ARAB ACADEMY FOR SCIENCE, TECHNOLOGY AND MARITIME TRANSPORT

COLLEGE OF COMPUTING AND INFORMATION TECHNOLOGY

A Multi-Paradigm Metaprogramming Facility for Automating Software Development Activities

Thesis submitted to the Computer Science Department
in partial fulfillment of the requirements for the degree of
"Master of Science"

Submitted by

Ahmed Abdel Mohsen Ahmed Hassan

B.Sc. Computer Science and Automatic Control 2003
Computer Science and Automatic Control Department
Faculty of Engineering, University of Alexandria
Alexandria, Egypt

Supervised by

Prof. Dr. Amin A. Shoukry
Professor of **********
Computer Engineering and Systems Department
University of Alexandria

Dr. Meer A. Hamza
Professor of **********
Computer Science Department
Arab Academy for Science,
Technology and Maritime Transport

**** 2006**

DECLARATION

[]

ACKNOWLEDGEMENT

[]

ABSTRACT

[]

CONTENTS

LIST OF SYMBOLS

[]

LIST OF TERMS

[]

LIST OF FIGURES

LIST OF TABLES

[]

CHAPTER 1: INTRODUCTION

Software applications are becoming increasingly large (i.e. have more concerns), more complex (i.e. increased cross-cutting), and more susceptible to change[1] . Development tools, on the other hand, do not provide the appropriate support to developers to face these challenges; because they, the tools, lack intentionality.

The lack of intentionality causes a large semantic gap to exist between the requirements and the implementation. Doubling the size of an application would typically cause the gap between its requirements and implementation more than to double[1]. Therefore, Developers spend an increasingly large time and effort to turn requirements into working applications.

During the last decade, a number of software development paradigms have emerged. Examples of these paradigms include: Intentional Programming (IP), Generative Programming (GP), Invasive Software Composition (ISC), and Model Driven Development (MDD). Certainly, such paradigms have helped us to have a better understanding of software development activities. Today, a large portion of the development work is considered to be non-creative rote work. Jacobson [] estimates that developers spend 80% of their time on non-creative work.

Today, we have the understanding as well as the motivation to automate a number of software development activities; automating any of the software development activities shall help bridging the large semantic gap between the requirements and the implementation, thus saving development effort. However, we lack the appropriate tools for achieving such goal. This work aims at finding better tools to automate rote software development activities.

1.1 ROTE SOFTWARE DEVELOPMENT ACTIVITIES:

To our knowledge, there are four software development activities that can be automated. These are: code generation, composition of code belonging to different concerns, adaptation of components, and propagation of changes.

[1] The empirical equations of software cost estimation can give us a sense of this gap.

Code generation is the process of translating high level abstract specifications into source code. For example, code generator can accept names of database tables (as a high level specification) and generate a persistence layer (low level source code). Automating the process of code generation would allow developers to work at a high level and more intentional level of abstraction.

Weaving a debugging aspect into an application is an example of composition. Automating the composition of code belonging to different concerns would increase intentionality in software development because it enables developers to focus on a single concern, and later on have it weaved with the rest of concerns.

By automating the adaptation of components, a component can be reused in more contexts, thus raising the level of reuse. Adaptation of components is sometimes called as "dependency injection".

By automating the propagation of changes, developers can change only one place in their source code and have the rest of dependent locations updated automatically. This can save developer's effort and decrease the probability of errors. For example, when changing the name of a method, all references to that method can be automatically updated.

1.2 METAPROGRAMMING:

The aforementioned activities can be automated by specifying them as metaprograms. Metaprograms are programs that work on object programs [2] or program fragments as data. A metaprogramming system provides developers with an interface to express their metaprograms and then executing them.

In general, there are two types of metaprograms: static and dynamic. Static metaprograms work on object programs that are out of execution. Dynamic metaprograms work on object programs while they are being executed. Throughout the rest of this thesis we will be using the word metaprograms and metaprogramming to refer to static metaprograms and static metaprogramming. Metaprogramming is sometimes referred to as program transformation.

Metaprograms work by acquiring data about object programs. Then the acquired data is used to customize a set of modifications. Finally, modifications are

[2] Object programs in the context of static metaprogramming means source programs (i.e. not compiled programs).

applied to object programs. The following example illustrates the working of a typical metaprogram.

1.3 MOTIVATING EXAMPLE:

Consider the class "TestClass" shown in Figure 1-1. "TestClass" has a single field "s" along with a getter and a setter. At the beginning of both methods, debugging information is printed to output stream. The debugging information contains the class name, method name, and all variables that the method refers to as well as their values. "TestClass" has a third method named "m".

```
public class TestClass {
    int s;

    public int getS() {
        System.out.println("DEBUG: class TestClass, method getS");
        System.out.println("DEBUG: field s = " + s );
        return s;
    }

    public void setS(int s) {
        System.out.println("DEBUG: class TestClass, method setS");
        System.out.println("DEBUG: field s = " + this.s );
        this.s = s;
    }

    public void m(TestClass2 tc2){
        tc2.s = 0;
    }
}
```

Figure 1-1: Class TestClass

Suppose that it is required to change the name of the field from "s" to "x". Figure 1-2 highlights the places that must be changed. The changes that must be made are: field declaration, field references ("s" can be referenced from any class residing in the same package as "TestClass"), names of the getter and the setter as well as their references, field name as well as its getter and setter names used in debugging information. In the next few paragraphs we shall present a number of approaches for achieving this task.

```
public class TestClass {
    int s;

    public int getS() {
        System.out.println("DEBUG: class TestClass, method getS");
        System.out.println("DEBUG: field s = " + s );
        return s;
    }

    public void setS(int s) {
        System.out.println("DEBUG: class TestClass, method setS");
        System.out.println("DEBUG: field s = " + this.s );
        this.s = s;
    }

    public void m(TestClass2 tc2){
        tc2.s = 0;
    }
}
```

Figure 1-2: Places to be changed

A metaprogram intended to rename the field "s" into "x" must, at first, identify the places that must be changed (data acquisition stage). An intentional way of specifying the data acquisition stage would allow developers to state "What" are the data to be acquired rather than "How" to acquire the data.

The first approach would be to view the class source code as a sequence of characters and state the problem as the replacement of occurrences of character 's' with 'x'. Figure 1-3 shows that there will be undesirable updates. This reflects the fact that this approach cannot be used to intentionally express the task of renaming a field.

```
public class TestClass {
    int s;

    public int getS() {
        System.out.println("DEBUG: class TestClass, method getS");
        System.out.println("DEBUG: field s = " + s );
        return s;
    }

    public void setS(int s) {
        System.out.println("DEBUG: class TestClass, method setS");
        System.out.println("DEBUG: field s = " + this.s );
        this.s = s;
    }

    public void m(TestClass2 tc2){
        tc2.s = 0;
    }
}
```

Figure 1-3: Changed Places Using the First Approach

The second approach would be to view the class source code as a sequence of tokens and then state the problem as changing tokens 's' into tokens 'x'. Although this approach leads to a fewer number of undesirable changes, a number of desirable changes (e.g. getter and setter names) are crossed out. A text search and replacement facility that supports regular expressions can be used to achieve this task. Figure 1-4 shows the places changed in "TestClass" using this approach.

```
public class TestClass {
    int s;

    public int getS() {
        System.out.println("DEBUG: class TestClass, method getS");
        System.out.println("DEBUG: field s = " + s );
        return s;
    }

    public void setS(int s) {
        System.out.println("DEBUG: class TestClass, method setS");
        System.out.println("DEBUG: field s = " + this.s );
        this.s = s;
    }

    public void m(TestClass2 tc2){
        tc2.s = 0;
    }
}
```

Figure 1-4: Changed Places using the Second Approach

The third approach is to specify the changed places based on the syntactic roles that tokens play. Following this approach, the problem can be stated as changing the tokens that represent either a field declaration or a field reference. Syntactic roles of tokens are calculated via parsing and they are typically stored in a parse tree or an Abstract Syntax Tree (AST). Figure 1-5 shows the changed places when the third approach is employed.

```
public class TestClass {
    int s;

    public int getS() {
        System.out.println("DEBUG: class TestClass, method getS");
        System.out.println("DEBUG: field s = " + s );
        return s;
    }
    public void setS(int s) {
        System.out.println("DEBUG: class TestClass, method setS");
        System.out.println("DEBUG: field s = " + this.s );
        this.s = s;
    }

    public void m(TestClass2 tc2){
        tc2.s = 0;
    }
}
```

Figure 1-5: Changed Places Using the Third Approach

The fourth approach would employ semantic constraints such as the target of a field reference. Using this approach a number of undesirable changes are crossed out. Figure 1-6 shows the changes using the fourth approach.

```
public class TestClass {
    int s;

    public int getS() {
        System.out.println("DEBUG: class TestClass, method getS");
        System.out.println("DEBUG: field s = " + s );
        return s;
    }
    public void setS(int s) {
        System.out.println("DEBUG: class TestClass, method setS");
        System.out.println("DEBUG: field s = " + this.s );
        this.s = s;
    }

    public void m(TestClass2 tc2){
        tc2.s = 0;
    }
}
```

Figure 1-6: Changed Places Using the Fourth Approach

The final approach would employ design time knowledge. For example, the fact that the methods "getS" and "setS" are named after the variable "s", or how the debugging strings are calculated based on its surrounding context. Design time knowledge is not captured by the programming language and therefore they are lost

during the development. Employing this knowledge makes it possible to correctly acquire the places that need to be changed.

Figure 1-7 shows a "rename field" wizard. The wizard is essentially a metaprogram that implements the aforementioned changes. It can be noticed that in case the "Update textual matches in comments and strings" check box is checked, developers would be forced to manually examine the changes before they are applied to the object program.

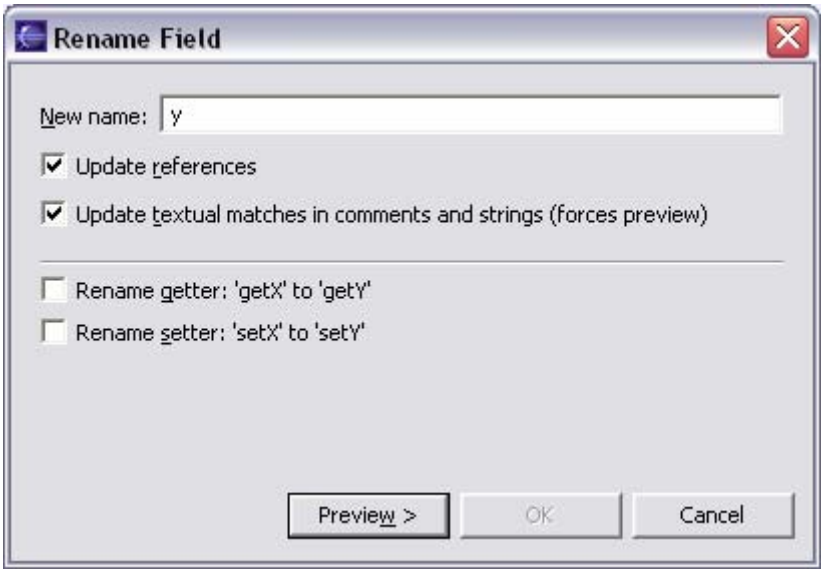


Figure 1-7: Rename Field Dialog Box

As seen from the previous example, metaprograms might be required to acquire global data that are semantically related to certain elements (e.g. references to a public field), and to apply global changes as well. This type of metaprogramming tasks is sometimes referred to in the literature as "global to global transformations". Development of such metaprograms is known to be largely ad-hoc [vis 03].

1.4 CONTRIBUTIONS OF THIS WORK:

It should be clear now that static metaprogramming can be of value to the field of software development. The first contribution of this work is a study of the benefits that static metaprogramming can bring to software development.

The second contribution of this work is a study of best practices for developing metaprograms. This study shall serve as a framework to evaluate metaprogramming systems; it is considered an advantage of a metaprogramming system to enforce or at least to support best practices.

This work also proposes views to support the capabilities of data acquisition concern in metaprograms. The last contribution of this work is a design and a proof of concept implementation of a metaprogramming system that supports views.

1.5 ORGANIZATION OF THE MATERIAL:

The rest of this thesis is organized as follows:

Chapter 2 Review

Chapter 3 how metaprogramming can aid the software development process

Chapter 4 requirements for metaprogramming system

Chapter 5 proposed system.

Chapter 6 implementation

Chapter 7 concludes this thesis

CHAPTER 2: REVIEW

This chapter starts with a presentation of a high level model of metaprogramming systems. And then for each of the model features, it gives a number of alternatives for implementing it. For each alternative, it discusses its pros and cons.

2.1 A MODEL OF METAPROGRAMMING SYSTEMS:

There are four differentiating features of metaprogramming systems. The first feature is how they internally represent object programs. The three other features are related to the way it allows developers to specify the three major concerns in metaprograms. These concerns are: data acquisition, modifications, and control. Figure 2-1 depicts a model for a typical metaprogramming system.

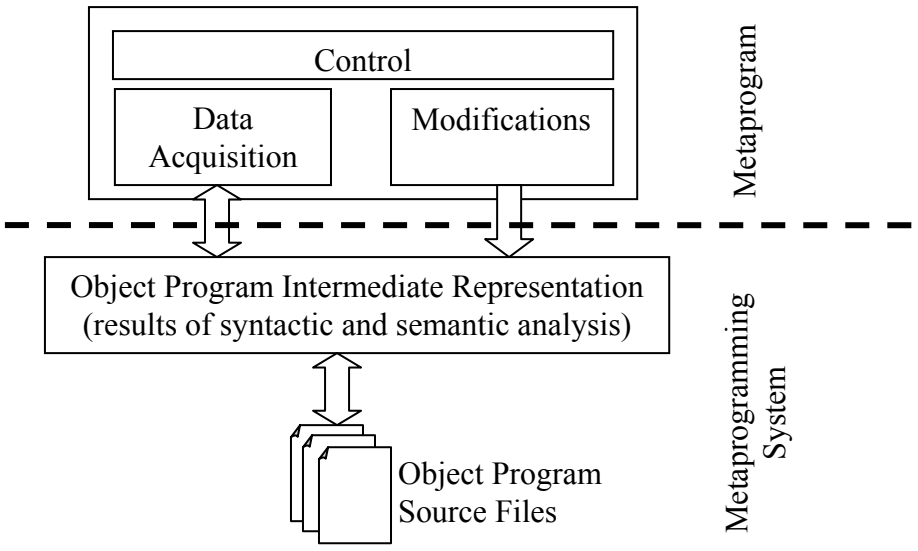


Figure 2-1: A Model for Metaprogramming Systems

To clarify the model in Figure 2-1 consider macros as a simple metaprogramming facility. Macro facilities treat object programs as a sequence of characters or tokens. Therefore, it can be said that macro facilities employ a textual representation for object programs. A macro typically consists of a prototype and a body. The prototype contains the macro name as well as a formal parameter list. The prototype is used for specifying the data acquisition concern. The acquired data

typically consists of the place of macro expansion as well as values of macro parameters. The macro body used to specify the modifications concern. It contains an object program fragment that shall replace a macro call. Typical macro facilities do not allow developers to explicitly control the macro expansion process.

It should be kept in mind throughout the rest of this chapter that metaprograms to automate the aforementioned programming activities need to acquire global data. The acquired data need to be specified based on syntactic and semantic information as well as design time knowledge. These metaprograms need also to apply global modifications to object programs. It is also typical to coordinate a number of metaprograms in order to achieve a large metaprogramming task.

2.2 OBJECT PROGRAM REPRESENTATION:

The intermediate representation of object programs is an important feature of a metaprogramming system that greatly affects its capabilities. Object programs are typically represented in form of a set of source files. This representation might cause some inconveniences in specifying and executing metaprograms. Therefore, most metaprogramming systems provide another intermediate representation for metaprograms that is more appropriate for specifying and executing metaprograms.

The intermediate representation should also allow metaprogramming systems to store the results of different analysis procedures applied to object programs. Typical analysis procedures include syntax analysis (parsing), semantic analysis (data flow, references, typing…), and special analysis based on design time knowledge. Ability to store analysis results is essential to eliminate the need to drive these results while specifying different concerns in metaprograms.

2.2.1 Text based program representation:

Textual program representation is the simplest to implement. A program is represented as a string. Using this representation scheme, metaprograms are specified as sequences of string manipulation operations. The actual selection of operations depends on the specification paradigm adopted by the metaprogramming system. For example, using the rewriting paradigm a metaprogramming system typically offers operations such as: pattern matching and replace.

Textual object program representation does not allow storing results of syntactic, semantic, or special analysis procedures. Therefore, even simple

metaprograms are hard to develop using a metaprogramming system that employs textual representation for object programs. For example, consider the problem of writing a metaprogram to find the fields of a given class. Among the issues that have to be considered: variables local to methods must not be counted, variables might be simple or arrays, variables might be initialized or not initialized, and restricting the scope of the search to the code of the class. Most of the time developers would find themselves forced to embed a compiler front end into their metaprograms. Simple macro facilities fall under this category of metaprogramming systems.

2.2.2 Tree based program representation:

Tree based program representation has become the most common representation scheme. Parser generators (such as: Antlr[], SableCC[], and many others) play a pivotal role in the wide adoption of tree based program representation. A key decision when choosing a tree representation of programs is: Weather to use abstract syntax or concrete syntax.

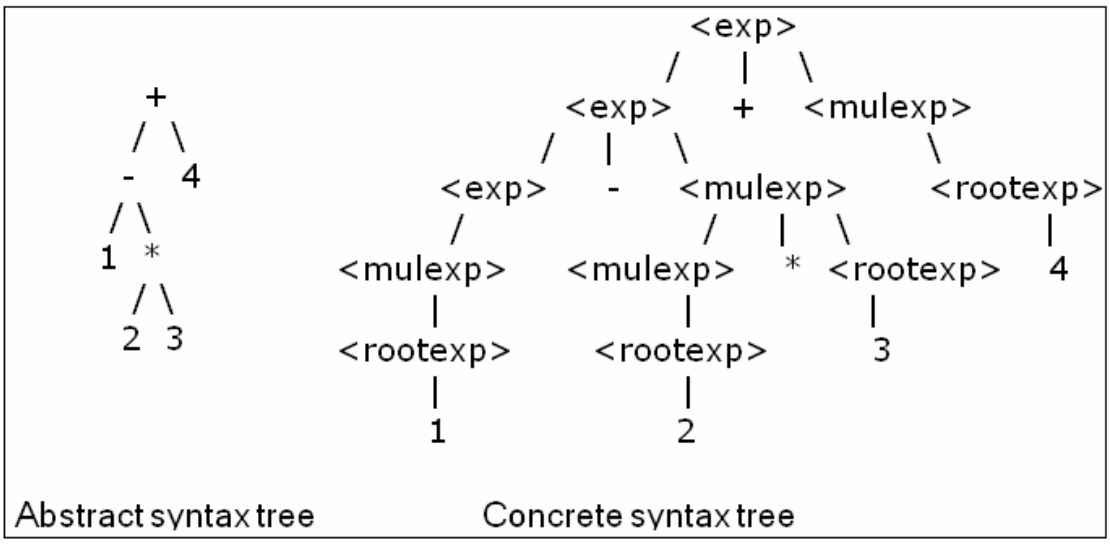


Figure 2-2: Abstract Syntax Tree Vs. Concrete Syntax Tree for Expression 1-2*3+4

Using a tree based program representation, syntactic information are readily available to developers to use when they write their metaprograms. The use of Abstract Syntax Trees (AST) gives developers another benefit. A single AST such as the one in Figure 2-2 can be equivalent to multiple concrete expressions. For example: 1-2*3+4, 1-(2*3)+4, … etc. Therefore, expressing metaprograms as operations on AST means that metaprograms need to handle a fewer number of cases which means

simpler metaprograms. It should also be noted that certain metaprograms need to operate on concrete syntax (For example: Metaprograms for enforcement of coding conventions). In such cases, it won't be appropriate to use AST.

As implied from its name, a tree based representation allows storing the results of syntactic analysis. However, it does not permit storing the results of semantic analysis or other special analysis.

2.2.3 Graph based program representation:

Graphs are suitable for representing not only the results of syntax analysis (parsing) but also the results of semantic analysis as well as the results of other special analysis based on design time knowledge. Thus all of these information can be made available to developers to use when writing their metaprograms. There are two key decisions that have to be taken when choosing a graph based program representation: choosing one of graph variants and deciding upon an ontology for object program source code.

Graphs have many variants: graph edges can be either directed or undirected. Graph nodes and edges can have labels. There are multigraphs, hypergraphs, and hierarchical graphs. In a multigraph, two nodes can have more than one edge. In a hypergraph, an edge may be connected to more than two nodes. Hierarchical graphs are graphs where a subgraph can be abstracted to one node and the bunch of edges between tow abstracted subgraphs to one edge. It should be noted that the relationship between hypergraphs and simple graphs resembles the relationship between high level programming language and a low level programming language; hypergraphs are more intentional for representing different situations. Hypergraphs can also be represented using simple directed graphs.

Designing an ontology for representing object programs means deciding what are the types of information about the object program are to be stored. As mentioned before, there are three classes of information: syntactic, semantic, and design time. Syntactic information can be represented at different levels of abstraction. For example, we can represent only information about packages and classes. A more concrete representation may contain the fields and methods inside each class. It is also needed to decide what types of semantic analysis results are to be stored (e.g. targets of method references, scope of variables, etc) and to decide what other special analysis procedures that are needed.

2.2.4 Logic based program representation:

Logic has been used for representing object programs. Information about programs can be expressed as predicates. For example: Class(x), Class(y), SuperClass(x,y). Using logic based program representation, facilitates queries. JQuery is a tool for searching and defining views on source code that uses logic based program representation.

2.3 SPECIFYING DATA ACQUISITION:

Code for the data acquisition concern is responsible for acquiring the necessary data from object programs. The acquired data may be either object program elements (i.e. lexemes of these elements) or places in object programs. Examples of object program elements include: public methods of a certain class or references to a certain public. Examples of places in object programs include possible places to insert a method in a certain class. To identify a place it is possible to identify its preceding or following element in the object programs. For example, a possible place to insert a method is "before method x".

An intentional way of specifying the data acquisition concern would allow developers to state "What is" the data to be acquired rather than "How to" acquire the data. An essential requirement for achieving intentionality in specifying the data acquisition concern is to provide developers with constructs to specify data to be acquired based on syntax, semantics, as well as design time knowledge.

2.3.1 Pattern matching and unification:

Pattern (or template) matching is the basic operation used to detect places in object programs. Macros are metaprogramming facilities that depend on pattern matching to identify places for macro expansions.

Patterns sometimes have variables. When such patterns are matched, values of these variables are determined. This is called unification. Unification is the basic operation used to acquire elements of object programs. A macro prototype (name and formal parameter list) resembles a pattern with variables. The formal parameter list is unified with the list of arguments following a macro call. Pattern variables can have constraints on the possible values that these variables can take or their cardinalities (e.g. single valued or multi-valued).

2.3.1.1 Patterns of concrete object syntax:

Patterns can be specified using the concrete syntax of object programs. Concrete syntax patterns are suitable only for fine grain patterns. Specifying coarse grain patterns using concrete syntax can be tedious and error prone [MetaJ]. For example, suppose that it is required to identify integer fields in a certain class. The concrete pattern in Figure 2-3 can be used to match integer fields. "FieldName" is enclosed in angular brackets to indicate that it is a variable.

```
int <FieldName>;
```

Figure 2-3: Concrete Syntax Pattern of Integer Fields

The problem with this pattern is that it can match local integer variables in addition to integer fields. For example, the pattern will have two matches with the sample object program shown in Figure 2-4 (a); one with field 'x' and the other with the local variable 'y'.

```
class C{
   int x;
   public int m(){
       int y;
   }
}
```

(a)

```
class C{
   public int m(){
       int y;
   }
   int x;
}
```

(b)

Figure 2-4: Sample Object Program

Since patterns of concrete syntax do not allow developers to explicitly specify syntactic constraints; the only solution is to specify syntactic constraints is to use coarse grain patterns such as the patterns given in Figure 2-5.

Figure 2-5 (a) shows a coarse grain pattern of concrete syntax. The problem with this pattern is that it matches only classes that consist of only a single integer field. The pattern in Figure 2-5 (b) is essentially the same pattern in Figure 2-5 (a) but with a number of extra variables. Variables are needed to make the pattern more generic (i.e. to make it match classes other than those consist of single integer field).

```
class <v1>{
   int <FieldName>;
}
```

(a)

```
<v1> class C <v2>{
   <v3>
   int <FieldName>;
   <v5>
}
```

(b)

Figure 2-5: Coarse Grain Patterns of Concrete Syntax

The pattern in Figure 2-5 (b) might not be what we really need; for example, when matching the pattern to the object program in Figure 2-4 (b), the variable "v3" may be matched to "public int m(){" at the same time "v5" may be matched to "} int x;" thus having "FieldName" matched to "y" which is not what we need.

Adding constraints on the values that variables can take is also possible at the cost of extra complexity incurred in developing properly working patterns. For example, adding a constraint that "v3" has to be a method or field is not enough since it is also possible to have a static code block at the beginning of the class. Such comprehensive description of constraints complicates the pattern. However it is indispensable for proper working of the pattern.

2.3.1.2 Patterns of abstract syntax:

Patterns of abstract syntax differ from patterns of concrete syntax in that every token in abstract syntax patterns is assigned a syntactic role. When an abstract syntax pattern is matched to an object programs, the syntactic roles of its tokens are taken into consideration. Patterns of abstract syntax constitute a tree rather than a sequence of tokens. Pattern trees are matched to the abstract syntax tree of object programs rather than the object program's text.

Non-terminal symbols of the language of object program can be employed to describe the syntactic roles of the tokens of the pattern. Figure 2-6 shows an abstract syntax pattern of a method with an integer local variable. Abstract syntax patterns are harder to specify than fine grain patterns; this is due to the fact that developers have to specify syntactic role of every token. Abstract syntax patterns are not suitable for expressing semantic constraints.

```
Pattern p = new Pattern();
Method m = new Method();
m.setVisibilityModifier(VisibilityModifiers.PUPLIC);
Variable v = new Variable();
v.setType(PrimitiveTypes.INT);
m.addVariable(v);
p.add(m);
```

Figure 2-6: An Abstract Syntax Pattern

2.3.2 Traversals:

When multiple fine grain patterns are used for data acquisition, there must be a way to control the order and the scope of the matching operations. Traversals can be employed to achieve such function. Figure 2-7 shows how a traversal can be used for data acquisition. Suppose that we have "class", "method", "statement", "field or local variable" as fine grain patterns. It is possible to employ a traversal to identify local variables based on the fact that local variables can descend only from methods and not from classes.

The traversal that we will follow to identify local variables is as follows: first start at the root of the object program and look for children of type class. For every child of type class, look for children of type method. Then, for every method, look for children of type "field or local variable".

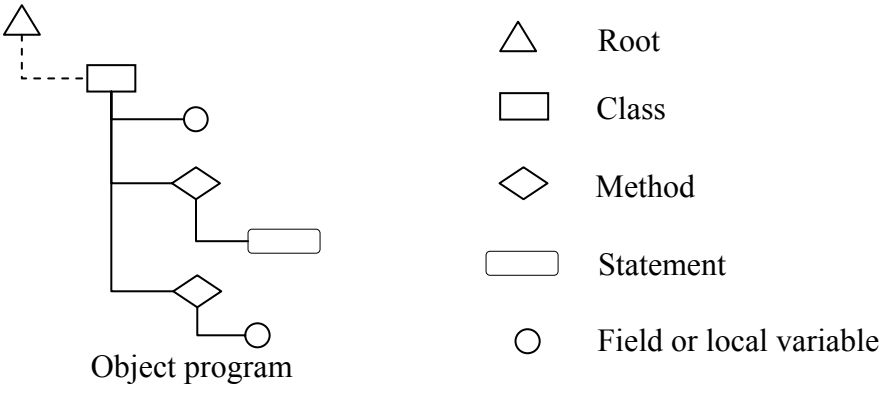


Figure 2-7: Using Traversals for Data Acquisition

As seen from the previous example, using traversals allows the use of more than one pattern. It also reduces the number of possible matches to fine grain patterns

and enables global data acquisition. However, traversals describe how to acquire data rather than what data are to be acquired. Therefore, traversals are less intentional for specifying data acquisition.

Traversals are typically specified using a general purpose programming language such as JAVA. Strategic Programming (SP) is a recent paradigm that allows traversals to be specified in a declarative manner using basic traversal strategies (such as ALL for visiting all of the children and ONE for visiting one of the children) and strategy combinators (such as SEQ for applying two strategies one after another and CHOICE for applying one of two strategies). Stratego is a metaprogramming system that employs SP [].

2.3.3 Attribute grammars:

In attribute grammars, attributes are attached to nodes of object program syntax tree. For each of the attributes, a specification of how to calculate its value based on values of other attributes belonging to the same node, its parent, or its children. Developers needn't specify the order of evaluation of attributes. Attribute grammars are suitable for global data acquisition. Pattern matching can take place after evaluating all of the attributes in the tree, thus matching can be done based on global information. Figure 2-8 illustrates how attribute grammars can be used to support data acquisition.

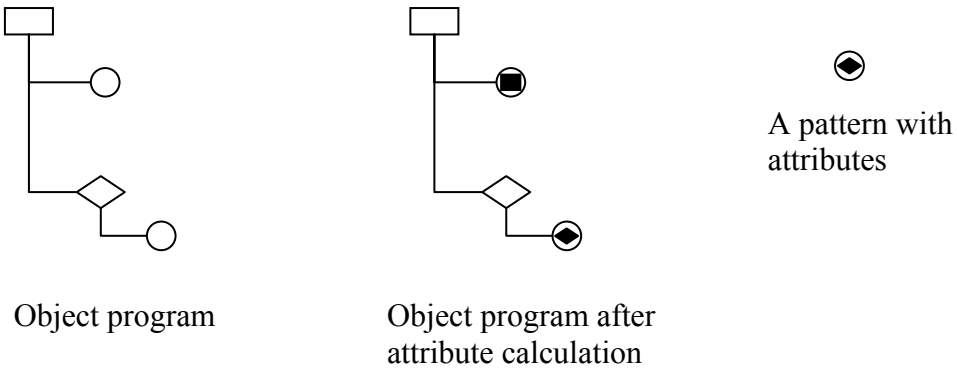


Figure 2-8: Using Attribute Grammars to Support Data Acquisition

2.3.4 Manual Data Acquisition:

Some metaprogramming facilities do not perform any data acquisition at all. All the necessary data are acquired from developers through dialog boxes or

configuration files. This setting is quite common for code generators. Code generators usually generate code that is encapsulated in black boxes.

2.4 SPECIFYING MODIFICATIONS:

The modifications concern is about changing the object program. Modifications can take the form of inserting new elements to existing object program at certain places such as adding a method to an existing class. Modifications can also take the form of deleting certain object program element or changing a property such as turning a method form being private to being public.

The easiest and most intentional way of specifying the modifications concern is to state "what are" the modifications need to be applied to object programs. However, this is not always the case and developers find themselves sometimes forced to specify "how" the modifications need to be applied.

There are two approaches to specifying modifications: forward and backward []. In the forward approach, modifications are specified as operations on elements of the object program. Clearly, this approach is less intentional because developers describe how modifications to object program are made.

In the backward approach, developers specify the target pattern that the object program (or a part of it) should finally look like. The metaprogramming system has to translate this target pattern into a list of insertions, deletions, and updates to object program elements. To do so, the metaprogramming system needs some sort of a "translational semantics".

There are three possible translational semantics: first is to insert all elements of the target pattern. Of course, no deletion can be specified using this approach. This approach is typical for code generators. The second approach employs two patterns LHS and RHS. The metaprogramming system contrasts the two patterns. Elements in the LHS but not in the RHS are translated to deletions. Elements on the RHS but not in the LHS are translated to insertions. Rewriting systems such as macros follows this approach. The last approach is to assign tags to elements of the target pattern that indicates which elements are to be inserted, deleted, or changed.

Metaprogramming systems that employ the backward approach for specifying modifications usually suffer from a major drawback which is the single place of insertion [] (i.e. they cannot be used to specify modifications that affect multiple

places in the object program). As a remedy, [] suggested that code generators should be equipped with invasive capabilities.

2.5 SPECIFYING CONTROL:

Non-trivial metaprograms are typically divided into multiple stages of data acquisition and modifications (i.e. a number of smaller metaprograms). For example, it is possible to employ a number of macros to achieve a single metaprogramming task. The control concern is about coordinating the execution of a number of small metaprograms in order to achieve a larger metaprogramming task.

Of course, specifying the control concern becomes harder when there are a large number of small metaprograms. Conflicts among the smaller metaprograms add up complexity to process of specifying control. Guaranteed termination of the larger metaprogramming task complicates the process of specifying control furthermore. Another issue is planning for efficient overall execution.

Figure 2-9 (a) shows an example of conflicting macros. Execution of the macro "I" before the macro "PI" shall destroy all the occurrences of "PI" in the code and hence make "PI" useless. Figure 2-9 (b) shows an example where execution of the macro "MAX" at first will produce two occurrences of MIN. However, execution of MIN at first will leave us with a single occurrence of MAX. Figure 2-9 (c) shows an example of a non-terminating macro.

```
#define  I   5
#define  PI   3.41

void main(){
    for ( int i=0; i < I; i++){
        printf("%f",PI);
    }
}
```

(a)

```
#define  MAX(a,b) (a)>(b)?(a):(b)
#define MIN(a,b) (a)<(b)?(a):(b)

void main(){
    int x=5,y=6,z=7,t;
    t = MAX(x,MIN(y,z));
        printf("%d",t);
    }
}
```

(b)

```
#define  MAX(a,b)   (MAX((a))>MAX((b)))?(a):(b)
```

(c)

Figure 2-9: Issues in Specifying Control

The algebraic approach for graph rewriting provides a method called "critical pairs analysis" to detect pairs of conflicting graph rewriting rules. It also provides a method for detecting the termination of a set of graph rewriting rules in certain a situation. However, detecting the termination is undecidable in general [graph96].

There are two approaches to specifying control: explicit and implicit. In the explicit approach developers specify the sequence of application of smaller metaprograms. To that end, a general purpose programming language such as JAVA can be employed.

In the implicit approach to specifying control, developers specify a set of constraints on the order of execution. The metaprogramming system is responsible for driving an execution sequence satisfying the given constraints. Clearly, this is more intentional than the explicit approach. Metaprogramming systems differ in the interfaces they provide to developers for specifying such constraints.

Constraints can take the form of application condition (i.e. a small metaprogram cannot be executed unless its application condition is satisfied). Conditions can also be checked after the application of the metaprogram. In case of failure, the work done by the metaprogram must be undone. Event driven execution of small metaprograms can also be employed for exception handling (e.g. when execution of one of the metaprograms results in two methods with the same name within the same scope, a special metaprogram can be fired to rename one of them).

One useful technique in specifying control is to divide the large set of small metaprograms into a number of smaller subsets. Only one of the subsets can be under execution at any given time. Deciding when set execution starts or stops as well as the order of set execution can be done using the same primitives as small metaprograms. (i.e. set execution condition, set stopping condition, explicit or implicit sequence of set execution…). Sets can also be recursively divided into smaller subsets.

CHAPTER 3: APPLICATIONS OF METAPROGRAMMING IN SOFTWARE DEVELOPMENT

This chapter starts with a discussion of the concepts of intentionality, reuse and agility in software development. It also shows how metaprogramming can help enhancing intentionality, reuse, and agility. Then it presents four recent software development paradigms that employ a disciplined form of metaprogramming.

3.1 INTENTIONALITY:

The concept of intentionality is best illustrated by means of an example. Suppose that we are writing a document and we need to insert line numbers at the beginning of every line in the document. Suppose also that the used word processor provides an insert-at-line-beginnings facility that accepts a sequence of characters and insert them at line beginnings. Using the insert-at-line-beginnings facility to insert line numbers is considered an intentional way of specifying the task. Clearly, if the insert-at-line-beginning facility was not provided, we would have been forced to simulate its working using the lower level type-a-character and move-the-cursor constructs. Moreover, we will be forced to repeat this simulation every time we need to insert line numbers or even to change the document.

An intentional way of specifying a task would require developers to state "what is" the task rather than "how" the task can be done. For a programming language to allow developers to intentionally specify their programs, it must provide constructs for every possible programming task. Clearly, this is not possible for a general purpose programming language. Instead, such a programming language can provide an extension mechanism that allows developers to add their own constructs.

This extension mechanism can be thought to be analogous to macro facilities of word processors. Returning to the previous example, if the used word processing facility allows its users to develop macros, we can develop a macro to insert line numbers at line beginnings. We can go further by developing a generate_line_numbers macro to generate the line numbers. This can be extremely useful in case we have a 1000 lines document.

The role that metaprogramming facilities can play in software development resembles the role that macro facilities play in word processing. Macros are used to specify extensions to the interface of a word processor. Metaprograms can also be used to specify extensions to a programming language.

3.1.1 Enhancing intentionality via metaprogramming:

There are two activities that can be automated via metaprogramming in order to enhance the intentionality. These are code generation and composition. In the previous example, the generate_line_numbers and the insert_at_line_beginnings macros resemble these two activities.

Code generation can be viewed as a translation of high level specification (e.g. first line number, an increment, and number of lines) into a lower level one (e.g. sequence of line numbers). By automating the translation, developers can work on a high level of abstraction. Working at a high level of abstraction means working with few details. Working with few details means higher productivity and ease of change.

Automated composition of code belonging to different concerns (e.g. line numbers and the rest of document in the previous example) allows developers to separately develop code belonging to each of them. By doing so, developers handle fewer details because details belonging to other concerns are made transparent. As was stated before, working with fewer details means higher productivity and ease of change.

3.2 REUSE:

Reuse can bring a number of benefits to software development such as: shorter development cycles, reduced cost, and increased reliability. Developers reuse source code manually via copy-paste-adapt operations. Artifacts such as design patterns are expressed as non invokable documents that cannot be effectively reused. This situation resembles providing a developer with a mathematics book instead of an invokable "sine" function.

Different reuse technologies allow for the automation of many aspects of the reuse process, thus rendering the reuse process more effective. For example, parameter passing mechanisms can be employed to achieve a limited form of adaptation, call and return mechanisms can also be employed instead of copy and paste. Figure 3-1 shows the landscape of current reuse technologies.

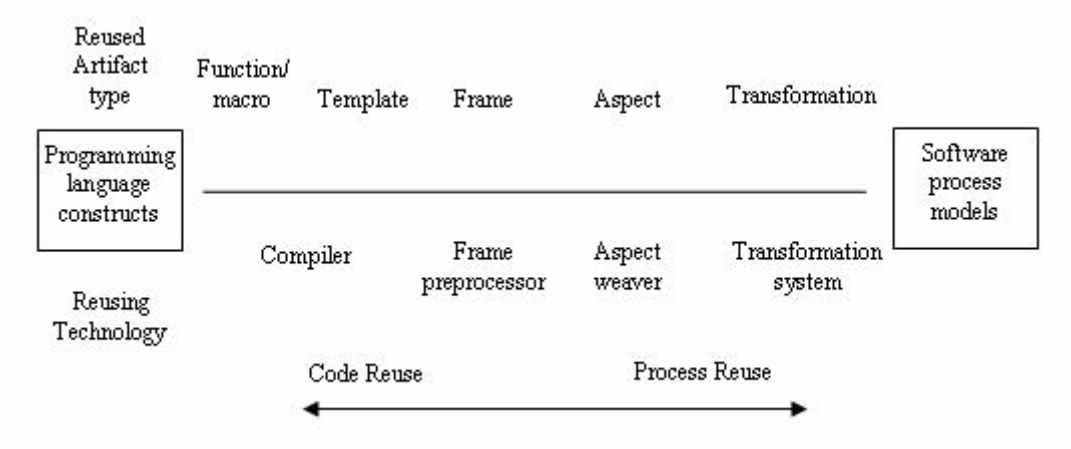


Figure 3-1: Reuse Technology Landscape

A recent ethnographic study [MK] showed that developers still reuse code via manual copy-paste-adapt operations at a relatively high rate. The study was performed on a set of researchers and showed that they are copying and pasting source code fragments at the rate of 16 times per hour, 4 times of them a complete method is reused. This study shows that there is a need for more powerful software reuse technologies.

It has been debated whether more powerful software reuse technologies should allow reusing larger "chunks" of source code or allow reusing the same source in more contexts [Batory]. The problem with the first approach is that the larger the reused artifact the more specific and less reusable it becomes.

Any reused artifact can be viewed as consisting of a fixed part and a variable part. The variable part can capture variability in either value (e.g. method parameter) or location (e.g. point cut of an aspect). The existence of a variable part in the reused artifact allows for its adaptation to a reuse context. Reuse technologies differ in their capabilities to handle variability in reused artifacts. Adaptation of artifacts can take place at compile time (e.g. templates and frames) or at runtime (e.g. functions). Reused artifacts can be passive (i.e. have no role in the adaptation process) or active (i.e. able to access their reusing context and adapt themselves to it).

3.2.1 Enhancing reuse via metaprogramming:

Metaprograms can be employed to adapt software artifacts to their reusing context. For metaprograms to be useful in this task, they must be independent of the reusing context. A reasonable degree of independence can be achieved by writing metaprograms that depend only on the meta-model of the context. The meta-model of

the context is less likely to change than the context. For example, a metaprogram for adding a getter of a certain field can be developed with the name of that field "hard coded" into the metaprogram. Clearly, such metaprogram will not be useful in other contexts.

Metaprograms can also be employed as a technology for reusing parts of the development process (e.g. the process of renaming a field or the process of applying a design pattern). Again, such metaprograms must depend only on the meta-model of object programs.

The distinction of code reuse and process reuse might seem to be artificial. This distinction begins to dissolve when process reuse is viewed as reusing artifacts with a small fixed part.

3.3 AGILITY:

The agile manifesto [http://agilemanifesto.org] was coined in 2001 by a number of software development experts. The manifesto consists of a number of principles. Here we are quoting the second principle: "Welcome changing requirements, even late in development. Agile process harness change for the customer's competitive advantage" [http://agilemanifesto.org/principles.html]

3.3.1 Enhancing agility via metaprogramming:

Changing software is challenging due to dependencies between different parts of the software. There are two approaches to deal with change: to regenerate the affected part along with all of its dependent parts, or to propagate changes to dependent parts. The first approach is supported by allowing users to specify changes to separate concerns at a high level of abstraction and then have these changed concerns automatically translated to lower level and composed together. The second approach is supported by the propagating changes to dependent places in the source code. It should be noted that the first approach reflects the fact that increased intentionality leads to increased agility.

3.4 BENEFITS OF METAPROGRAMMING IN SOFTWARE DEVELOPMENT:

Metaprogramming can be employed to automate four software development activities. These activities are: code generation, composition, adaptation of

components, and propagation of change. In the previous sections we discussed how automation of these activities can support software development. Figure 3-2 summarizes the benefits that metaprogramming can bring to software development.

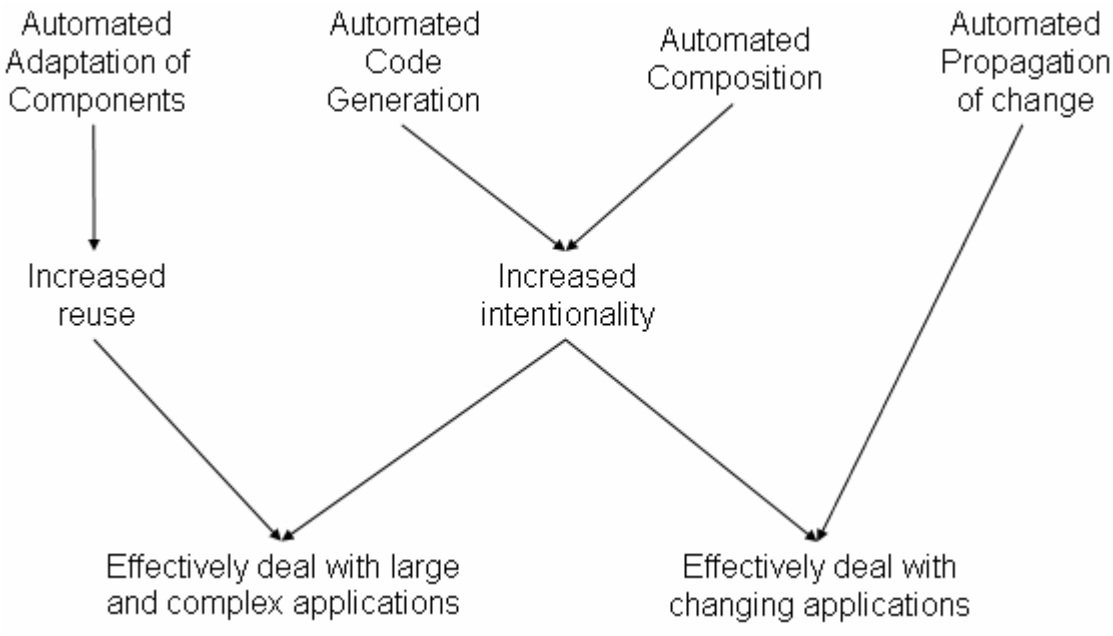


Figure 3-2: Benefits of Metaprogramming

3.5 RECENT SOFTWARE DEVELOPMENT PARADIGMS:

3.5.1 Generative Programming:

Generative programming (GP) is a software engineering paradigm based on modeling software families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge [].

Generative programming can be applied to well understood problem domains. GP is outlined in Figure 3-3. Suppose that we want to apply GP to the construction of GUIs. First we have to specify the problem space (i.e. how users will specify a GUI). A suitable approach is to have an XML file describing the GUI. The XML file would typically contain elements to represent the GUI domain concepts such as 'Form', 'Button', 'Text Box', with the nesting of these elements reflecting the structure of the GUI.

The next step is to specify the solution space. Here the solution space would typically contain a class for each of the concepts in the problem domain (Button, Textbox …). The last step is to construct a generator that accepts XML files containing a GUI instance and construct a GUI using classes defined in the solution space.

Problem spaces or domains can be either compositional or transformational. Compositional domains can be modeled as a gallery of components. Products are described by selecting appropriate elements from the gallery. A transformational problem domain consists of a set of possible transformations. Products are described by describing a sequence of transformations to be applied.

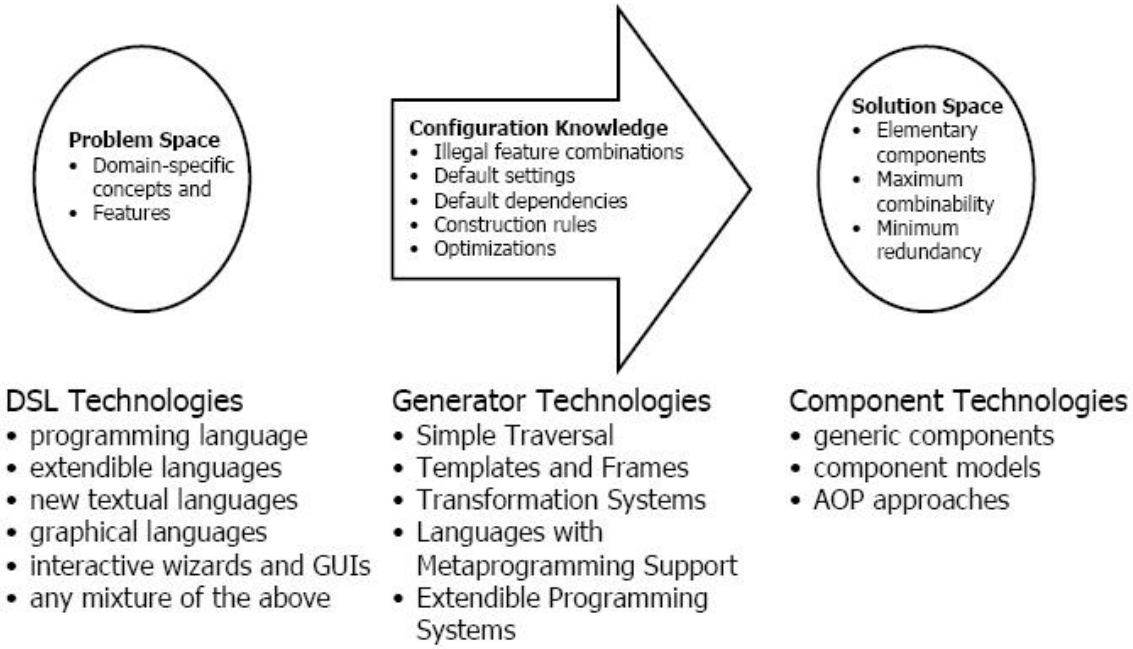


Figure 3-3: Generative Programming

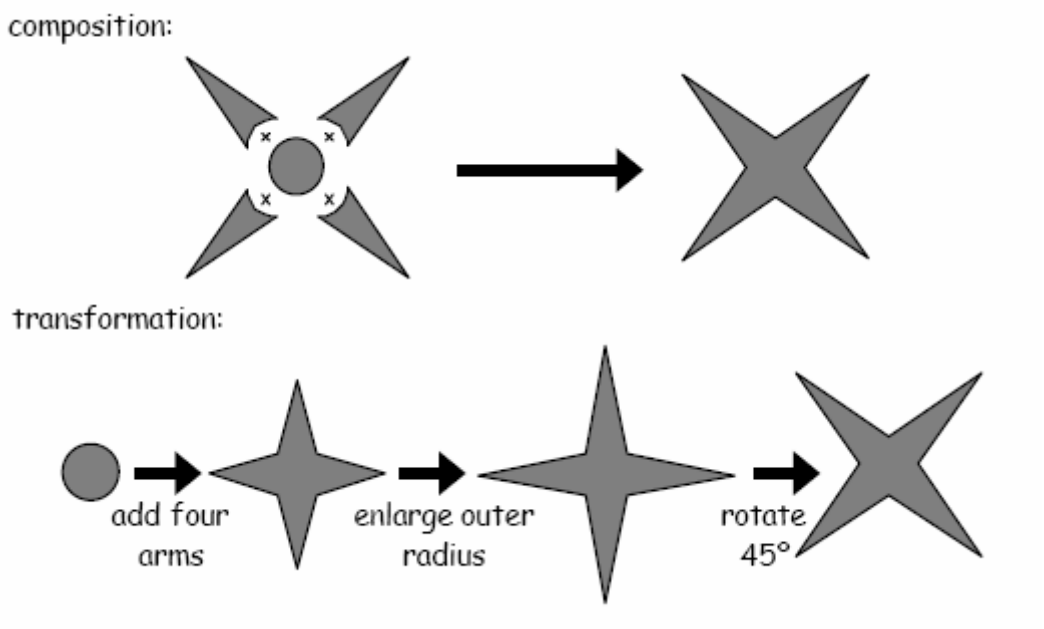


Figure 3-4: Compositional Vs. Transforamational Generation

Generators can also be compositional or transformational; in transformaitonal approach, generator can start with a simple base program and add up features by transforming it. In composistional approach, generator start with a simple base and add up features by adding components to it. Figure 3-4 illustrates the difference between compositional and transformational generation.

3.5.2 Intentional Programming:

In Intentional Programming (IP) developers are allowed to extend their programming language with new programming abstractions or intentions. New intentions can be domain specific or general purpose such as design patterns. This way, new abstractions can be adopted and used more rapidly than waiting for these abstractions to be incorporated into a new programming language.

The following example illustrates IP. Suppose that we need to enhance the traditional for loop by adding another continuation condition that is checked after the execution of every statement inside the loop body. To specify new intention, a developer has to specify how this intention is compiled. This is specified using reduction semantics (i.e. the developer specifies how this new intention is reduced to lower level intentions). Figure 3-5 shows how the enhanced for loop can be reduced to a traditional for loop and a set of if statements.

```
for(int i=0; i<10; i++; i<10){
  /*S1*/ x++;
  /*S2*/ y--;
}
```

```
for(int i=0; i<10; i++){
  if(i<10){
      /*S1*/ x++;
  }else{
      break;
  }
  if(i<10){
      /*S2*/ y--;
  }else{
      break;
  }
}
```

Figure 3-5: Enhanced For Loop Intention

In IP, Programs are represented as a tree rather than as a text file. Every node in the tree points to its declaration using a graph like pointer. Therefore, editing programs is performed through a structure editor rather than a text editor. An instance of an intention is represented internally as a tree. When an intention is specified, developers provide rendering methods for displaying it, type in methods for entering it, reduction methods for compiling it, and even source control methods for resolving conflicts when a developer checks in his working copy of source code.

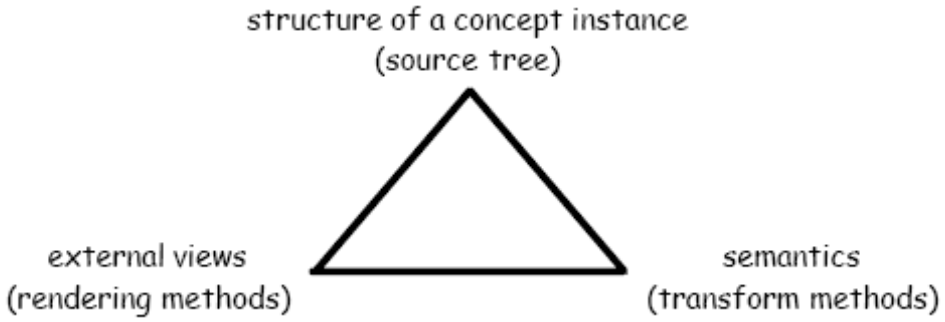


Figure 3-6: Separation of structure, external representation, and semantics

An intention can have more than one rendering method. The appropriate rendering method can be chosen based on the context or user selection. An intention can also have more than one reduction or source control methods. This separation of structure and external representation and semantics is illustrated in Figure 3-6.

3.5.3 Invasive Software Composition:

Standard composition treats components as immutable black-boxes and plugs them together as they are. Invasive Software Composition (ISC) goes one step further and transforms components when they are embedded into a reuse context. Because

components can be adapted more appropriately to reuse requirements they can better be reused.

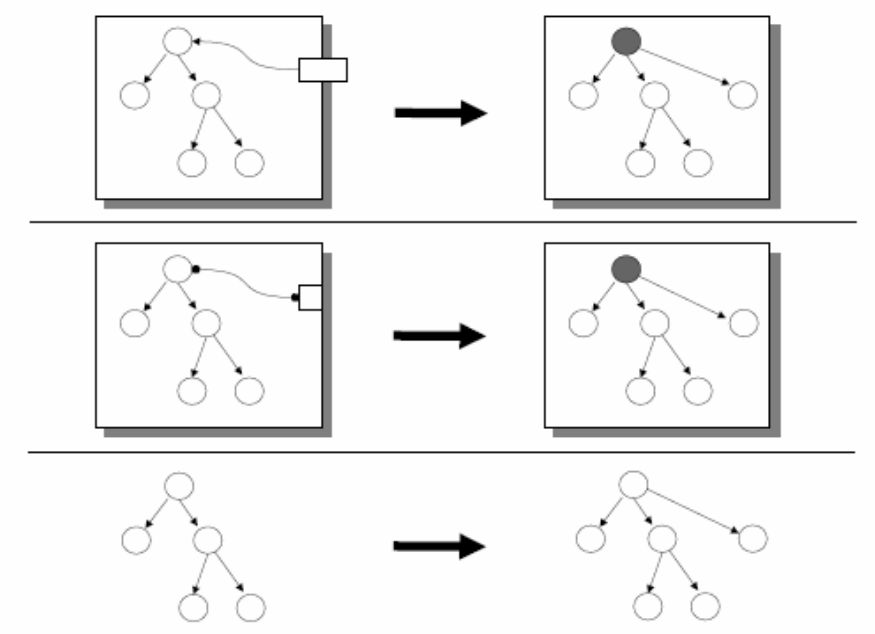


Figure 3-7: Composers can work at different levels of abstraction

The transformation process is applied at well defined places in the program called hooks. Hooks either refer to a program element or to a position in the program. Hooks can be either implicit (such as method entry, method exit) or declared.

The transformation process is performed via composers (or composition operators). Composers can work at different levels of abstraction; they can operate directly on the program text or some other representation. They can operate on implicit hooks. This level abstracts from syntax elements and provides more abstract concepts which were defined in the programming language entries (such as method entries, super classes …). Composers can also operate on declared hooks. This level incorporates design knowledge of the component. The use of declared hooks allows composition to be carried on based on the semantics of the composed components.

A composition script is a recipe for integrating a system from its components. In a typical composition script, composers are applied to components. A composition system is responsible for executing composition scripts to drive a system out of its components.

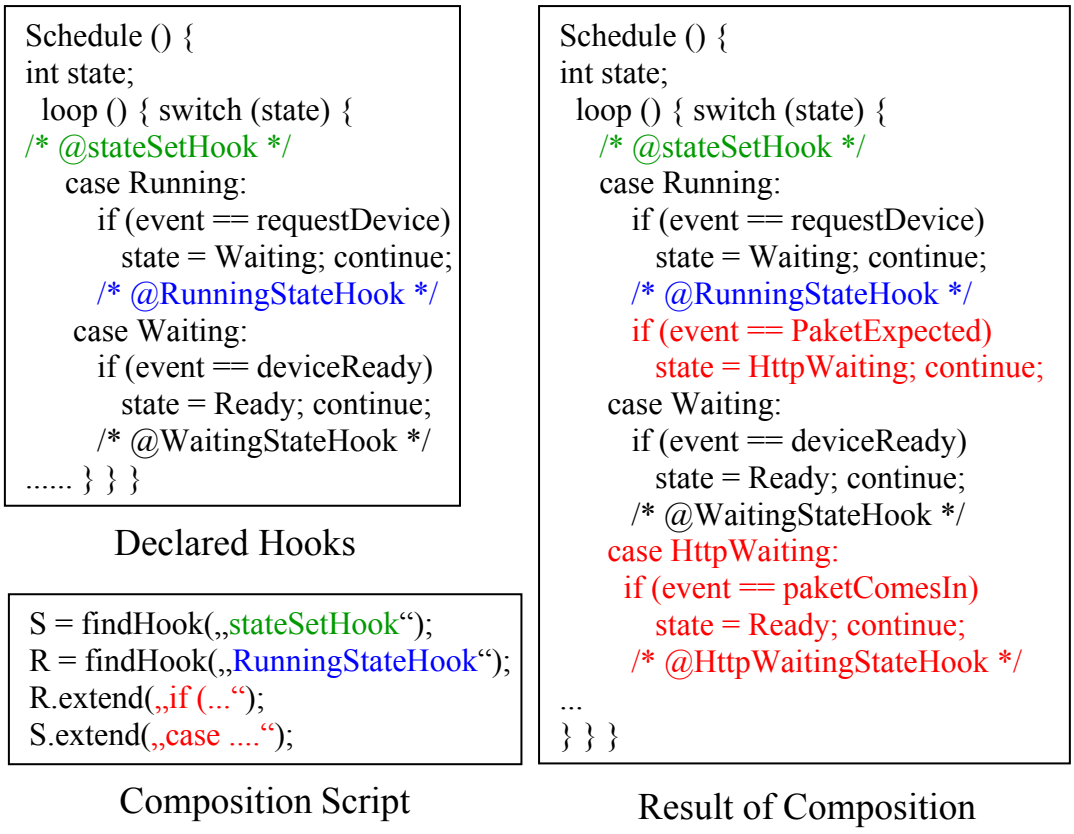


Figure 3-8: Applying a composition script to a component with declared hooks

Invasive software composition can be considered as a unifying theory for software composition. It unifies generic programming, view based programming, and aspect oriented programming. All of these approaches to software composition can be through composers. Further, developers can construct other composers to perform arbitrary composition operations.

3.5.4 Model Driven Architectures:

Model Driven Development (MDD) calls for the use of models of software artifacts in software development. The rationale behind this is that software artifacts have became too complicated and contain too many implementation details that makes them less reusable. A model typically captures the relevant features of a certain artifact or a set of artifacts. Models have been used for a while to support the development. In MDD, models are used in the development. For example, source code can be generated out of models.

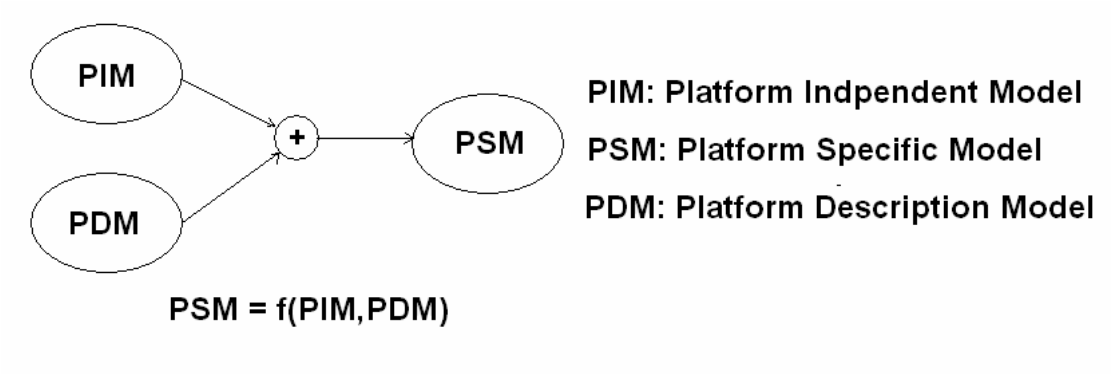


Figure 3-9: Relations between PIMs, PSMs, and PDMs

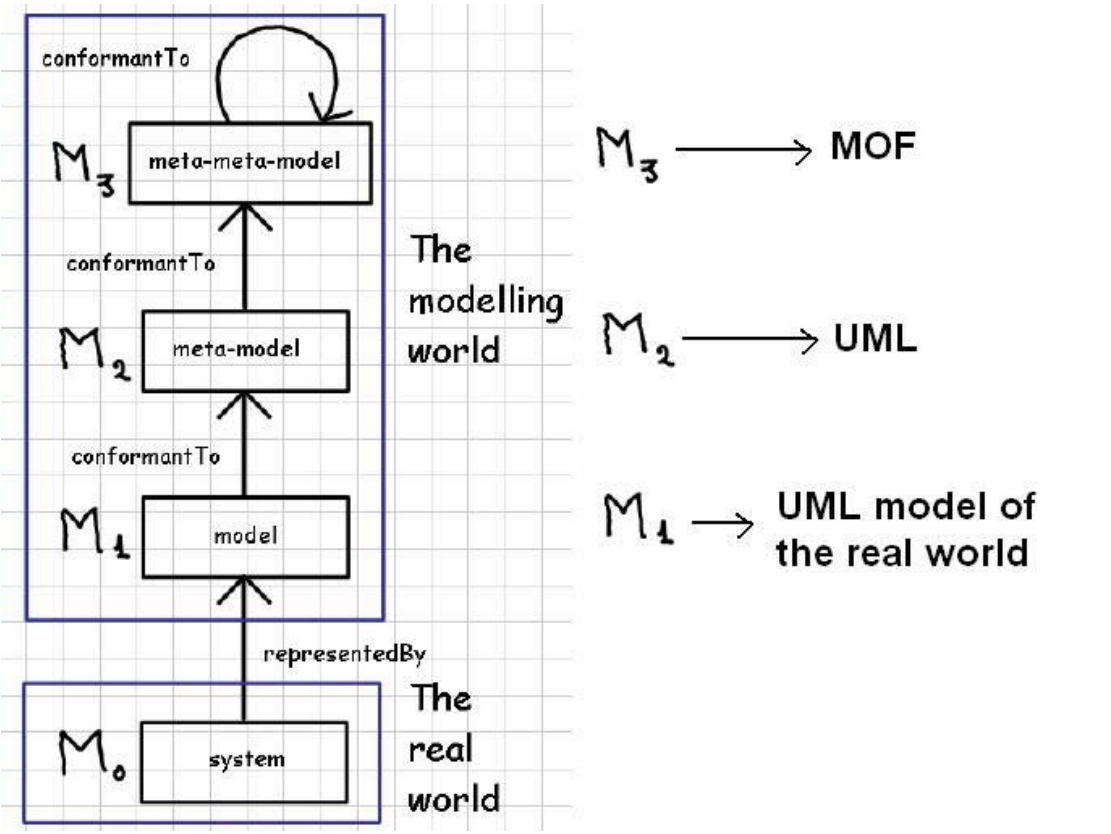


Figure 3-10: The four modeling levels

Model Driven Architecture (MDA) is an MDD methodology proposed by the Object Management Group (OMG). In MDA, models use UML-like languages (i.e. conform to the Meta Object Facility (MOF) which is the meta-model of UML. Figure 3-10 shows the four modeling levels). Models in MDA fall in one of three categories: Platform Independent Models (PIM), Platform Specific Models (PSM), and Platform Description Models (PDM). Their relationship is illustrated in Figure 3-9. A PIM such as user interaction model can be reused across a number of platforms (e.g. desktop,

web, etc) because it filters out the complexities related to the implementation. As a final design step, a PIM is merged with a PDM to get a PSM.

Models in MDA can be mapped to other models. There is a standard for model mapping called Query Views and Transformation (QVT). QVT is still under development. Therefore, tools use ad-hoc solutions to implement these model mappings. There are two types of mappings: horizontal mappings, vertical mappings. Horizontal mappings are mappings between models on the same level of abstraction. They can be used for merging or optimizing models. Vertical mappings are mappings between models at two different levels of abstraction. Refining mappings are mappings of models to other models at a lower level of abstraction. Abstracting mappings are mappings of models to other models at a higher level of abstraction.

CHAPTER 4: PRINCIPLES FOR DESIGNING INTENTIONAL METAPROGRAMMING SYSTEMS

This chapter investigates a number of best practices for developing metaprograms. These best practices are used as a basis for designing and evaluating metaprogramming systems; it is considered an advantage of a metaprogramming system to enforce or at least support a best practice.

4.1 SEPARATION OF CONCERNS:

As seen from chapter 2, a metaprogram typically handle a number of concerns. Allowing developers to separately specify these concerns shall help reducing the overall complexity of specifying metaprograms as well as improving their modularity.

A possible way to enforce the separation of concerns in metaprograms is to express each of them using a programming language tailored for expressing that concern. Moreover, there could be more than one programming language tailored for expressing a certain concern in different modes of usages. For example, consider the concern of data acquisition. We may employ concrete patterns for acquiring localized data and at the same time employ abstract syntax patterns for acquiring non-localized data.

4.2 USE OF CONCRETE OBJECT SYNTAX (BY EXAMPLE APPROACH):

Human developers tend to use examples to describe programming tasks to other developers. Documentation of libraries contains snippets showing how to use certain functionality. Best practices documents do also use examples to demonstrate ideas such as design patterns.

```
public <t> get<v1>(){
        return <v2>;
}
```

Figure 4-1 Getter Template

A by example approach, such as templates, can be used for code generation as well as source code query [MetaJ]. Templates can be considered as simple documents with placeholders. Templates can be used for code generation simply by filling these place holders. They can also be used for source code query by matching templates to existing source code. Query results are those elements that match template placeholders.

```
class TestGettersAndSetters{
        public int x;
        public int y;
        public int z;
        private int w;

        public int getX(){
                return x;
        }

        public void setX(int x){
                this.x = x;
        }

        public int getZ(){
                return x/y;
        }

        public int getW(){
                return w;
        }

        public int setW(int w){
                this.w = w;
        }
}
```

Figure 4-2: TestGettersAndSetters Class

Consider the Getter Template shown in Figure 4-1 when used to query the "TestGettersAndSetters" class shown in Figure 4-2, the results would be {<t=int, v1=X, v2=x>, <t=int, v1=W, v2=w>, <t=int, v1=Z, v2=x/y>}.

4.3 EXPLICITLY SPECIFYING SYNTACTIC CONSTARINTS:

Syntactic constraints take the form of assigning syntactic roles to pattern tokens. As seen from the motivating example in chapter1, it is necessary to be able to

employ syntactic constraints in data acquisition. It was also discussed in chapter 2 that abstract syntax patterns are more effective than concrete syntax patterns (or templates) in specifying syntactic constraints. However, concrete syntax patterns are much easier to express because abstract syntax patterns require developers to assign a syntactic role for every token.

A good compromise is to use a concrete syntax pattern as a starting point for specifying an abstract syntax pattern. And then have syntactic roles assigned to its tokens. This can be achieved through parsing concrete patterns. Developers are sometimes required to provide extra information to guide the process of parsing concrete patterns.

```
Pattern p = new Pattern();
Field f = new Field("int <x>;");
p.add(f);
```

Figure 4-3: Converting a Concrete Pattern to Abastract Pattern

Figure 4-3 shows the process of converting a concrete pattern ("int <x>";) into an abstract pattern. As it can be seen, it was mentioned that the concrete pattern constitutes a field name. This information is used by the parsing algorithm to assign syntactic roles for tokens of the concrete pattern.

4.4 EXPLICITLY SPECIFYING SEMANTIC CONSTRAINTS:

Specifying semantic constraints using lexical or syntactical notations can be tedious and error prone. Suppose that we need to specify that "v2" must be of type "t". This can be done as in the Enhanced Getter Template shown in Figure 4-4.

```
<t> <v2>;
public <t> get<v1>(){
        return <v2>;
}
```

Figure 4-4: Enhanced Getter Template

Problems arise because there is more than one way to make "v2" of type "t" and even the line "<t> <v2>" can appear in more than place (e.g. before or after the

method). When specifying semantic constraints using lexical or syntactical notations developers are enforced to provide a comprehensive list of situations that can satisfy the constraint. Returning to the previous example, problematic situations that might occur include: What if the variable was defined in a super class? What if there were other statements between the variable definition and the method definition? What if the variable was defined after the method definition?

Semantic constraints can also be useful in code generation as well; they can be used to validate or filter the values supposed to fill the template placeholders. This can be extremely useful in case these values were collected from different sources and need to be filtered. Semantics are best described using predicates such as 'type(v2)=t'.

4.5 EXTENSIBILITY:

As seen from the motivating example in chapter 1, it is necessary to be able to employ constraints on design time knowledge in data acquisition. "v2 = CapitalizeFirst (v1)" is an example of a constraint on design time knowledge that can be added to the getter template specified in Figure 4-1.

In order to enable developers to employ such constraints in data acquisition, metaprogramming systems should provide developers to define their predicates.

4.6 NO GET IT RIGHT THE FIRST TIME:

Developers would typically think of developing metaprograms that are guaranteed to produce syntactically and semantically correct object programs. Clearly, such a constraint would greatly increase the difficulty of a naturally difficult problem. Even human developers frequently make mistakes during the development process and use tools to detect these mistakes. We believe that such a constraint would be extremely important in case these metaprograms were used for automated optimization. But, when metaprograms are used to support the development process, relaxing this constraint a bit would ease the development of metaprograms as well as improving their modularity.

For example, suppose that we need to add getters for all public member of the TestGettersAndSetters class. A straight forward solution would be to query the class for all of its public fields as well as their types. Then use the Getter Template to generate getters for all of these public fields. Such a solution will result in adding the

following methods to the class: getX, getY, getZ. Clearly, this will yield two duplicated methods getX, getZ.

A developer adopting a get it right the first time approach will try to solve the slightly different and more complex problem of adding getters to all public members of the class provided that the class doesn't have a method named get<Varname>. Of course, this approach will yield a more complicated query. It will also introduce another problem; after applying the add getters transformation, A developer may be deceived by the method named getZ.

The alternative approach is to leave the simple add getter metaprogram intact. Following its execution, a duplicate method exception is raised. The handler can rename one of the methods, or even delete one of them.

The second approach keeps the original metaprogram in a much simpler form. It also provides greater opportunity for reusing parts of the metaprogram, since inconsistency detection as well as exception handling subroutines can be used in conjunction with other metaprograms.

4.7 NOT FULLY AUTOMATED:

In some cases, expressing queries to object programs might be extremely difficult; this is the case for queries that must employ design time knowledge. The extreme difficulty might not be justified by the benefits gained form the metaprogram. In such cases, it might be a good decision to pop up a dialog box to the developer asking him to manually collect certain information form the source and then use this information in the rest of the metaprogram.

4.8 MODULARITY AND COMPOSITIONALITY:

Dividing large metaprogramming tasks among a number of small metaprograms can help reducing the overall complexity of such tasks. It also increases the chances for reusing these small metaprograms. The key to enhancing modularity is to have powerful mechanisms for composing and coordinating metaprograms.

4.9 UNIFORM REPRESENTATION OF OBJECT PROGRAMS AND METAPROGRAMS:

Metaprograms are essentially programs. While developing metaprograms, developers might be forced to perform the same rote tasks they perform while developing other programs. Therefore, metaprograms can be employed to generate parts of other metaprograms, compose metaprograms, adapting metaprograms, and to propagate changes.

CHAPTER 5: VIEWS IN METAPROGRAMMING SYSTEMS

This chapter starts with a presentation of our notion of using views to support data acquisition. Then it shows where views fit in the architecture of typical metaprogramming systems. Finally, it gives a discussion of the issues of view representation and view specification.

5.1 NOTION:

"The ability of defining views on program representation can enhance the data acquisition capabilities of metaprogramming systems and allow them to effectively perform global data acquisition using much simpler tools that are suitable only for performing local data acquisition. This way, views enable much simpler expression of the data acquisition concern in metaprograms."

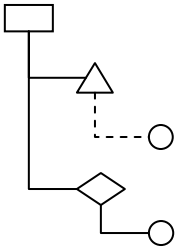
This notion can be seen clearly from figure 5-1. Figure 5-1 (a) depicts a pattern (or template) with a multi-valued attribute which is suitable for performing global data acquisition. However, specifying such patterns as well as building systems that uses them can be somehow problematic; the dashed edge in the figure actually resembles a "path". Therefore, it can be unified with a similar edge or with a number of edges and nodes.

Figure 5-1 (b) depicts the same pattern in (a) except for the dashed edge which is replaced with a solid one. Solid lines are used to draw edges that can be unified with only single edges. This makes the pattern in (b) suitable for local data acquisition.

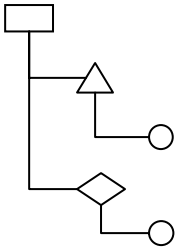
Figure 5-1 (c) depicts an object program fragment that has a match to the pattern in (a) but not (b). (c) can be matched to (a) by matching the dashed edge to the parallelogram and the two edges incident on it.

Figure 5-1 (d) depicts a view on the object program fragment in (c). The view depicted in (d) abstracts some of the details (i.e. the parallelogram). It should be clear that the view in (d) matches both the patterns in (a) and (b). This means that the simpler pattern in (b) can be replace the more complex pattern in (a).

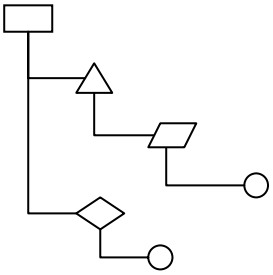
Figure 5-1 (e) depicts another view on the object program fragment in (c). The view depicted in (e) contains an extra localizing edge. Again, the view depicted in (e) can be matched to both the patterns in (a) and (b).



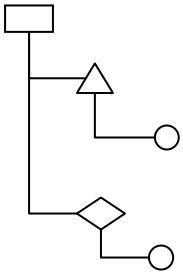
(a) A coarse grain Pattern with a multi-valued variable.



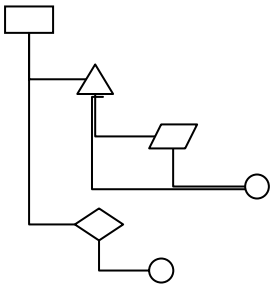
(b) Same Pattern with no multi-valued variables.



(c) An object program with a possible match.



(d) A view of the object program with some abstractions.



(e) A view of the object program with an extra localizing edge.

Figure 5-1: Using Patterns in Data Acquisition

5.2 VIEWS IN METAPROGRAMMING SYSTEMS:

To show where views fit in a typical metaprogramming system, Figure 5-2 gives a comparison between the architecture of a typical metaprogramming system, in (a), and another metaprogramming system that employs views, in (b).

Figure 5-2 (a) depicts a typical metaprogramming system; source files are syntactically and semantically analyzed and the results of the analysis are stored in an intermediate representation. Code for both the data acquisition and modification concerns in metaprograms operates directly on this intermediate representation.

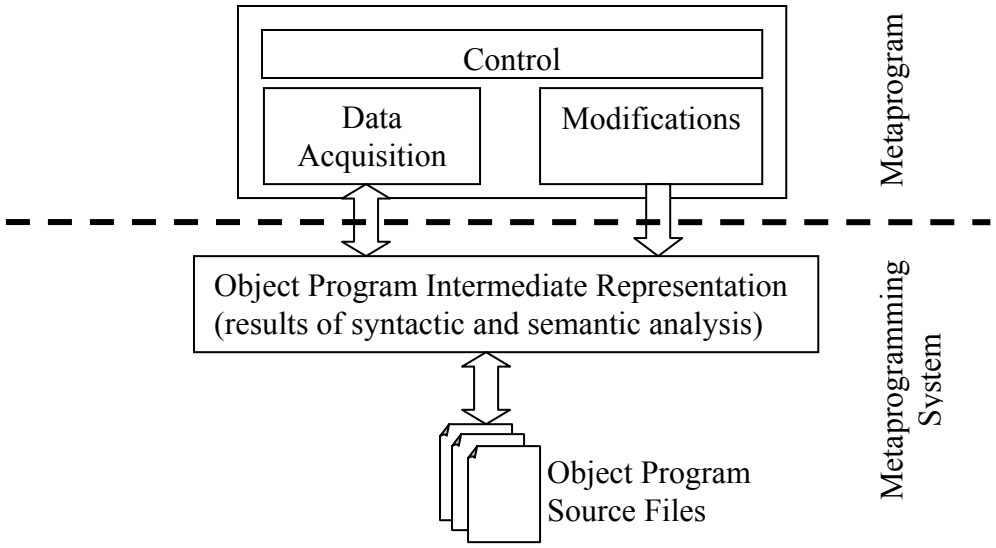


Figure 5-2 (a): A Typical Metaprogramming System without Views on Program Representation

Figure 5-2 (b) depicts a metaprogramming system that employs views. Source files are also syntactically and semantically analyzed and the results of the analysis are also stored in an intermediate representation. The intermediate representation is further modified by abstracting away some of its elements and adding extra localizing edges according to some "view specification" forming a view on the representation. The formed view is stored in a "view representation". Code for the data acquisition concern in metaprograms operates on the formed view rather than on the intermediate representation.

However, code for modifications still has to work on the intermediate representation because the view might contain some elements that does not map directly to source elements. Therefore, operations on these elements cannot be mapped directly to operations on source elements.

The rest of this chapter will be devoted to a discussion of how views can be represented and how they can be specified.

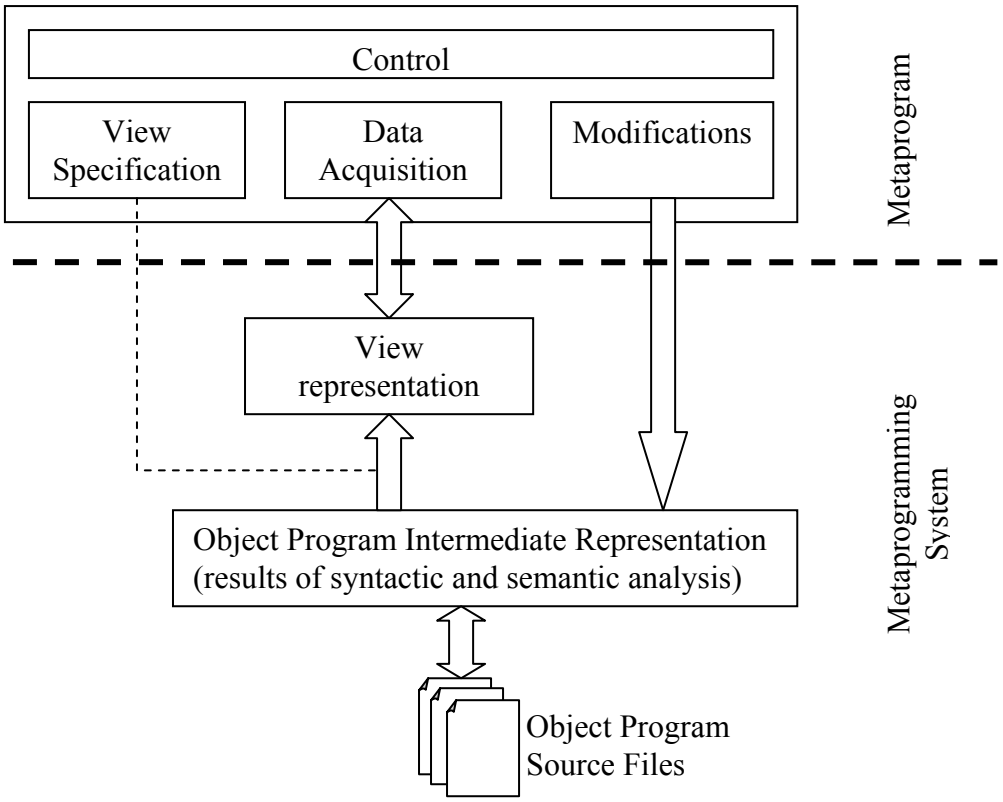


Figure 5-2 (b): A Metaprogramming System with Views on Program Representation

5.2.1 View Representation:

The proposed suitable representation to views is attributed labeled graphs. Node attributes and edge labels allow for easier and more powerful selection of nodes and edges during data acquisition. Attributes and labels can have their values derived from non-local source elements, thus enabling global data acquisition.

The representation should allow attributes to be stored as functions. Storing attributes as functions eliminates the overhead of evaluating these functions every time the view is calculated. For the same reason, edges can also be stored either as functions or as predicates.

The representation should allow the user to store an arbitrary number of nodes, edges, attributes, and labels of arbitrary types. View need also to be updated whenever the underlying intermediate representation is modified. The simplest way of doing this is to recalculate the whole view.

5.2.2 View Specification:

To specify a view, it is needed to describe what nodes, edges, attributes, labels are included and what the values of both attributes and labels are. Views can be specified using traversals and pattern matching, attribute grammars, and even rewriting.

Traversals and pattern matching can be used to specify views in two modes: forward and backward. In the forward mode, the intermediate representation is traversed and its elements are filtered according to some criteria. Elements satisfying the filtering criteria constitute the view. Developers need to specify the traversal strategy and the filtering criteria. However, it might be enough to use a fixed traversal strategy (e.g. Depth first search) and require the developers to specify only the filtering criteria. In the backward mode, the view must be somehow ready except for certain "holes". A set of custom traversal of the intermediate representation are executed to calculate the values that shall fill the "holes". The main drawback of the backward mode is that developers must be aware (at least to some extent) with the structure of the intermediate representation. Adaptive programming [] allow developers to specify their traversals in a structure independent manner. It should also be noted that the two modes are not mutually exclusive. For example, the nodes of a view can be specified using the forward mode while the attributes, edges, and labels are specified in the backward mode.

Custom traversals can be encapsulated in form of functions or predicates (i.e. Boolean functions). These functions and predicates can be specified using a general purpose programming language which allows for adding some extra logic to the traversals. These functions or predicates can be stored as is to the view. These stored functions and predicates can serve as attributes, edges, or labels.

In attribute grammars, attributes are assigned to nodes in the view. For each of these attributes, a specification of how to calculate its value based on values of other attributes belonging to the same node, its parent, or its children. Developers needn't specify the order of evaluation of attributes. Attribute grammars are suitable for global data acquisition.

A sequence of rewriting steps can also be applied to the view to add, delete or modify any of the nodes, edges, attributes or labels. To specify a rewriting rule, one

has to specify its LHS, RHS, and in some cases the rule application conditions or the order of rule execution.

CHAPTER 6: PROPOSED SYSTEM

This chapter proposes a metaprogramming system that employs views. This chapter starts with a description of the responsibilities and interactions between the three subsystems comprising it. Then, the implementation of each of the three subsystems is described. Finally, a case study is given to show that the proposed system can handle non-trivial metaprogramming tasks.

6.1 SYSTEM ARCHITECTURE:

The proposed system consists of three subsystems: Metaprogram handling subsystem, view calculation subsystem, and source code analysis subsystem. The three subsystems and their interactions are illustrated in Figure 6-1.

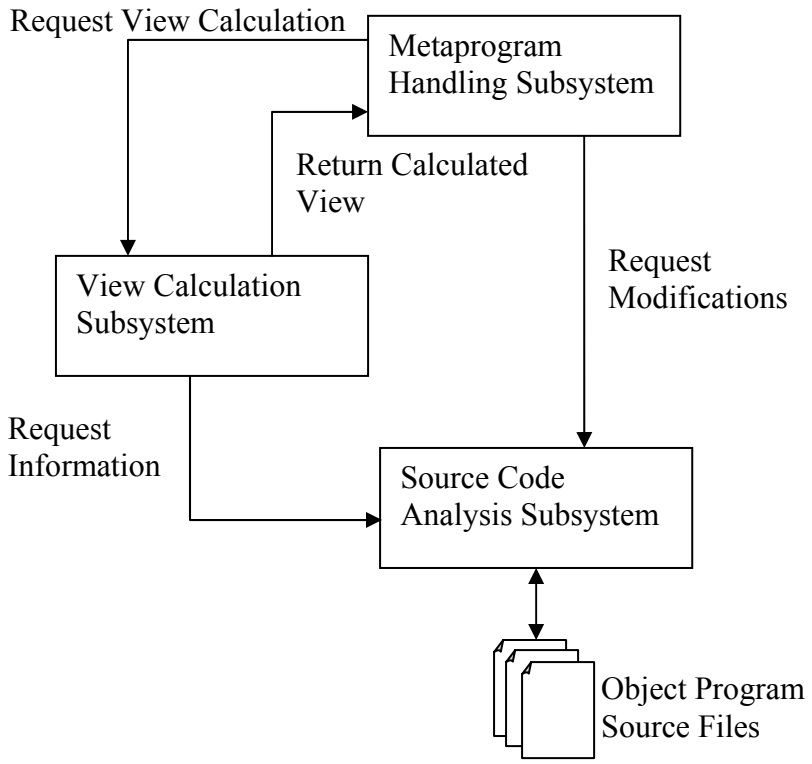


Figure 6-1: Proposed System Architecture

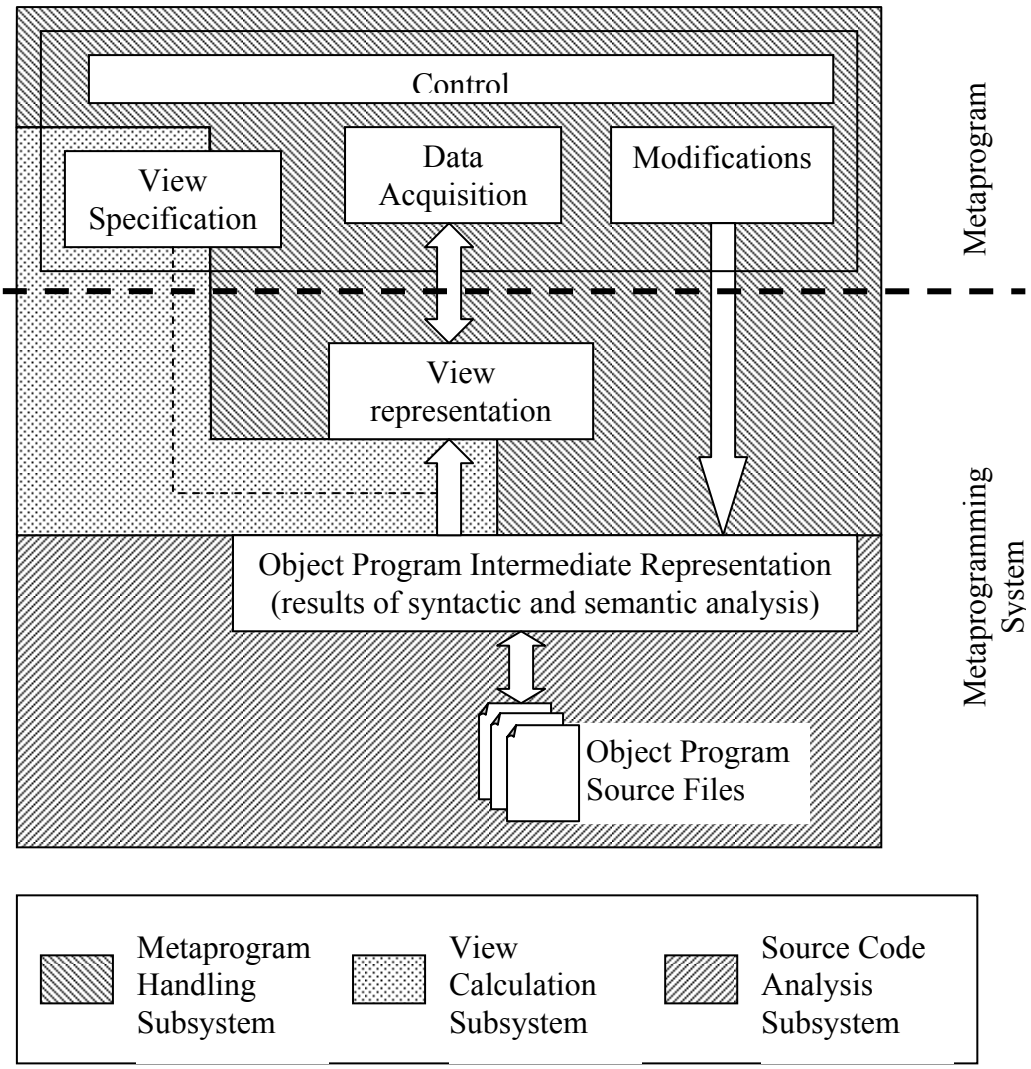


Figure 6-2: Functions of subsystems

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

7.1.1 Metaprogramming:

Employing metaprogramming to support software development can help increasing levels of intentionality, reuse, and agility, thus helping developers to effectively deal with larger and more complex applications and changing requirements.

Developers currently benefit form limited metaprogramming facilities such as code generators, intentions, composers…etc. In the short term, developing metaprogramming technologies shall enable much easier construction of such tools. Ultimately, the development of metaprogramming technologies shall enable ordinary developers not only to program but also to metaprogram.

7.1.2 Metaprogramming systems:

A framework for evaluating metaprogramming systems has been proposed. The framework is based on a set of best practices; it is considered an advantage of a metaprogramming system to enforce or at least support best practices.

7.1.3 Views:

Views have been proposed to support best practices as required in our evaluation framework. Views allow easier specification of the data acquisition concern in metaprograms. A proof of concept implementation of a metaprogramming facility that employs views has also been carried out.

7.2 FUTURE WORK

7.2.1 Multiple views:

Our proposed system allows developers to construct a single view and perform any number of data acquisition operations based on it. It seems a good idea to allow developers to have multiple views.

7.2.2 Updatable views:

Our system allows developers to use views for data acquisition only. It seems a good idea to allow developers to modify views then have these modifications reflected to object programs source.

7.2.3 Intermediate processing concern:

An additional processing phase can be added between the data acquisition phase and the modifications phase. This additional processing phase may constitute a separate concern. This processing phase is responsible for preparing the acquired data. For example, a list of names may need to be sorted or aggregated. The larger the amount of collected data, the more important the concern of data preparation becomes.

Without separating this concern, data preparation can be merged into the data acquisition phase (i.e. all sorting, aggregation is performed during pattern matching or the traversal) this adds complexity to a naturally complex process. Because now it is not only required to analyze programs but also to produce the results in its final form. Data preparation can be also done once after collection. It can also be done on demand (i.e. later on when certain piece of data is required, preparation can be done on the fly)

7.2.4 Dynamic metaprogramming:

Throughout this work we focused on static metaprogramming. It seems a good idea to explore the possibilities of extending this work to dynamic metaprogramming; the later the binding the greater the potentials [sebesta's book].

REFERENCES

[1] Software factories book

[2] Second reference

[3] Third Reference

ملخص

الأكاديمة العربية للعلوم و التكنولوجيا و النقل البحرى

كلية الحاسبات و تكنولوجيا المعلومات

A Multi-Paradigm Metaprogramming Facility for Automating Software Development Activities

رسالة مقدمة لقسم علوم الحاسب
ضمن متطلبات نيل درجة ماجيستير علوم الحاسب

مقدمة من

أحمد عبد المحسن أحمد حسن

بكالوريوس علوم الحاسب و التحكم الآلى 2003
كلية الهندسة جامعة الأسكندرية

إشراف

أ.د. أمين أحمد فهمى شكرى
أستاذ ***** - قسم هندسة الحاسبات و النظم – كلية الهندسة جامعة الأسكندرية

د. مير أحمد حمزة
أستاذ **** - كلية الحاسبات و تكنولوجيا المعلومات - الأكاديمة العربية للعلوم و التكنولوجيا و النقل البحرى

2006 **