

Metaprogramming: an art of programming programs using Software Product Lines

Mahua Banerjee¹, Dr. Chiranjeev Kumar²

¹Department of Information Technology, Xavier Institute of Social Service, Ranchi, India

²Department of Computer Science and Engineering, Dhanbad, India

¹banerjee.mahua@rediffmail.com, ²k_chiranjeev@yahoo.co.uk

Abstract: Compositional languages and meta-programming are two approaches of code reusing. Both have its merits and demerits. We distilled the better of these two approaches by coupling disciplined meta-programming features with a compositional language like C++. In this paper we present a meta-programming approach to implement software product lines by designing an Array Product Line to implement different array operations by generating meta-expressions followed by respective C++ source code. A GUI has been designed to handle the APL for generation of meta-expression and metaprogramming.

Keywords: meta-expression, meta-programming, Array Product Line.

I. INTRODUCTION

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

In meta-programming, 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 programming template meta-programming is the most widely used meta-programming facility, as, e.g., in C++, where templates, which are parametric functions or classes, 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. Moreover, template specialization allows encoding 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 esoteric compared to conventional programming. For the same reasons, maintaining and evolving code which exploits template meta-programming is rather complex.

The aim of this paper is to distill the better of these two approaches, that is, to couple disciplined meta-programming features with a composition language, in the context of C++ like classes. More precisely, we propose a formal framework for extending a class-based language, equipped with a given class composition mechanism, to allow programmers to define their own derived composition operators. These definitions can exploit the full expressive power of the underlying computational language by software product lines. We present a meta-programming approach to implement software product lines. In recent years the software product line approach has emerged as a promising way to improve software productivity and quality. Many technologies, such as GenVoca, XVCL and Template Meta-programming, have been proposed to develop reusable product line assets. Lopez-Herrejon and Batory proposed the Graph Product Line (GPL) as a standard problem for evaluating product line technologies [1]. This paper presents a novel perspective in the field of software product lines, the Array Product Line. We present some of the basic concepts of the Array Product Line development. A fundamental idea in APL is using features to describe and differentiate programs within a family, where a feature is an increment in functionality. A particular program is specified by selecting a set of features. The selection of features generates a meta expression. Finally the generated meta expression leads to program code. We also summarized some of the possible benefits, risks and costs associated with this approach to software reuse of Product Lines, in general.

II. Metaprogramming Terminologies

Central to any process of building software is manipulation of the source code of programs. To enhance these processes, to ease the task of the programmer, is to relieve the programmer from all repetitive and conceptually redundant operations, so that one may focus



on the essence of programming, that is, on the problems that have never been solved yet. This means manipulation of code must be automated as much as possible and to 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. However, metaprogramming is almost never consciously integrated in the processes of development, and 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 Czarniecki [2]:

- 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. It is the ability of the program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession.
- Introspection: the ability of a program to observe and therefore 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.
- Metalevel architecture: an architecture where a metalevel provides information about selected system properties and makes the software self-aware while a base level includes the application logic.
- Partial evaluation: it is a technique to optimize the system when running some code multiple times on sets of input data in which one part is varying and another part is constant. In that case, it is useful to pre-evaluate the code with respect to the constant part of the data.

III. Metaprogramming in C++

A. Template metaprogramming

Template metaprogramming is a metaprogramming technique in which templates are used by a compiler to generate temporary source code which is merged by the compiler with the rest of the source code and then compiled. The use of templates can be thought of as compile time execution. This technique is supported by C++.

Template metaprogramming is much closer to functional programming than ordinary idiomatic C++ is. This is because variables are all immutable, and hence it is necessary to use recursion rather than iteration to process elements of a set. It allows the programmer to focus on architecture and delegate to the compiler the generation of any implementation required by client code. On the other hand template metaprogramming is cumbersome. With respect to C++, the syntax and idioms of template metaprogramming are esoteric compared to conventional C++ programming. So template metaprograms are very difficult to understand. Often it is very hard to make the intent of the code clear to a maintainer, since the natural meaning of the code being used is very different from the purpose to which it is being put. These major drawbacks of template metaprogramming can be removed by using Declarative Domain Specific Language in Software product lines.

IV. Software Product Lines and Model Driven Development

Declarative languages can be used to specify programs instead of using formal logic specifications. In order to automate program construction Software Product Lines (SPL) can be integrated with declarative languages. The basic idea in SPL is using features to describe and differentiate programs within a family, where feature is an increment in functionality [3]. Features are used in many disciplines for product specifications. Some web sites provide different options to the customers for selecting a product with certain configurations. Such pages are Declarative Domain Specific Languages. Product lines have been used by the manufacturing industry for a long time to reduce costs and increase productivity. However product line practice in the software industry is a relatively new concept. This new concept of Software product line development promises a step towards Model Driven Development. SPL can be a very powerful tool in MDD as SPL can be decomposed into three dimensions:



- (i) The first dimension is in regard of reuse assets of the product line: architecture, components and systems.
- (ii) The second dimension is in regard of the organizational views of product line: business, organization process and technology.
- (iii) The third and final dimension of product line is in the context of the life cycles of the product line assets: development, deployment and evolution.

These three dimensions of SPL make Model Driven Development (MDD) an emerging paradigm for software creation. It not only advocates the use of Domain Specific Languages (DSL) but also encourages the use of automation. An MDD model is written in DSL to capture the details of a slice of a program’s design. Several models are typically required to specify a program completely. Program synthesis is the process of transforming high level models into executables, which are also considered models [4].

There are many MDD technologies. OMG’s Model-Driven Architecture is most well known where models are defined in terms of UML and are manipulated by graph transformations [5]. A refinement in this approach has been done by integrating DDSL for a product line. A GUI environment is provided to display the different features of a SPL. A particular program is specified by selecting a set of features. The selection outputs a meta-expression. Evaluating the expression synthesizes the specified program. In a meta-expression constants are the base programs and function adds features to programs. Different meta-expressions synthesize different programs of that domain (product-line). This concept is explained in the next section by designing a SPL called Array Product Line (APL).

A. Array Product Line

Array Product Line (APL) is a family of classical array applications that was inspired by early works on Feature Oriented Programming (FOP) and Product Lines. According to Batory constants and functions of an FOP domain model is algebra (expression). Constant and functions (i.e. operators) are composed to define a domain of programs that can be synthesized. This can be implemented with many different technologies ranging from simple objects to sophisticated metaprograms and program transformation systems [6].

APL is a typical product line in which applications are distinguished by the set of features that they implement and no two applications have the same set of features. Further, applications are modeled as sentences of a grammar where tokens are the names of the features.

B. Operations and its GUI Based Environment

APL deals with standard array problems 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. The following table shows the sentences where tokens are names of features:

Tokens	Expressions
APL	At Opr Alg
At	Number char
Opr	Traversal Insertion Deletion Sorting Searching Multiplication
Alg	One Dim. Two Dim

Figure 1 shows a GUI that implements the grammar to model the expressions in the above table and allows APL products to be specified declaratively as a series of radio-buttons and check-box selections.

C. Semantics of the APL Features

An array type is determined by the elements it stores i.e. either number or character elements. The different operations that can be performed are traversal, insertion, deletion, sorting, searching, summation and multiplication over the array or with a scalar. Based on the array type and operations performed, an algorithm will be generated from the meta-expression which is obtained from the GUI option selected.

The meta-expression to get an algorithm for a traversal of a one dimensional array, storing number is:

$$APL = \text{Number. Traversal. One Dim.} \dots\dots (Eqn. 1)$$

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 in the example, the selection of one dimensional array will disable the multiplication option.



D. Steps involved in designing and handling APL

- 1) A GUI based environment has been designed, similar to the Fig.1
- 2) A meta-expression is obtained as users select APL features (similar to Eqn. 1) by inserting the terms as in Fig. 2.
- 3) This technology is known as Feature Oriented Programming (FOP) where model of a product-line is algebra: constants represent base programs (e.g. One Dim. Array) and functions add features (Traversal, Number) to base programs.
- 4) We apply introduction of AOP (Aspect Oriented Programming) that 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, add traversal and number data type). Introduction is meta-programming addition [7].
- 5) The base program will contain the declaration of the one/two dimensional array. Say $F(x)$ is a feature which modifies the base program d_f and introduces new members like t_f (traversal option) and n_f (Number option). A general form of all FOP features F is: $F(x) = n_f + t_f + d_f$
- 6) A program code is to be synthesized by evaluating this meta-expression.

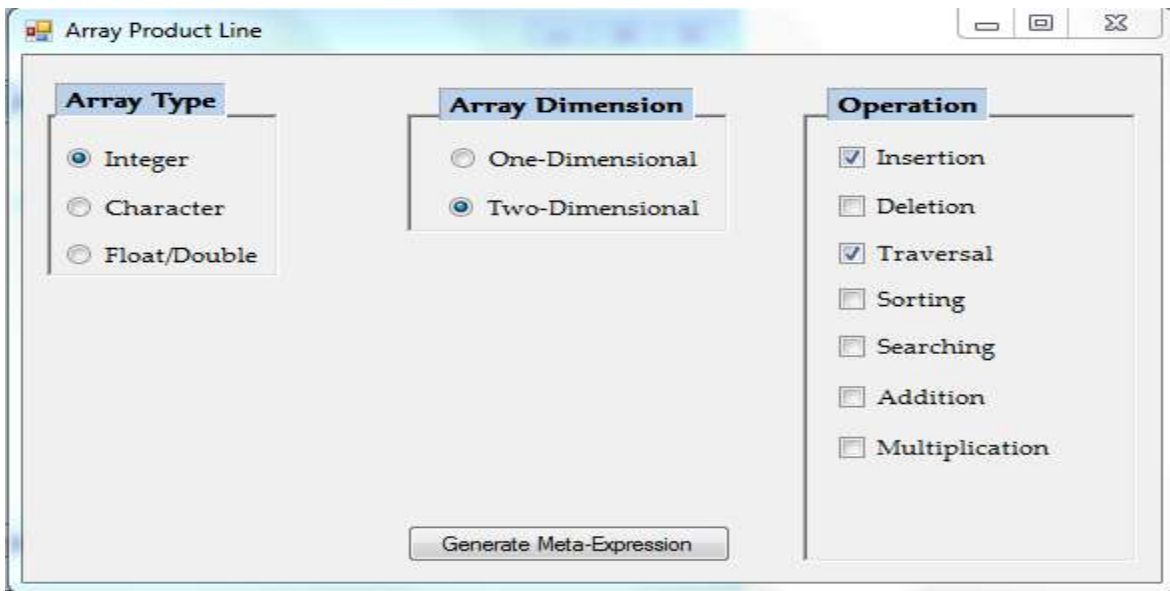


Figure 1

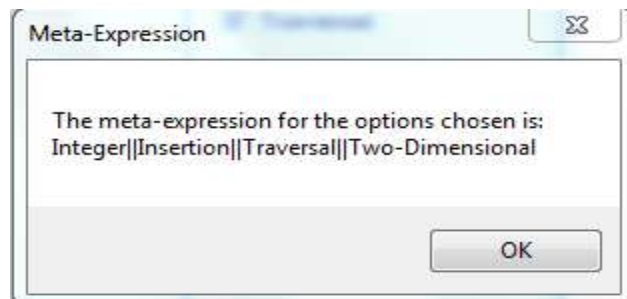


Figure 2



E. Designing APL with Meta-Expression Generation

As per the APL described above we proceed for the designing of the APL environment with meta-expression generation and conversion of the expression into C++ codes. The conversions of some selected features are done. The codes of some activities are given below:

- **The code in c# controlling the working of the Software**

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows.Forms;
using System.Diagnostics;
using System.Security.Permissions;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
        }
    }
}
```

First of all we provide a basic code to activate the options of the algorithms

```
//Opens/closes options under the addition function i.e. scalar and complete.
private void checkBox6_CheckedChanged_1(object sender, EventArgs e)
{
    if (checkBox6.Checked == true)
    {
        panel4.Visible = true;
        checkBox8.Visible = false;
        checkBox7.Visible = true;
    }
    else
    {
        panel4.Visible = false;
        checkBox7.Visible = false;
        checkBox8.Visible = true;
    }
}
```

Next, we fetch the options on the press of the “Generate Code” Button. Based on these options first we determine whether the choices have been configured in our code, i.e. will they be able to generate the required code. If not, we present an error message saying that “Some of the options or combinations chosen have not yet been configured. Sorry for the inconvenience.”. If atleast one option isn’t chosen from all three categories of the data type, dimension and algorithm we present the error message “Please choose atleast one option from each container.”.

If all options are checked out correctly, we then move on to fetch the appropriate code text file to edit and include as the header.



Next, we check if the insertion option has been chosen and also we check whether it is for 1-d or 2-d. Based on these selections, we load the appropriate text file and add the code into the file of C++ code. Similarly, we check if deletion is included and perform similar tasks for each selections. This is followed by the code generation according to the options selected. If all options are checked out correctly, we then move on to fetch the appropriate code text file to edit and include as the header.

We then convert the text file into a header file and copy it into the include folder of the default program that runs c++ codes in the user directory. This can be achieved by providing an installations wizard that can modify the particular strings in the code. We thus created the final cpp file that can be used by the user directly with functions included for the specified options. Then, a batch file is executed to start the default compiler on user's machine.

The following codes enable one to generate the meta-expression:

//code to generate meta-expression on button click

```
private void button2_Click(object sender, EventArgs e)
{
    //conditions to display error message for invalid/incomplete selections
    if ((radioButton1.Checked == false && radioButton2.Checked == false && radioButton3.Checked == false) ||
        (radioButton4.Checked == false && radioButton5.Checked == false) || (checkBox1.Checked == false && checkBox2.Checked ==
        false && checkBox3.Checked == false && checkBox4.Checked == false && checkBox5.Checked == false && checkBox6.Checked
        == false && checkBox7.Checked == false && checkBox8.Checked == false))
    {
        label4.Visible = true;
        label4.Text = "Please choose atleast one option from each container.";
    }
    else if (checkBox8.Checked == true || checkBox7.Checked == true || checkBox6.Checked == true || checkBox4.Checked ==
    true || (checkBox5.Checked == true && radioButton5.Checked == true) || (checkBox2.Checked == true && radioButton5.Checked ==
    true))
    {
        label4.Visible = true;
        label4.Text = "Some of the options or combinations chosen have not yet been configured." + System.Environment.NewLine
        + "Sorry for the inconvenience.";
    }
    else
    {
        //complete code to generate meta-expression based on selection
        label4.Visible = false;
        string chosen = "";
        if (radioButton1.Checked == true)
            chosen += radioButton1.Text;
        else if (radioButton2.Checked == true)
            chosen += radioButton2.Text;
        else if (radioButton3.Checked == true)
            chosen += radioButton3.Text;

        if (checkBox1.Checked == true)
            chosen += "|" + checkBox1.Text;
        if (checkBox2.Checked == true)
            chosen += "|" + checkBox2.Text;
        if (checkBox3.Checked == true)
            chosen += "|" + checkBox3.Text;
        if (checkBox5.Checked == true)
            chosen += "|" + checkBox5.Text;

        if (radioButton4.Checked == true)
            chosen += "|" + radioButton4.Text;
        else if (radioButton5.Checked == true)
            chosen += "|" + radioButton5.Text;
```

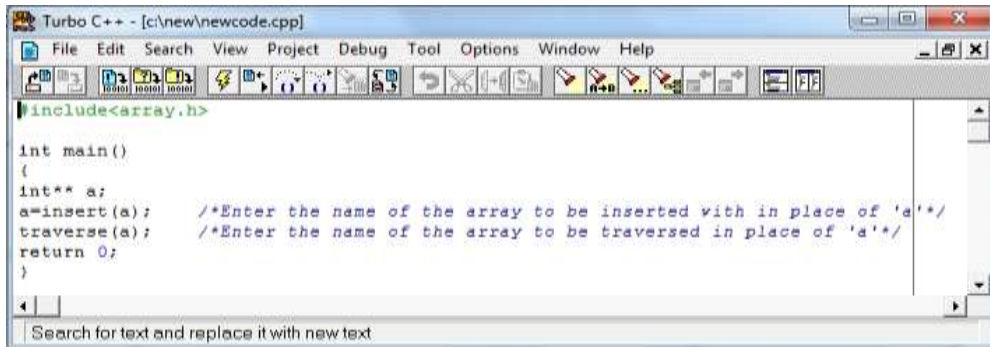



```
//display the meta-expression on a dialog box
MessageBox.Show("The meta-expression for the options chosen is:" + System.Environment.NewLine + chosen, "Meta-
Expression");
button2.Visible = false;
button1.Visible = true;
}
}
}
```

F. Conversion of meta-expression to C++ code

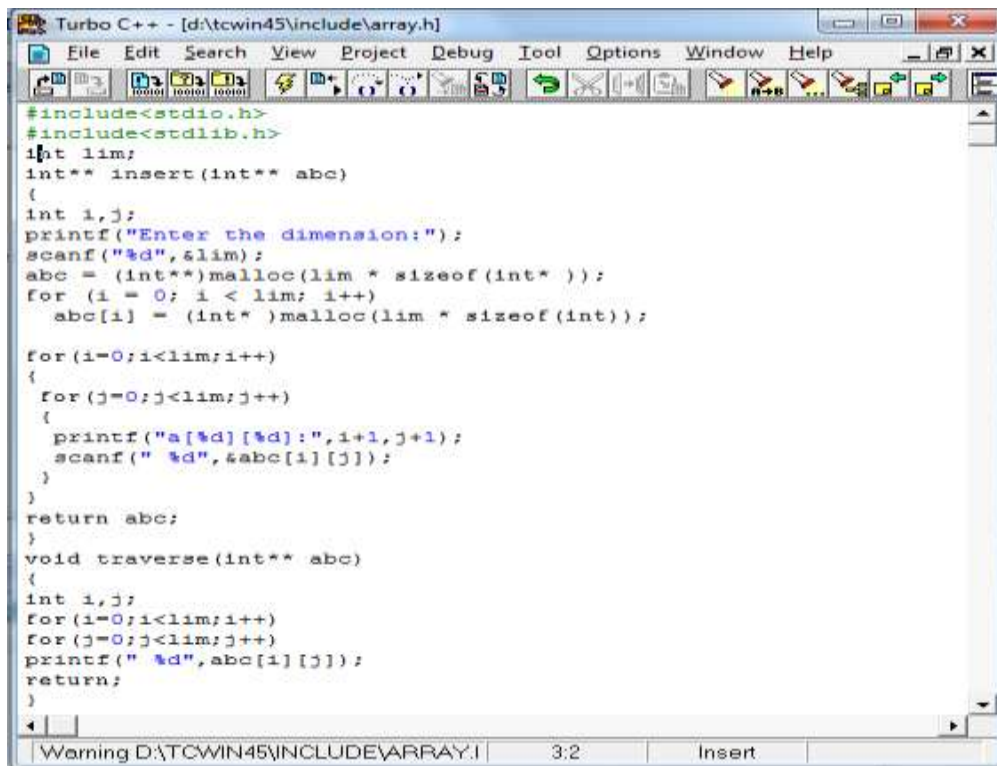
Output:

Once the “Generate Code” button is pressed, the GUI runs the batch code from a text file to open the default C++ editor and compiler with a new file named “newcode.cpp” with a set of pre-written code as shown below. The header file created is shown further below.



```
Turbo C++ - [c:\new\newcode.cpp]
File Edit Search View Project Debug Tool Options Window Help
#include<array.h>

int main()
{
int** a;
a=insert(a); /*Enter the name of the array to be inserted with in place of 'a'*/
traverse(a); /*Enter the name of the array to be traversed in place of 'a'*/
return 0;
}
```



```
Turbo C++ - [d:\tcwin45\include\array.h]
File Edit Search View Project Debug Tool Options Window Help
#include<stdio.h>
#include<stdlib.h>
int lim;
int** insert(int** abc)
{
int i,j;
printf("Enter the dimension:");
scanf("%d",&lim);
abc = (int**)malloc(lim * sizeof(int* ));
for (i = 0; i < lim; i++)
abc[i] = (int*) malloc(lim * sizeof(int));

for(i=0;i<lim;i++)
{
for(j=0;j<lim;j++)
{
printf("a[%d][%d]:",i+1,j+1);
scanf(" %d",&abc[i][j]);
}
}
return abc;
}
void traverse(int** abc)
{
int i,j;
for(i=0;i<lim;i++)
for(j=0;j<lim;j++)
printf(" %d",abc[i][j]);
return;
}
```



CONCLUSION

Software design is an art. Automating the key tasks in program design, synthesis and maintenance is yet a challenge. Metaprogramming is a technique for automation and sophisticated program design. Template metaprogramming is not as popular because of its complexity. Hence we found an alternative mechanism for metaprogramming by generating metaexpressions through a GUI environment. The meta-expression can be further converted to source code on user's demand. We experimented this in Array Product Line on some selected array operations like insertion, deletion etc. As a future work all the array operations can be activated. Moreover, the Model Driven approach through AOP (introduction) and FOP has been explained in this context. Metaprogramming can be popular and user friendly only when the user interaction with the environment is simple. This is achieved in APL by interacting through a GUI. This way of automation will lead to higher level programming languages, declarative languages for specifying programs in narrow domains, sophisticated IDEs etc. As a future work, the source code generated in C++ can be replaced with AspectC++ language which brings AOP in C++ domain.

REFERENCES

- [1]. R.E. Lopez-Herrejon and D.Batory, "A Standard Problem for Evaluating Product-Line Methodologies", GCSE 2001.
- [2]. Czarnecki, K. and Eisenecker, U., "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2000.
- [3]. K. Kang, et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Tech. Report CMU/SEI-90-TR-21.
- [4]. D. Batory. "From Implementation to Theory in Product Synthesis", POPL 2007 keynote.
- [5]. A. Kleppe, J. Warmer, and W. Bast, "MDA Explained: The Model-Driven Architecture, Practice and Promise", Addison-Wesley, 2003.
- [6]. Don Batory, "A Tutorial on Feature Oriented Programming and Product-Lines", ICSE'03.
- [7]. R. Lopez-Herrejon, D. Batory, and C. Lengauer, "A Disciplined Approach to Aspect Composition", PEPM 2006.

