Automatic Detection of Metamorphic Properties of Software

Final Report – Spring 2009

Sahar Hasan – sh2503@columbia.edu

1 Introduction

The goal of this project is to demonstrate the feasibility of automatic detection of metamorphic properties of individual functions. Properties of interest here, as described in Murphy *et al.*'s SEKE 2008 paper "Properties of Machine Learning Applications for Use in Metamorphic Testing", include:

1. Permutation of the order of the input data
2. Addition of numerical values by a constant
3. Multiplication of numerical values by a constant
4. Reversal of the order of the input data
5. Removal of part of the data
6. Addition of data to the dataset

While focusing on permutative, additive, and multiplicative properties in functions and applications, I have sought to identify common programming constructs and code fragments that strongly indicate that these properties will hold, or fail to hold, along an execution path in which the code is evaluated. I have constructed a syntax for expressions representing these common constructs and have also mapped a collection of these expressions to the metamorphic properties they uphold or invalidate. I have then developed a general framework to evaluate these properties for programs as a whole.

2 Approach

My approach has been to identify key features of the code that indicate that the function or application in question upholds (or fails to demonstrate) metamorphic properties, based purely on the code itself. Starting with the permutative property, I studied functions written in Java for merge, selection, bubble, and gap sort, sequential and binary search, numerical algorithms like bisection, inverse iteration for smallest eigenvalues of a matrix from linear algebra, and general mathematical functions like mean, variance and dot (inner) product. I then proceeded to study permutative, additive, and multiplicative properties in Java implementations of machine learning applications from the Weka 3.5.8 toolkit, in particular: Naïve Bayes, support vector machines, C4.5 trees, and k-nearest neighbors.

For the second half of the semester, I developed a syntax for code patterns exhibiting these properties and then turned to compiling sets of expressions and their corresponding properties. The code used for this process was primarily for mathematical calculations and search algorithms, involving integers and arrays of integers, under the assumption that the trends observed for these can be extended to other basic types and data structures, and that the expression syntax developed is general and extensible enough to abstract lower-level programming language details.

Although all code studied is in Java, it is expected that these ideas can be applied to other imperative languages like C. In fact, it may even be preferable to work with code in a common intermediate representation like three-address code to avoid syntactic and semantic peculiarities of any given language.

I used Murphy *et al.'s* "Automated Metamorphic System Testing" approach and testing tool *Amsterdam* for verification and validation of properties inferred from code, as described in the paper "Automatic System Testing of Programs without Test Oracles".

3 Expression Syntax

Simple statements involving input data are rarely enough to determine the validity of metamorphic properties, and in general the task requires knowledge of the context in which input data is being manipulated. For instance, when considering the permutative property, inference is significantly different when operations are applied to only a single element of an input list as opposed to when we iterate over the entire list. In order to adequately and accurately describe the context and conditions under which metaphoric properties hold, I developed a syntax for expressions representing code patterns relevant to this properties.

The three basic elements of an expression are variables, constants, and operations. I have chosen to make a clear distinction between what is considered a constant and a variable in these patterns to allow for the possible difference in inferences of metamorphic properties, corresponding to expression patterns, in both cases. Motivations for this and related issues are discussed in section 7.

3.1 Variables and Constants

A value of a constant, as defined here, does not change within the scope of a block, while the value of a variable does, depending on computations performed in the block itself. From a practical standpoint, the distinction can be made in one of two ways. In the first approach, assuming a certain level of contribution in this process from the programmer, names for constants are annotated with underscores, "_constant", and variable names with dollar symbols, "$variable". Secondly, we can use techniques from

data flow analysis to determine all reaching definitions in a procedure or program; all variables on the left-hand side of definitions that are "killed" are not constants. This information can be stored in a symbol table while translating intermediate representation code into the expression syntax defined below[1].

The letter 'c' denotes constants in pattern expressions and variables are denoted by the letter 'v', followed by an optional subscript for uniqueness. For example, variables are v, v_1, v_2 and constants are c, c_1, c_2.

3.2 Operations

Based on common program constructs relevant to identifying metamorphic properties, I have defined operations to constitute the primary element of an expression. Variables and constants are defined to be base or simple types of operations.

3.2.1 Basic Operations

- **loop(start, end)**

The loop() operation is equivalent to a "for x in range" statement. For example, in Java the following for-statement can be translated using the loop() operation as follows:

```
for(int i = 0; i < n; i++){…} → loop(0,N)
```

Additionally, we can optionally assign a label to the loop for use in subsequent parts of the expression by adding "var=" as a prefix:

```
for(int i = 0; i < n; i++){…} → i=loop(0,N)
```

- **compare(operation1, operation2, optional comparison operator)**

The compare() operation corresponds to an if-statement. The parameters of the compare() operation describes two operands and operator for a boolean expression as "operation1 operator operation2", such that a translation from Java would be of the form:

```
if ( x < y ){…} → compare(x, y, <)
```

compare() operations can be augmented for compound boolean expressions involving logical AND (&&), OR (||), and NOT (!) if necessary as follows:

```
if ( x < y && x > z ){…} → compare(x, y, <) and compare(x, z, >)
```

- **assign(operation1, operation2)**

1

The assign() operation corresponds to an assignment statement, “operation1 = operation2”

```
x = 4 → assign(x, 4)
```

- **aop(operation1, operation2, optional operator)**

The aop() operation describes arithmetic operations of the form “operation1 operator operation2”. For instance, we can translate the following statement fragment:

```
y + z → aop(y, z, +)
```

For a complete expression statement in Java, we can employ the assign() operation:

```
x = y + z → assign( x, aop(y, z, +) )
```

- **index(operation1) : #identifier**

The index() operation denote indexing into an element of an array or list at the index number denoted by ‘value’. The second portion of the statement denotes which array or list is being indexed, where the hash symbol indicates a unique, distinguishable identifier. In Java, a given array index can be translated as follows:

```
arr[i] → index(i) : #arr
```

- **call(identifier, optional operation1 … operationk)**

The call() operation represents a function call, where the identifier is the function call, and the optional and variable list of operations (0 to k) are the parameters passed into the function. In Java, a function call can be translated as follows:

```
sequentialSearch(array, target); → call(sequentialSearch, array,
target)
```

We can assign the return value of a function call to a variable as follows:

```
result = sequentialSearch(array, target); → assign(x,
call(sequentialSearch, array, target))
```

- **return(operation1)**

The return operation represents a return statement. Returns are implicit in the last element of an expression statement, but can be specified explicitly if different control flow is relevant to a given pattern. For example:

```
if(x == 1) return 1;
else return 0;
→ (compare(x, 1, ==) : return(1) ) ; return(0)
```

3.2.2 Derived Operations

For the sake of convenience, additional operations can be derived from the basic operations above:

- **sum(optional operation1, operation2)**

The sum() operation describes an augmented assign statement "operation1 += operation1", where the plus operator, and subsequently the word "sum", are taken to be generic operators for any arithmetic operation. It is equivalent to the following Java code and translated expressions:

```
x += a[i] → sum(x,index(i):#a) → assign(x,aop(x,index(i):#a,+))
x = x-a[i] → sum(x,index(i):#a) → assign(x,aop(x,index(i):#a,-))
```

- **swap(operation1, operation2)**

The swap() operation represents a common code fragment used in sorting algorithms. It is equivalent to following set of Java statements and corresponding translated expressions:

```
temp = a[i];  → set(temp, index(i) : #a)
a[i] = a[j]; → set(index(i) : #a, index(j) : #a)
a[j] = temp; → set(index(j) : #a, temp)
where i ≠ j
```

- **update(operation1, operation2, optional comparison operator)**

The update() operation is a equivalent to updating or setting a variable conditionally. The statement is of the form "if(operation1 compare_op operation2) {operation2 = operation1;}". For instance, the following set of Java statements and translations are equivalent:

```
if( y > x){
      x = y;
}

→ update(y, x) → compare(y, x, >) : assign(x, y)
```

3.3 Expression statements

An expression statement is comprised of basic and derived operations or elements delimited by colons. An expression is of the general form '*operation1 : operation2'*, which can be read as "operation1 applied to element2" or "operation2 subject to element1", where 'operation' and 'element' are interchangeably used for semantic convenience. For instance, the statement "loop(0,N) : aop(index(i) : #array1, index(i) : #array2)" can be interpreted as "an arithmetic operation on array1[i] and array2[i] subject to a loop-for-all", or equivalently "a loop-for-all applied to an arithmetic operation on array1[i] and array2[i]". Thus, the colon describes nesting and scoping

between blocks of code. A semi-colon can be used to delimit operations at the same nesting level.

Labels denoted by ":=" can be applied to the beginning of any expression to indicate the starting instruction for a new procedure, and to reference function calls within other expressions. For instance:

```
public int foo(){
      int x;                                  → foo:= assign(x, call(bar))
      x = bar(x);
      return x;
}
public int bar(int a){

      a = a + 1;                              → bar:= assign(a, aop(a, 1, +))
      return a;
}
```

4 Expression Pattern Mappings

Using the expression syntax defined in the previous section, I have constructed pattern expressions from actual Java code, and have mapped these to the metamorphic properties they uphold or "kill". The listed expression patterns below are not exhaustive, but lay the groundwork for the identification of more code patterns and corresponding metamorphic properties as larger varieties of code are studied and possible machine learning techniques are employed.

In the properties column of the table, letter P stands for the permutative property, while A and M stand for the additive and multiplicative property respectively. A '!' preceding any of the letters means that the property does not hold for a given pattern. Multiple patterns in a row marked by '->' means that all such variations of the pattern exhibit the same set of properties.

#	Pattern	Properties
1.	-> index(v) : #id -> index(c) : #id -> index(aop(…)) : #id *Example:* list[value-1];	!P, A, M
2.	-> aop(index(i) : #id, index(i+1) : #id) -> aop(index(i) : #id, index(j) : #id) i≠j *Example:* (list[value-1]+list[value])/2;	!P, A, M

3.	-> assign(v, aop(c1, c2)) *Example:* x = (b – a) / a;	!P, A, M
4.	-> aop(index(i) : #id1, index(i) : #id2) *Example*: x = v[i] + u[i];	!P, A, M

5.	-> i=loop(0, N) : sum(index(i) : #id)
	Example:
	for(int i = 0; i < n; i++)
	sum += list[i];
		P, A, M

6.	-> i=loop(0, N) : sum(aop(index(i) : #id1, index(i) : #id2)) *Example:* x += v[i]*u[i];	!P, A, M
7.	->i=loop(0, N) : sum(aop(index(i) : #id, c)) -> i=loop(0, N) : sum(aop(aop(index(i) : #id,c),c)) *Example:* for(int i = 0; i < n; i++) sum += Math.pow((list[i] – mean), 2);	P, A, M

8.	-> i=loop(0, N) : assign(index(i) : #id1, aop(index(i) : #id2, c)) *Example:*
	for(int i = 0; i < n; i++)
	u[i] = x[i] /length
		!P, A, M

9.	-> i=loop(0, N) : compare(index(i) : #id, c)
	Example:
	for(int i = 0; i < n; i++)
	if(array[i] == target)
	return i;
		P, !A, !M

10.	-> i=loop(0, N) : update(aop(index(i) : #id, c), v)
	Example:
	for(int i = 0; i < n; i++)
	if((array[i] – target) < max)
	max = array[i] – target;
		P, A, M

11.	-> i=loop(0, N-1) : update(aop(index(i) : #id, index(j) : #id), v)
	Example:
	for(int i= 0; i < n-1; i++)
	if((arr[i] + arr[i+1]) > max)
	max = arr[i] + arr[i+1];
		!P, A, M

12.	-> i=loop(0, N) : assign(index(i) : # id1, index(i) : #id2) *Example:* simple copy operation	P, A, M

13.	-> i=loop(0, N) : sum(index(i) : #id, c)
	-> i=loop(0, N) : sum(index(i) : #id, aop(index(i) : #id, c))
	Example:
	for(int i = 0; i < n; i++)
	a[i] -= value;
		!P, A, M

14.	-> i=loop(0, N) : compare(index(i) : #id, c) : assign(index(i) : #id, c) *Example:*
	for(int i = 0; i < n; i++)
	if(arr[i] < target)
	arr[i] = target;
		!P, !A, !M

As pattern 10 and 11 indicate, we can allow for as many nested arithmetic operations (aops) without affecting the metamorphic properties of the code, provided that the at least one of the operands of the aop is a constant or variable and not an index into another list or another index into the same list.

5 Analysis Framework

Translated code expressions can be matched against a given collection of pattern expressions in order to determine the metamorphic properties of each expression sequence. However, applying these patterns to code fragments independently and in isolation will lead to incorrect inferences about the properties of the program as a whole. Thus, I have developed a general framework in which execution flow is used to evaluate the propagation or extinction of properties through the various procedures or code blocks of a program.

The model defined here employs techniques and algorithms similar to those used in data-flow analysis. The model defined here is similar to a flow graph used for a data-flow abstraction:

- A program is represented by a directed graph.

- The nodes of a graph correspond to blocks of translated expressions. Code is partitioned into blocks following an algorithm analogous to the partitioning of three-address instructions into basic blocks describe in Aho *et al's*. book "Compilers: Principles, Techniques, & Tools". The basic idea is that control flows from the beginning to the end of the block contiguously, and any procedure calls (or equivalently "jumps in the code") result in a segmentation of the code into blocks starting with the statement jumped to and the statement following the jump. There are directed edges from one node to the other in this situation, and when flow naturally "falls" through to the next node.

- Entry and exit nodes are included in the graph, and correspond to the start of the main() function of a program and the termination of the main procedure respectively.

- In a data-flow schema on basic blocks, every node or block has associated with it an "IN" and an "OUT" set of program values or states. In this case, the values are the he metamorphic properties exhibited by that block of code. Therefore, the domain of this data-flow schema is the set of metamorphic properties and analysis will proceed in the forward direction.

- The universal set U is defined to be the set of all metamorphic properties. That is, U={permute, add, multiply, invert, reverse, remove, augment}. In this framework, I assume that the program starts out by exhibiting all the metamorphic properties, thus the entry node is initialized with the universal set. In essence, as control flows through the blocks of code, properties are "knocked-off" if they are guaranteed not to hold.

- There can be several possible execution paths as depicted by the flow graph, but the invalidation of a metamorphic property along even one path is sufficient to invalidate the property for the entire program. As a result, I define IN[B] for any block B to be:

$$IN[B] = \cap_{P \text{ predecessors of } B} OUT[P]$$

- A block or node in the graph "kills" a metamorphic property when an expression or several expressions in the block match to patterns that do not exhibit that property. The set of all properties thus killed is referred to as the kill set, or $kill_B$, for that block.

- As discussed, control flows straight through the start and end of the block. Thus, the properties at the start and end of a block are related by transfer functions of the form: "$OUT[B] = f_B(IN[B])$ ". The transfer function for this model is:

$$OUT[B] = IN[B] - kill_B$$

The iterative algorithm to compute metamorphic properties is then defined to be:

```
OUT[ENTRY] = U;
for (each block B other than ENTRY) OUT[B] = U;
while (changes to any OUT occur)
      for(each basic block B other than ENTRY){
            IN[B] = ∩P predecessors of B OUT[P];
            OUT[B] = IN[B] – killB
      }
```

Algorithm 1 Iterative algorithm to compute metamorphic properties

After running this iterative algorithm until convergence, the resulting set OUT[EXIT] will contain information about which metamorphic properties hold for the program.

6 Sample Analysis

The following code in fig. 1 defines a class “AnalysisExample” that takes in an array of integers, finds the largest sum of consecutive elements and then uses that sum as a target value for the element in the array closest to it.

```
1 public class AnalysisExample{
2
3     public static void main(String[] args){
4
5           int[] input_array = {11, 7, 21, 5, 6, 14};
6           int answer = findGreatestSumAndClosestValue(input_array);
7     }
8
9     public static int findGreatestSumAndClosestValue(int[] values){
10          int sum = findGreatestSumOfConsecutiveElements(values);
11          int closest = findClosestValue(values, sum/2);
12
13          return closest;
14    }
15
16    public static int findGreatestSumOfConsecutiveElements(int[]
values){
17          int maxValue = -10000;
18
19          for (int i = 0; i < values.length-1; i++){
20           if (values[i] + values[i+1] > maxValue)
21                maxValue = values[i] + values[i+1];
22          }
23
24          return maxValue;
25    }
26
27    public static int findClosestValue(int[] values, int target){
28          int distance = 1000000;
29          int closestIndex = -1;
30
31          for (int i = 0; i < values.length; i++){
32                if (Math.abs(values[i] - target) < distance){
33                      distance = Math.abs(values[i] - target);
34                      closestIndex = i;
35                }
36          }
37          return values[closestIndex];
38    }
39 }
```

Figure 1 Java program to find the greatest sum of consecutive elements in an array and then find the closest value in the array to it.

6.1 Translation

The program in fig. 1 can be translated into the following set of expressions in fig. 2:

```
1  main:= assign(answer, call(findGreatestSumAndClosestValue,
input_array))
2
3  findGreatestSumOfConsecutiveElementsAndFindClosestValue:=
4   assign(sum, call(findGreatestSumOfConsecutiveElements, values)) ;
5   assign(closest, call(findClosestValue(values, sum/2)))
6
7  findGreatestSumOfConsecutiveElements:=
8   i=loop(0, N-1) : update(aop(index(i):#values,index(i+1):#values),
maxValue)
9
10 findClosestValue:=
11  i=loop(0, N) : update(aop(index(i):#values, target), distance)
```

Figure 2 Expression translation for Java program in fig. 1

6.2 Pattern matching

Line 8 in fig. 2 matches the following pattern expression:

```
i=loop(0, N-1) : update(aop(index(i) : #id, index(j) : #id), v)
```

The pattern is mapped to the metamorphic properties <!permute, add, multiply>, thus the expression in line 7 does not uphold the permutative property.

Similarly, line 11 matches the pattern expression:

```
i=loop(0, N) : update(aop(index(i) : #id, c), v)
```

The pattern maps to the properties <permute, add, multiply>, thus the expression in line 11 upholds all three properties.

6.3 Construction of flow graph

The flow graph constructed from the translated expressions is shown in fig. 3:

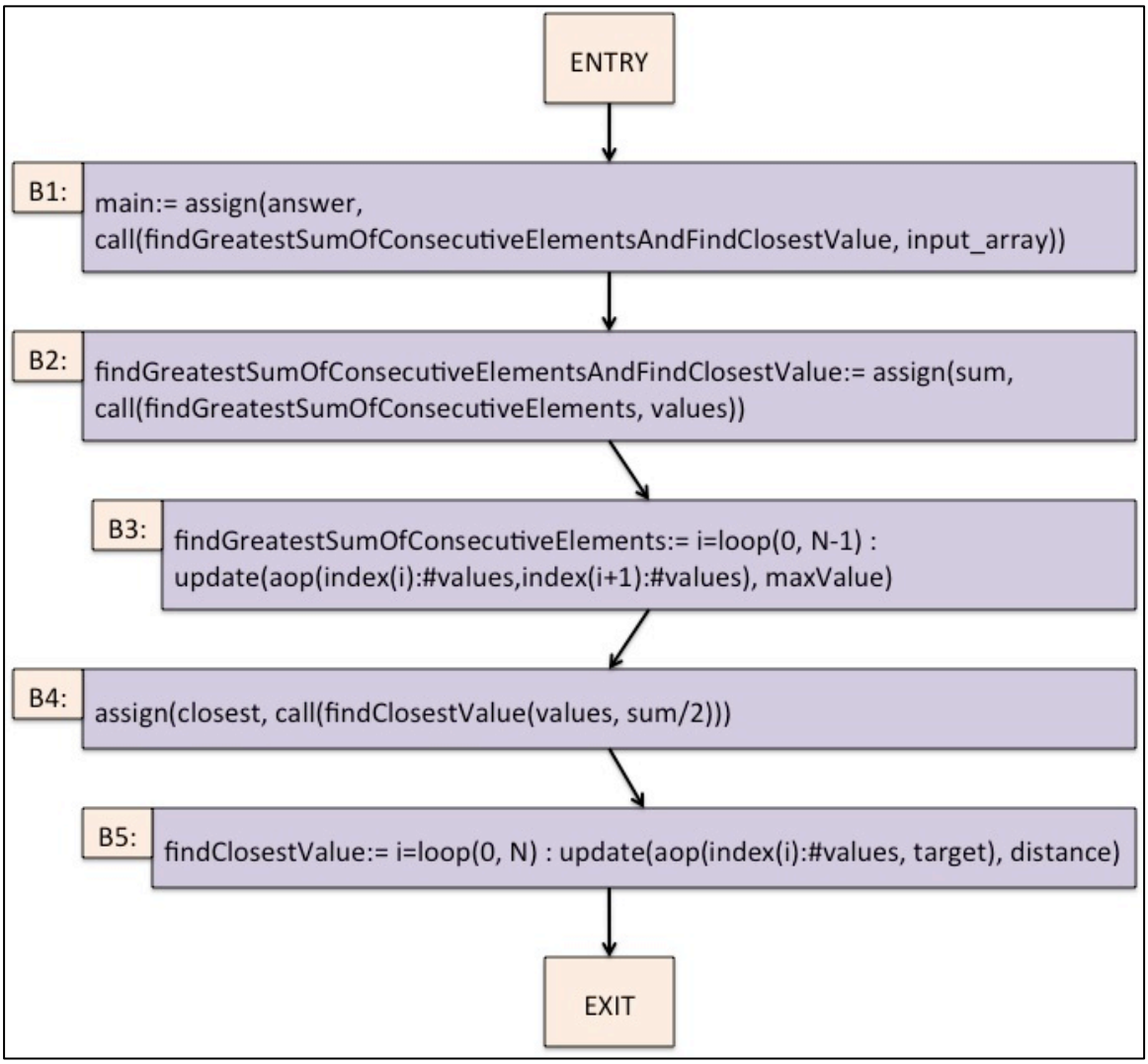

Figure 3 Flow graph of translated expressions

6.4 Data-flow analysis

The seven metamorphic properties can be represented by bit vectors, where bit i from the left represents property i, and the properties are numbered:

1. permute
2. add
3. multiply
4. invert
5. reverse
6. remove
7. augment

Thus, the bit vector for the universal set U is <111 1111>. The kill sets for the blocks corresponding to figure 2 are as follows:

Block B	$kill_B$
B1	000 0000
B2	000 0000
B3	100 0000
B4	000 0000
B5	000 0000

Applying the algorithm defined above to the program "AnalysisExample", the IN and OUT sets for the blocks in the diagram are computed as follows:

Block B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$
B1	111 1111	111 1111	111 1111
B2	111 1111	111 1111	111 1111
B3	111 1111	111 1111	011 1111
B4	111 1111	011 1111	011 1111
B5	111 1111	011 1111	011 1111
EXIT	111 1111	011 1111	011 1111

The algorithm terminates after the second iteration, with no changes to any OUT occurring. The final OUT set for EXIT is <011 1111>, meaning that the program has the additive and multiplicative properties (and possibly the other four properties not studied in this project), but does not have the permutative property.

7 Limitations and Practical Concerns

7.1 Interpretation of properties and the "constant issue"

It is important to define the context in which the metamorphic properties are said to apply when there are multiple inputs, including a mix of lists and constants. For instance, in a binary search algorithm, the additive and multiplicative properties fail if we only consider the array and not the target value, or vice versa. In terms of a naïve form of the syntax for expressions, a fragment of Java code from the binary search algorithm will be translated as follows:

```
if(array[mid] < target)        → compare(index(mid) : #array, target)
      low = mid+1;             MATCH: compare(index(id1) : #id2, id3)
```

The syntax for these pattern expressions is considered naïve because it makes no distinction between the kinds of variables involved in the expression, by "id" we simply mean an identifier or a name.
If we are only considering metamorphic properties with respect to the input array, then corresponding set of metamorphic properties for this expression are {!A, !M}. (The

permutative property is irrelevant here, and would just be “P”. It is not additive or multiplicative since the other operand is not being scaled.

If we consider the metamorphic properties to apply to all input values, then the set of metamorphic properties for that translated expression should be {, !A, M}. The additive and multiplicative properties now hold because we are scaling the “target” value being compared with the array element. Thus, It is tempting to then apply addition or multiplication to all input values, but that isn’t always correct either if, for example, the values being passed in are indices of an array.

Since there is no way to generalize or determine when some or all input values should be scaled, the latter interpretation of “metamorphic properties of the input” should not be adopted. But it is a matter of judgment whether it is more conservative to kill a property or uphold a property; it often depends on how the results of this process are used, that is, whether the results are intended to be informative or corroborative.

Nevertheless, there are improvements that can be made to the syntax to remedy this flaw, namely, that we make a distinction between constants and variables. A value of a constant, as defined here, does not change within the scope of a block, while the value of a variable does, depending on computations performed in the block itself. From a practical standpoint, the distinction can be made in one of two ways. In the first approach, assuming a certain level of contribution in this process from the programmer, names for constants are annotated with underscores, “_constant”, and variable names with dollar symbols, “$variable”. Secondly, we can use techniques from data flow analysis to determine all reaching definitions in a procedure or program; all variables on the left-hand side of definitions that are “killed” are not constants. This information can be stored in a symbol table while translating intermediate representation code into the expression syntax defined below[2].

Now expression patterns can incorporate the distinction expressions of the form:

```
1. compare(index(v1) : #id1, c1)

2. compare(index(v1) : #id1, v2)
```

The property set of expression 1 is {!A, !M} because we can be assured that the value of the variable representing constant c1 is not going to be changing. On the other hand, the property set for expression 2 is {A, M}. This is the best automated decision to make in this situation for one of two reasons: either the value of v2 is being changed by computations and manipulations made involving the input array id1—in which case scaling for addition and multiplication directly applies—or it is changing in some other way that is unrelated to the scale of the input.

---

[2] A.V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, & Tools, 2nd Edition.

7.2 Syntax

The syntax might be a limitation in other ways, in the sense that for more complicated expressions, the nesting nature of the expressions will tend to explode. It may also be impossible to list all the possible patterns if we do a blind translation for code such as the following:

Source:

```
for(int i = 0; I < n; i++){
…
      for(int m = i+1; m < n; m++){
            double l = A[m][i]/A[i][i];
            for(int  j = 0; j < n; i++){
                  A[m][j] = A[m][j] – A[i][j]*l;
            }
      }
}
```

Translation:

```
i=loop(0, N): m=loop(i+1, N):assign(x, aop(index(i):#A, index(j):#A));
j=loop(0, N):assign(index(f):#A, aop(index(f):#A,aop(index(g):#A, l)))
```

where f and g are flattened, row-major representations of index A[m][j] and A[i][j].

The corresponding collection of pattern domains and property ranges can never be this specific, and will need some way to remain general enough, but still be able to recognize that the forward elimination algorithm above has the property set {!P, A, M}.

An ideal solution would be to throw out irrelevant details during the translation phase from source to expression. We can do this by relying on trends that become apparent from the syntax and mappings itself. For instance, nested arithmetic operations between the same element of the same array and a constant or a variable can be grouped together without affecting inferences of metamorphic properties.

7.3 Limitations of a static approach

In the syntax defined above, I have assumed that there is a way to determine whether loops are made over all elements of an input list or array [loop(0,N)], which is fundamental to the permutative property. In practice, this isn't possible when dynamic arrays or lists are used except at runtime because it could depend on user input or could be the result of computations during execution. There are two possible ways to handle this issue. We can leave it to the translation phase to identify the range of a loop by recognizing common colloquialisms for iterations over all elements such as "N", "length", "array.length", "size", "end", etc. Alternatively, we can introduce a certain

degree of involvement from the user by having them annotate the code with the expected range of a loop in a standard format that can be identified during translation.

8 Conclusions and Future Work

In summary, I expect the overall framework for the automatic detection of metamorphic properties of software to be comprised of four major phases or components:

1. Translation: a syntax-directed translation scheme from source language to expressions. The syntax of source expressions should allow the expressions to be built up through a context-free top-down parsing of the source.
2. Regular expression pattern matching of collected pattern expressions with source expressions. During this phase we can map and associate metamorphic properties with every expression.
3. Construction of a flow graph and computation of the kill sets for every block in the graph. The kill sets will simply be the union of all the properties that do not hold for expressions in the block.
4. Data-flow analysis using the algorithm defined in section 5.

Currently phases three and four need only to be implemented, and will work as expected. The primary source of future work is in phases one and two. A proper syntax is essential to both the pattern matching phase and to the construction of a correct flow graph. By using the syntax for expressions defined here, future research could entail expanding the syntax as outlined in section 7.2 and building up the corpus of pattern expressions. With a finalized syntax, the syntax-directed translation scheme can also be implemented. Research can also involve incorporating machine learning techniques into the recognition of pattern expressions once a rudimentary translator is implemented.