Metamorphosis, State Machines, and Object Oriented Design

Peter M. Maurer
Department of Computer Science
Baylor University
P. O. Box 97356
Waco, TX 76798-7356

Abstract

Metamorphic programming is an effective tool for creating efficient and elegant solutions to many programming problems, at least once you get over the shock of seeing code that violates many of the accepted rules of good programming. We have used metamorphosis for many years to solve problems in the logic-level simulation of VLSI circuits. These solutions have provided some spectacular gains in performance, inspiring us to look for metamorphic solutions to other problems. We have found metamorphic solutions to many problems including string searching, sorting, and depth first search, most of which provide performance gains over conventional coding. A few of these solutions are presented here. These programs violate the rules of good programming, but with a few minor compiler enhancements, our programming techniques become clean and well structured.

1. Introduction

"Metamorphic programming can make your code run five to sixty times faster, and is so simple that it will transform your programs into straight-line code. The only catch is, you will have to violate most of the rules of good programming, and you may have to invent one or two new algorithms."

Five years ago I would have laughed at this, but after writing and testing dozens of metamorphic programs, *I believe*!

In metamorphic programming, objects to change their identity during program execution. Function definitions change and sometimes, although rarely, data items change type or become hidden. An object can change its behavior over time to adapt to differing conditions or respond to new needs.

Metamorphosis is an efficient way to handle object states, particularly those states that affect the behavior of the object. Existing algorithms handle state information by using state variables which are decoded to produce the required behavior for a particular state. But this decoding represents a duplication of work! It recovers information that, at one time, was readily available. To illustrate, consider a binary semaphore *S* with two states, 1 and 0. The *P* and *V* operations do two distinctly different things depending on the state of *S*. In state 1 the *P* operation changes to state 0 and the *V* operation is ignored. In state 0, the *P* operation blocks the calling process while the *V* operation either unblocks a process or changes back to state 1. If we initialize the semaphore to state 1 and perform the *V* operation, the state will change from 1 to 0. At the moment of change, the new behavior of the *P* and *V* functions is known. But in a typical implementation this knowledge is deliberately discarded. The state is encoded as a zero or a one, which must then be decoded by generic *P* and *V* functions to determine the correct behavior. It is more elegant and more efficient to have a separate set of *P* and *V* functions for each state. With such functions, it is no longer necessary to test the state, or even to record its value.

Let's assume that a semaphore has pointers to its P and V functions and that run-time binding is done using these pointers. The following code shows the new functions. (This isn't exactly legal C++, but you get the idea.)

```
P0()          P1()
{             {
   P = &P1;      Block Current Process;
   V = &V1;      Queue Current Process;
}             }

V0()          V1()
{             {
   return        if (Process is queued)
}                   Dequeue     &     Unblock
              Process;
                 else       {
                    P = &P0;
                    V = &V0;
                  }
              }
```

When we replace the *P* and *V* functions, we are changing the semaphore's behavior and effectively changing its identity. In state 0 the semaphore is an object that does nothing. In state 1 it is an object that queues and dequeues processes.

2. Metamorphosis and Polymorphism

Although metamorphosis may seem strange, it similar to the polymorphic types used in conventional object oriented programming. Using polymorphism it is possible to process a heterogeneous set of objects without using type-codes or type-decoding[1,2]. Metamorphosis is the extension of polymorphism to dynamic codes.

Metamorphosis can be implemented using mechanisms similar to those used to implement polymorphism. The key to polymorphism is, of course, the virtual function. (The term *polymorphism* can mean many different things, but here we use the term exclusively for types created using inheritance and virtual functions.) Unlike conventional functions, which are bound to their function calls at compile time, virtual functions are bound at run time. In the class definitions below, the pointer variable, *MyPtr*, can point to an object of type *MyPoly* or an object of type *MyDerv*. The executable code assigns a pointer of each type to *MyPtr*, and then calls *MyFunc*. Because *MyFunc* is bound at run-time, the two calls produce different results. The first call prints "Apple" while the second prints "Orange". If the binding had been done at compile time, both function calls would print the word "Apple".

```
class MyPoly
{
   MyPoly * Next;
   virtual void MyFunc( )
   {
      cout<<"Apple";
   }
}
```

```
class MyDerv : public MyPoly
{
   virtual void MyFunc( )
   {
      cout<<"Orange";
   }
}
```

```
MyPoly * MyPtr;
MyPoly * MyPtr;
MyPoly Obj1;
MyDerv Obj2;

MyPtr = &Obj1;
MyPtr->MyFunc( );
MyPtr = &Obj2;
MyPtr->MyFunc( );
```

Dynamic binding is often used to process a heterogeneous collection of objects, as in the following code, which prints the type of each object in a list of objects.

```
Shape * Head;
…
float Total = 0.0;
for (MyPoly * Temp = Head ; Temp != NULL ; Temp=Temp->Next)
{
   Temp->MyFunc();
}
```

In the days before polymorphism, a type-code would have been used to distinguish between *MyPoly*, and *MyDerv*. The loop would decode the type-code to determine the correct *MyFunc* function. Like a state-code, the type-code represents lost information. When *MyPolys* and *MyDervs* are created, the correct procedure for printing the message is well known. Polymorphic types allow the correct *MyFunc* function to be appended to an object when it is created.

Both polymorphism and metamorphic programming allow us to replace explicit codes with subroutine addresses. Because these addresses give us specific behavior, they are significantly more useful than numeric codes.

3. Metamorphosis and Simulation

My students and I were first drawn to metamorphic programming because we wanted to find faster ways to simulate logic-level digital circuits. For a new VLSI circuit, a significant part of the development time is devoted to simulation, and the faster you can simulate a circuit, the more quickly you can bring it into the marketplace. The demand for speed is so overwhelming that it is worthwhile to explore "peculiar" types of programming if there is any chance that they will improve performance. We began exploring metamorphic techniques because we believed they would help us reduce simulation time, but we were amazed when we saw the final results. For most circuits we had a 7x increase in performance. The minimum improvement was 5x, and for one anomalous circuit we had an increase of 60x. We would have been happy with 50%. (See the sidebar for the experimental results.)

The key to the increase in performance was not just metamorphic programming, but a collection of algorithms that were specifically designed for metamorphic programming. We have found that metamorphic programming improves the speed of many algorithms, but a direct translation into metamorphic code usually gives only a modest improvement. As in polymorphic programming, the strength of metamorphic programming lies in its ability to process a collection of heterogeneous objects. We have organized our algorithms to take advantage of this.

Our first step is to translate a logic circuit into a collection of objects. (This is true for *any* simulation.) A circuit, such as that of Figure 1, is transformed into interconnected collection of gates and nets (as wires are called) with an object representing each. Net objects have a *value* element that maintains the state of the circuit. Except for flip-flops, gates have no state and are treated as pure functions. Special scheduling techniques are used to simulate gate and net delays.

Experimental Data

The following table compares the speed of our metamorphic simulator to that of a conventional simulator. Several standard simulation benchmarks are used[9]. The column labeled EVCF (Event Driven, Conditional-Free) gives the times for our metamorphic simulator. The results are expressed in CPU seconds of execution time. The hardware was a SUN 300MHz single processor Ultra SPARC-II with 128MB of RAM. Fifty thousand random input vectors were used for each test.

Circuit	Conventional Event-Driven	EVCF	Speedup
C432	10.8	1.4	7.71
C499	12.1	1.7	7.11
C880	20.2	4.0	5.05
C1355	43.2	5.6	7.71
C1908	82.5	8.1	10.19
C2670	89.3	13.6	6.57
C3540	128.5	15.3	8.40
C5315	252.9	27.5	9.20
C6288	2549.5	42.1	60.56
C7552	396.8	40.2	9.87

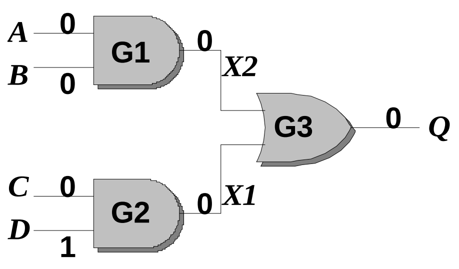

Figure 1. A Sample Circuit.

During simulation, nets that change value are linked into a queue of pending changes. Objects are inserted into the tail of the queue, and are processed when they reach the head of the queue. Each object has one or more functions that change to reflect the state of the object. The simulator traverses the linked list and executes the current function for each object.

The main innovation that led to our dramatic increases in performance was the modeling of *both* gates and nets as state machines. It is obvious that nets have states, because they must have a value of either zero or one. Surprisingly, gates also have states. Compare the two AND gates of Figure 1. If either input of G1 changes, nothing happens.

However, if input C of G2 changes, output X1 changes, and the change propagates through gate G3 to output Q. Clearly G1 and G2 are in different states. We can consider the nets A and B to be state machines that transmit values to G1. In turn, we can consider G1 to be a state machine that that transmits values to X2.

The states of X1 and X2 are important only because of the effect they have on the state of G3. An explicit 1/0 value is not required for these nets, and any convenient method for representing the state will do. It *is* necessary to maintain a 1/0 state code for nets A, B, C, D, and Q because we must examine inputs for changes, and because we must report output values to the user.

Figure 2 shows the state machine for a net, and the data structure used to implement it. The input signal of the state machine is a change in value of the net. The output signals, *I* and *D*, are sent to the gate state machine. (For historical reasons, these signals are also known as *Increment* and *Decrement*.) The *Proc* element of the data structure points to the subroutine that will process the next change in the net. There is one subroutine for zero-to-one changes and another for one-to-zero changes. This pointer is the only state information that is maintained for the net. The *Next* and *Previous* elements are used for queuing, while the *Gate* element points to the gate that will receive the output signals from the net.

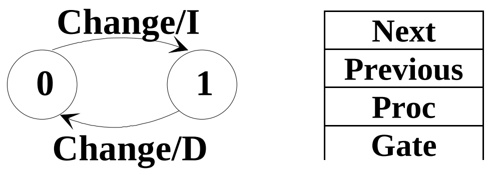

Figure 2. A Net State Machine.

The state machine for a two-input AND gate and its associated data structure are given in Figure 3. This machine receives *I* and *D* signals from two different net state machines. Since the two inputs are symmetric, it is not necessary to distinguish between them. It is necessary, however, to keep track of the number of inputs that are equal to 1. If both inputs are equal to 1, then the output is equal to 1, otherwise the output is equal to 0. The output changes when a transition is made between states 1 and 2. The *Q* output causes the output of the gate to be added to the end of the simulation queue.

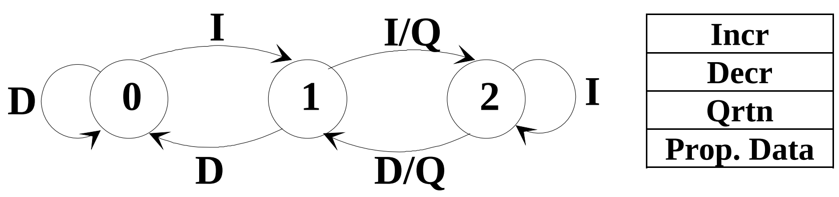

Figure 3. A Gate State Machine.

Figure 3 shows the gate data structure. The *Incr* and *Decr* elements maintain the state of the machine. These elements point to the subroutines that handle the *I* and *D* inputs from the input machines. The input state machines "transmit" their inputs by calling one of these routines directly. The *Qrtn* element maintains the queuing state of the gate. The queuing actions are dependent on the timing model, and are beyond the scope of this article.

The code to support the net and gate state machines is surprisingly simple. That for the net state machine is given below. The two subroutines *DProcessor* and *IProcessor*, toggle back and forth between one another to maintain the state of the net. The only difference between the two is that *IProcessor* calls the *Incr* subroutine and *DProcessor* calls the *Decr* subroutine. We have replaced tail recursions with computed goto statements to improve performance.

```
IProcessor:
   Cev->Proc = &&DProcessor;
   Cgt = Cev->Gate;
   goto * Cgt->Incr;

DProcessor:
   Cev->Proc = &&IProcessor;
   Cgt = Cev->Gate;
   goto * Cgt->Decr;
```

The code for the gate state machine given below is only slightly more complicated than the code for the net state machine. The *Decr0* and *Incr2* routines are never called. The other four routines change state by assigning new subroutine addresses to *Incr* and *Decr*. The two routines *Incr1* and *Decr2* call the queuing subroutine to queue the output net, while the routines *Incr0* and *Decr1* advance to the next queued net. As before, computed goto statements are used in place of subroutine calls.

```
Incr0:
   Cgt->Incr = &&Incr1;
   Cgt->Decr = &&Decr1;
   Cev = Cev->Next;
   goto *Cev->Proc;

Decr0:
   Cev = Cev->Next;
   Goto *Cev->Proc;

Incr1:
   Cgt->Incr = &&Incr2;
   Cgt->Decr = &&Decr2;
   goto *Cgt->Queue;

Decr1:
   Cgt->Incr = &&Incr0;
   Cgt->Decr = &&Decr0;
   Cev = Cev->Next;
   Goto *Cev->Proc;

Incr2:
   Cev = Cev->Next;
   goto *Cev->Proc;

Decr2:
   Cgt->Incr = &&Incr1;
   Cgt->Decr = &&Decr1;
   Goto *Cgt->Queue;
```

The rest of the code for our simulator is similar to that given above. The subroutines contain assignment statements, but no conditional statements and no loops. Computed goto's are used in place of subroutine calls. The code is a straight-line series of assignments with a few labels and computed goto's.

More information can be found in references [3-6].

4. But is this Good Code?

If nothing else, our simulation algorithm is peculiar-looking. What is more, we have managed to violate most of the rules of good coding. We obviously don't consider the goto harmful, in fact it seems to be our most important tool. Not just gotos, but *computed* gotos, the very worst kind! We also don't seem believe that object definitions ought to be static. Indeed, we seem to go to extraordinary lengths to violate this rule, even to the extent of inserting assembly language into high-level programs. This is hardly the sort of thing we would recommend in computer science 101.

We do, in fact, admit that our code leaves something to be desired. *But it's not our fault!* It's the fault of our tools. We *believe* in metamorphic programming, but using function pointers isn't the way to do it. We ought to be able to restrict object metamorphosis to a specified collection of definitions, each one of which is static. With function pointers we could morph anything to practically anything else. This isn't good, but our tools won't let us do anything else. (We could use the *State* pattern from the gang-of-four patterns[7], but that isn't necessarily elegant or efficient either.)

And what about all those gotos? We use them because our tools don't give us any way to specify cheap function calls. When we go from one subroutine to another, we don't need a new stack frame, we don't need a new return address, we don't need a new set of parameters, and we don't need any new local variables. We just need to get from one place to another, and we don't want to pay for a bunch of stuff we don't need. We could use tail recursion and ordinary function calls. Then we could cross our fingers and hope that our optimizer will be able to undo all the damage, but this seems a little chancy. It's like leaving off the *inline* keyword, hoping the compiler will guess right about what needs to be expanded in-line. (See [8] for a concurring opinion.)

No, we don't need better code. We need better tools. But before we discuss better tools, we need to look at a few more metamorphic algorithms. After all, no one is going to create a new set of tools just for one algorithm.

5. Other Metamorphic Algorithms

We have implemented metamorphic solutions to many common computer science problems, and are convinced that metamorphosis could be a powerful tool for many different problems. Any algorithm that uses state-data is a candidate for metamorphic programming. Algorithms, such as string matching and lexical analysis, which are explicitly state based, are readily adaptable to metamorphic techniques. Graph algorithms that maintain state data, such as shortest path and depth first search, are also good candidates. Even straightforward algorithms like sorting are somewhat state-based, since the behavior of the algorithm changes when the end of a list is encountered. We have chosen two examples to illustrate metamorphic programming. These are insertion sort, Quicksort. A number of other examples can be found on our website[9].

5.1. Insertion Sort

Our algorithm is based on the iterative algorithm given below. The objects to be sorted are stored in a doubly linked list and sorting is done by calling the same function for each object in the list. Each object has two functions, a forward routine and a backward routine. The forward routine replaces the outer loop of the iterative algorithm, while the backward routine replaces the inner loop.

```
for (long i=1 ; i<n ; i++)
{
   long x = L[i];
   for (long j=i-1 ; j>=0 && L[j]>x ;
j--)
   {
      L[j+1] = L[j];
   }
   L[j+1] = x;
}
```

The code for the forward and backward routines is given in below. Each object to be sorted points to the *Forward* and *Backward* functions. During the forward traversal, each object is removed from the list and reinserted into its proper position in the sorted portion of the list. A backward traversal is used to locate the proper position for the removed element. Once the removed element is reinserted, the forward traversal resumes. Terminator objects pointing to the EOL and SOL routines are used at the ends of the list to terminate traversals.

```
Forward:
   This = Current;
   Current = Current->Next;
   BackPtr = This->Prev;
   // unlink;
   This->Next->Prev = This->Prev;
   This->Prev->Next = This->Next;
   goto * BackPtr->BackwardRtn;
```

```
Backward:
  if (This->Value < BackPtr->Value)
  {
     BackPtr = BackPtr->Prev;
     goto * BackPtr->BackwardRtn;
  }
  else
  {
     This->Next = BackPtr->Next;
     This->Prev = BackPtr;
     BackPtr->Next->Prev = This;
     BackPtr->Next = This;
     goto * Current->ForwardRtn;
  }
```

```
EOL:
   return
```

```
SOL:
  This->Next = BackPtr->Next;
  This->Prev = BackPtr;
  BackPtr->Next->Prev = This;
  BackPtr->Next = This;
  goto * Current->ForwardRtn;
```

This example illustrates one of the most important benefits of metamorphic programming: the elimination of “Are we there yet?” programming. The iterative insertion sort algorithm is like a child on a long trip who continually asks “Are we there yet?” The outer loop executes the same test “i<n” over and over, searching for the end of the list. In object oriented programming, objects should “know” when they are at the end of the list. Repetitive testing shouldn’t be required. Admittedly we’ve cheated a bit by using terminator objects, but the algorithm can easily be rewritten to eliminate them. We invite the interested reader to give it a try.

5.2. Quicksort

Metamorphic programming does not require linked lists; arrays will work just as well. In our Quicksort algorithm the objects to be sorted are stored in an array. When a list is split, two new sub-lists are created. The algorithm continues iteratively with one list and pushes the other onto a stack. If the current list contains fewer than two elements, the stack is popped. The popping continues until a list with two or more elements is found or until the stack becomes empty. When the stack becomes empty the algorithm terminates.

A list is split by calling the *Process* function of each object. Each object has two data items, a value and a pointer to a processing routine. The last element in the list points to the *LastTest* subroutine. All other objects point to the *Test* subroutine. Because lists are divided into smaller and smaller sub-lists, it is necessary to morph the last object in each list into a list terminator.

Stack processing is also done metamorphically. Each stack element is an object that contains the list boundaries and a pointer to a processing routine. The last stack element is a terminator whose processing routine terminates the sort algorithm. The code for the *Test* and *LastTest* routines is given below.

Information and Conditional Branches

Array and list processing loops like "for (i=0 ; i<n ; i++)" are not just annoying, they are bad mathematically. A conditional test provides information to a program. If p is the probability that "i<n" is true, then the information provided by the test is $I_t = p\lg\frac{1}{p} + (1-p)\lg\frac{1}{(1-p)}$. This formula achieves its maximum value, 1, when p=.5. (That is, when true and false are equally likely.) For an array of ten elements there will be nine true results and one false result making *p*=.9. In this case, $I_t = .9\cdot\lg\frac{10}{9} + .1\cdot\lg 10 = 0.469$. If there are one thousand elements, then $I_t = .999\cdot\lg\frac{1000}{999} + .001\cdot\lg 1000 = 0.0114$. The point is that the program is doing much work to obtain little information. Compare the array termination test with the key-comparison test in Quicksort, "*This->Value < Pivot->Value*". The probability of this condition being true is .5 (a fact that we have verified experimentally), thus the test yields the maximum amount of information. (The information provided by the insertion sort key test, "*This->Value < BackPtr->Value*," goes to zero as *n* goes to infinity, suggesting that the algorithm is not using its comparisons effectively.)

```
Test:
  if (This->Value < Pivot->Value)
  {
     Split++;
     Swap(This->Value,
       Split->Value);
  }
  This++;
  goto * This->Process;
```

```
LastTest:
  if (This->Value < Pivot->Value)
  {
     Split++;
     Swap(This->value,
        Split->Value);
  }
  Swap(Pivot->Value,
     Split->Value);
  // Demorph last element
  This->Process = &&Test;
  // push first sublist
  List->First = First;
  List->Last = Split-1;
  List->Process = &&NewList;
  List++;
  // iterate through 2nd sublist
  First = Split+1;
  goto NewList;
```

The *NewList* routine, which sets up a new list and pops the stack, is given below.

```
NewList:
   if (Last <= First)
   { // Pop List
      List--;
      First = List->First;
      Last = List->Last;
      goto * List->Process;
   }
   // Set up list and process
   Split = First;
   Pivot = First;
   This = First+1;
   Last->Process = &&LastTest;
   goto * This->Process;
```

6. Better Tools

If metamorphic programming is ever to become a serious alternative to iterative programming, we need to eliminate warts from our metamorphic programs. We need *real* metamorphic objects, not just objects that are programmed that way. We need objects that are declared to be metamorphic, with clearly specified rules that are checked by the compiler. This is not really a new idea because almost anything you can think of has been implemented in some language somewhere, and metamorphosis is no exception. But in our research we cannot afford to fool around with arcane or experimental languages. We are attempting to build simulation tools that will be used to verify the next generation of VLSI circuits, and we need to concentrate on mainstream languages like Java, C++, and Visual Basic. The same is true for anyone who wants to use metamorphic programming in a mainstream application. Fortunately, the changes required to support metamorphosis are relatively minor.

To show the simplicity of metamorphic language features, it is necessary to look at the low-level implementation of an object. Suppose we have three classes, A, B, and C. A is the base type from which B and C are derived. Classes B and C do not define any new data items. Class A has a number of virtual functions that are overridden in classes B and C. Other than the overrides, classes B and C define no new functions. The compiler creates an object called a vtable for each class. The vtable contains a pointer to each virtual function defined by the class, or inherited by the class. Each object of the class contains a pointer to the vtable.

Because of the restrictions we have placed on classes B and C, All four objects have the same data items, and all three vtables have the same layout. It is possible to morph objects between types A, B, and C by replacing the vtable pointer. Because this replaces all virtual functions, we call this *complete metamorphosis*. Complete metamorphosis is quite trivial to implement. Syntactically, we could use a statement such as the following. (We are using verbose statements for clarity. We expect that compiler designers would choose something more elegant.) This statement could be implemented with one or two assembly language instructions, and could be easily verified. Few, if any changes would be needed in class definition syntax.

Morph Object1 [from A] to B;

We could go one step further with complete metamorphosis and permit the classes A, B, and C to define different sets of data items. In this case, it would be necessary to formally declare A, B, and C as mutually morphable classes, since any object of type A would need to contain all data items declared by B and C, even though these items would not be accessible to A's functions.

Complete metamorphosis is easy, but it's not always feasible. This is especially true when we combine several state machines together into a single object. Suppose we have five state machines embedded in a single object, with five states for each machine. To model the object using complete metamorphosis, it would be necessary to define 3125 different classes to capture all state combinations. To simplify the construction of such objects, we propose a more dynamic technique called *partial metamorphosis*. Partial metamorphosis allows individual members of the vtable to be replaced, but requires each object to have its own personal copy of the vtable. (The affected portion of the vtable could be integrated into the object itself, eliminating the double indirection.)

Partial metamorphosis can be faked using function pointers, but there are problems with this approach. The only tool that the compiler has for determining the correctness of an assignment to a function pointer is the function type. This can lead to difficult-to-diagnose program errors if an incorrect value is assigned to a pointer variable. We need an alternative technique that restricts the list of functions that can participate in a metamorphosis operation.

In the code below, we define a function ABC, which has no body of its own, but will act as a dynamic reference to either A, B, or C. It is necessary for the function headers of ABC, A, B, and C to be identical. Even though the function header of D is identical to that of ABC, ABC is not permitted to refer to D. We can enhance the morph statement for partial metamorphosis, as shown below. In this case the operands of the statement are functions instead of objects and classes.

morph ExampleObject.ABC to ExampleObject.A;

```
class CExample
{
public:
   void ABC(void) one of A, B, C;
private:
   void A (void)
   {
      …
   }
   void B (void)
   {
      …
   }
   void C (void)
   {
      …
   }
   void D (void)
   {
      …
   }
};
```

7. Metamorphic Functions

Even though we've committed the worst of all sins by using computed goto's, we insist that a slight change in compiler technology could transform this into clean well-structured code. We were forced into using goto's because we need cheap function calls that share the same parameters, local variables, and return address. Even though each routine is a segment of a larger function, we usually don't think of them this way. It is easier to program if we consider the code segments to be functions, and the goto's to be function calls. Because the stack frame is shared between these "functions," it is convenient to think of the "function call" as replacing the body of the current function. Thus, we tend to think of a code segment as a *metamorphic function* that can be dynamically transformed into some other function.

The concept is similar to the concept of multi-threading. Threads are cheap processes that share the same address space and other resources, metamorphic functions are cheap function calls that share stack frames. We believe that metamorphic functions should be organized into mutually morphable groups. When a function calls another function from its own group, no new stack frame will be created. When it calls a function in a different group, a new stack frame and return address will be created. The declaration should be similar to the C++ *inline* declaration, except that a group name will be used to identify the group to which the function belongs. This syntax is illustrated in below, using the keyword *segment*. We also use the keyword *primary* to identify functions that can be called from outside their group. Although most of our applications clearly distinguish between the primary entry point of a function and its internal segments, we are not convinced that this would be a useful distinction in a more general context. A single keyword may suffice for all metamorphic functions.

```
class MyObj
{
public:
   MyObj *Next;
   primary Exam1 void ProcessList(void)
   {
      ProcessObject();
   }
   segment Exam1 virtual void ProcessObject(void) = 0;
};

class OtherObj1 : public MyObj
{
public:
   segment Exam1 virtual void ProcessObject(void)
   {
      // process object here
      if (Next != NULL)
      {
         Next->ProcessObject();
      }
   }
};

MyObj * Head;
```

In the preceding code, we assume that there are several different classes derived from *MyObj*, and that each of them overrides the function *ProcessObject*. The variable *Head* is assumed to be the head of a list of objects. To process this list, we use the single function call given below. No loop is required. It is possible to avoid testing the *Next* variable for *NULL* by using a trailer object whose sole function is to terminate a linked list of objects, or by using a different function for the last object in the list.

```
Head->ProcessList();
```

8. Conclusion

We have found metamorphic programming to be an effective tool in our search for more efficient algorithms, particularly in the area of logic simulation. We have created metamorphic solutions for many different problems, far too many to describe here. We have found that virtually all of these implementations have given us some increase in performance, although seldom to the degree that we have observed in logic simulation. On the other hand, we are just scratching the surface of metamorphic programming. There are many variant solutions to the problems we have discussed here, some of which may be significantly more efficient than the solutions we have found. Much more work is needed to discover the most efficient and effective metamorphic techniques for various different problems. The most important problem is the lack of metamorphic constructs in mainstream high-level languages. It is our hope that such features will be provided in the future, and that metamorphic programming will become an important tool in the future.

9. Acknowledgement

The author would like to thank Professor Greg Speegle for his many helpful comments during the preparation of this paper.

10. References

1. Cardelli, L, and Wegner, P. “On Understanding Types, Data Abstraction, and Polymorphism,” ACM Computing Surveys, Vol. 17 No. 4, Dec 1985, pp. 471-522.
2. Abadi, M. and Cardelli, L., "A Theory of Objects," Springer, Heidelberg, 1996.
3. Maurer, P, “The Shadow Algorithm: A Scheduling Technique for both Compiled and Interpreted Simulation ,” IEEE Transactions on Computer Aided Design, vol 12, No. 9, Sept. 1993, pp.1411-1413.
4. Maurer, P, “The Inversion Algorithm for Digital Simulation” IEEE Transactions on Computer Aided Design, July 1997, pp. 762-769.
5. Maurer, P, “Event Driven Simulation Without Loops or Conditionals,” ICCAD 2000, Nov. 2000, pp. 23-26.
6. Lewis, D., “A Hierarchical Compiled Code Event-Driven Logic Simulator,” IEEE Transactions on Computer Aided Design, Vol. 10, No. 6, June 1991, pp.726-737.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., “Design Patterns: Elements of Reusable Object-Oriented Software” Addison Wesley, New York, 1995.
8. Steele, G. L. Jr. “Debunking the ‘expensive procedure call’ myth, or procedure call implementations considered harmful, or lambda, the ultimate GOTO,” ACM Conference Proceedings, pp. 153-162, 1977.
9. Brglez, Pownall, Hum, “Accelerated ATPG and Fault Grading via Testability Analysis,” ISCAS-85, pp. 695-698.
10. Maurer, P. “The Metamorphic Programming Website: Examples,” http://cs.ecs.baylor.edu/~maurer/Metamorphic.