved by selecting features which is involved in SPL. Hence, a GUI has been designed which enables to select the features for a particular type of array operations. On selection, a metaexpression has been generated. This is further used for source code generation. It can be said that SPL has been represented in APL which is a very common data handling method for linear data structure. This present work of generating
such SPL is not only for reusability, customizability etc. for large-scale software but also for small-scale software. Such small-scale software is often required by different category of users ranging from students to researchers. APL can also be used as a tool for data processing.