Morphing Software for Easier Evolution

Shan Shan Huang[1,2], and Yannis Smaragdakis[2]

[1] Georgia Institute of Technology, College of Computing, ssh@cc.gatech.edu
[2] University of Oregon, Department of Computer and Information Sciences
yannis@cs.uoregon.edu

1 Introduction

One of the biggest challenges in software evolution is maintaining the relationships between existing program structures. Changing a program component (e.g., a class, interface, or method) typically requires changes in multiple other components whose structure or meaning depend on the changed one. The root cause of the problem is redundancy due to lack of expressiveness in programming languages: Extra dependencies exist only because there is no easy way to model one program component after another, so that changes to the latter are automatically reflected in the former. For example, in the Enterprise Java Bean (EJB) standard, local and remote stub interfaces must mirror the bean class structure exactly. A change in the bean interface must be propagated to the stub interfaces, as well. Tools and methods have been developed to support writing code that is immune to changes in program structure (e.g., [10, 11]). But these tools either require separate declarations of a program's structural properties (e.g., class dictionaries in [11]), or use potentially unsafe runtime reflection [10]. Furthermore, these tools focus on adapting code, and not the static structure of a class or interface, to evolving program structure.[1]

Another obstacle in software evolution is the extensibility of software components, particularly when source code is unavailable. Aspect Oriented Programming (AOP) [9] and its flagship tools, such as AspectJ [8] provide a solution approach. AspectJ allows a programmer to extend a software component by specifying extra code to be executed, or even change the component's original semantics entirely by circumventing the execution of original code, and provide new code to execute in its place. AspectJ is a powerful tool, but often has to sacrifice either discipline or expressiveness. For example, AspectJ aspects are strongly tied to the components they apply to—there is no notion of typechecking an aspect separately from the application where it is used. This means that *generic* AspectJ aspects (i.e., aspects that are specified so that they can be later applied to multiple, but yet unknown, components) are limited in what they can express. For example, AspectJ cannot express intercepting all calls to the methods of one class, and forwarding them to methods of another class, using the intercepted arguments: the aspect to do so needs to be custom-written

[1] Tools have been developed to specifically target generating EJB stubs [5] so that consistency between the bean class and its stubs is managed automatically. But this is a solution to one specific problem, and not generally applicable.

for the specific classes, methods, and arguments it will affect. AspectJ also does not provide explicit means of controlling aspect application. For example, the order of aspect composition may affect behavior in ways unanticipated by the developers.

With these two obstacles in mind, we recently introduced a language feature that we call "morphing" [7]. Morphing supports a powerful technique for software evolution, and it overcomes many of the shortcomings of existing solutions. We discuss morphing through MJ—a reference language that demonstrates what we consider the desired expressiveness and safety features of an advanced morphing language. MJ morphing can express highly general object-oriented components (i.e., generic classes) whose exact members are not known until the component is parameterized with concrete types. For a simple example, consider the following MJ class, implementing a standard "logging" extension:

```
class MethodLogger<class X> extends X {
  <Y*>[meth]for(public int meth (Y) : X.methods)
  int meth (Y a) {
    int i = super.meth(a);
    System.out.println("Returned: " + i);
    return i;
  }
}
```

MJ allows class `MethodLogger` to be declared as a subclass of its type parameter, `X`. The body of `MethodLogger` is defined by static iteration (using the `for` statement—the central morphing keyword) over all methods of `X` that match the pattern `public int meth(Y)`. `Y` and `meth` are pattern variables, matching any type and method name, respectively. Additionally, the `*` symbol following the declaration of `Y` indicates that `Y` matches any number of types (including zero). That is, the pattern matches all `public` methods that return `int`. The pattern variables are used in the declaration of `MethodLogger`'s methods: for each method of the type parameter `X`, `MethodLogger` declares a method with the same name and signature. (This does not have to be the case, as shown later.) Thus, the exact methods of class `MethodLogger` are not determined until it is type-instantiated. For instance, `MethodLogger<java.io.File>` has methods `compareTo` and `hashCode`: the only `int`-returning methods of `java.io.File` and its superclasses.

MJ morphing supports disciplined software evolution in the following ways:

- MJ allows a class's members to mirror those in another class, e.g., one of its type parameters. The structure of an MJ generic class adapts automatically to the evolving interfaces of its type parameters. (MJ's `for` construct can also be used in declaring statements. Thus, MJ can be used to adapt code to changing program structures, as well.)
- MJ generic classes support modular type checking—a generic class is type-checked independently of its type-instantiations, and errors are detected if they can occur with *any* possible type parameter. This is an invaluable property for generic code: it prevents errors that only appear for some type parameters, which the author of the generic class may not have predicted.

– MJ allows programmers to use both a "transformed" version of a class and the original class at will. For example, a programmer may refer to both the original `java.io.File` and its logged version `MethodLogger<java.io.File>` within the same piece of code.
– Order of composition is explicit in MJ: given two MJ classes, adding two pieces of functionality, e.g., logging through `MethodLogger` and synchronization through `MethodSynchronizer`, applying logger before synchronizer is simply, `MethodSynchronizer<MethodLogger<java.io.File>>`.

In addition to the above properties, MJ differs from existing "reflective" program pattern matching and transformation tools [2–4, 12] by making reflective transformation functionality a natural extension of Java generics. For instance, our above example class `MethodLogger` appears to the programmer as a regular class, rather than as a separate kind of entity, such as a "transformation". Using a generic class is a matter of simple type-instantiation, which produces a regular Java class, such as `MethodLogger<java.io.File>`.

We next elaborate on how MJ morphing supports adaptation to evolving program structures and its modular type safety properties through examples.

2 Adapting Structure to Changing Structures

The structure of an MJ generic class can evolve consistently with the structure of its type parameter. This property allows writing feature extensions and adaptations that are inherently evolvable. We illustrate the benefits of this property using a common design pattern: wrapper [6]. A wrapper class declares the exact same methods as the class it wraps. It delegates each method call to the wrapped class, adding functionality before or after the delegation. The `MethodLogger` class of the previous section is a classic wrapper.

For a real world exposition of the wrapper pattern, consider the class `java.util.Collections`, a utility class provided by the Java Collections Framework (JCF) [1]—the standard Java data structures library. `java.util.Collections` provides a number of methods that take a particular kind of data structure, and return that data structure enhanced with some additional functionality. For example, the method `synchronizedCollection(Collection<E> c)` takes a `Collection c`, and returns a synchronized version of that collection. The implementation of this method returns an instance of the wrapper class `SynchronizedCollection<E>`. For each method in the `Collection` interface, `SynchronizedCollection` defines a method with the exact same signature. The bodies for these methods all first synchronize on a mutex, and then delegate the call to the underlying `Collection` object. `java.util.Collections` offers similar methods that return synchronized versions of other kinds of data structures: `synchronizedSet(Set<E>)`, `synchronizedSortedSet(SortedSet<E>)`, `synchronizedList(List<E>)`, etc. Each of these methods, in turn, requires its own wrapper class definition: `SynchronizedSet<E>`, `SynchronizedSortedSet<E>`, `SynchronizedList<E>`, etc.

Note that these wrapper class definitions are tightly coupled with the interface of the data structures they are wrapping. If the interface of `Collection<E>` changes (e.g., a new method is added, or an existing method is now taking an extra argument), the class `SynchronizedCollection<E>` needs to be redefined, as well. Note also the redundancy at both the class and the method level. At the class level, all `Synchronized*` wrapper classes have the same structure, yet one wrapper class needs to be defined for each data structure to be synchronized. If the need to synchronize a new data structure arises, then a new wrapper class needs to be defined. At the method level, all methods within a `Synchronized*` class share a highly regular structure: first synchronize on a mutex, then delegate the call. Yet they still need to be defined individually. Java provides no way to modularly impose this structure on all methods.

With MJ, however, we can remove such dependency and redundancy, with a single MJ class:[2]

```
public class SynchronizeMe<interface X> implements X {
  X x;
  Object mutex;
  public SynchronizeMe(X x) { this.x = x; mutex = this; }

  //For each non-void method in X, declare the following:
  <R,A*>[m] for(public R m(A) : X.methods)
  public R m (A a) { synchronized(mutex) { return x.m(a); } }

  // Similarly for each void-returning method in X
  <A*>[m] for(public void m(A) : X.methods)
  public void m(A a) { sychronized(mutex) { x.m(a); } }
}
```

`SynchronizeMe` decouples the synchronization feature from the interfaces needing such a feature. Its definition adapts effortlessly to the interfaces it wraps. There is no need to modify `SynchronizeMe` when the underlying wrapped interface changes. `SynchronizeMe` can be instantiated with any interface to provide a synchronized implementation of that interface, thus replacing all `Synchronized*` wrapper classes. Additionally, `SynchronizedMe` removes method-level redundancy using a static iteration block to impose the same structure on all methods.

The full version of the above MJ class consists of less than 50 lines of code, replacing more than 600 lines of code in the JCF. Similar simplifications can be obtained for other nested classes in `java.util.Collections`, which account in total for some 2000 lines of code in the original JCF implementation: `UnmodifiableSet`, `UnmodifiableList`, `UnmodifiableMap`, etc. are replaced by a single morphed class, and the same is done for `CheckedSet`, `CheckedList`, `CheckedMap`, etc.

While adding logging or synchronization functionality is doable with AOP tools such as AspectJ, MJ allows the "morphing" of a wrapper class in much

[2] This example implementation uses `this` as the mutex. A more flexible implementation could provide a constructor that allows programmers to choose their own mutex. In fact, this is the strategy adopted in the JCF.

more interesting ways. For example, one can declare a MJ class `MakeLists<C>` such that, for each single-argument method of its type parameter `C`, `MakeLists<C>` has a method of the same name, but takes a *list* of the original argument type, invokes the original method on each element of the list, and returns a list of the original method's return values:

```
class MakeLists<C> {
  C c; // wrapped object.
  ... // constructor initializing c.

  <R,A>[m] for(R m (A) : C.methods)
  List<R> m (List<A> la) {
    ArrayList<R> rlist = new ArrayList<R>();
    if ( la != null )
      for ( A a : la ) { rlist.add(c.m(a)); }
    return rlist;
  }
}
```

This transformation is not expressible using AspectJ. Consider how this functionality can be implemented using plain Java: a `MakeListsSomeType` class would have to be declared for every `SomeType` that we want to have this extension for. Additionally, if the structure of `SomeType` changes, e.g., a new single-argument method is added, or an existing method changes its argument or return types, `MakeListsSomeType` would need to be modified to reflect those changes, as well. In contrast, the MJ generic class `MakeLists<C>` works for any class `C` without advanced planning of which types this extension can be added to. It also morphs with the structure of each `C`, without further programmer intervention.

3 Modular Type Checking

For an example of modular type checking, consider the following "buggy" class:

```
class CallWithMax<class X> extends X {
  <Y>[meth]for(public int meth (Y) : X.methods)
  int meth(Y a1, Y a2) {
    if (a1.compareTo(a2) > 0) return super.meth(a1);
    else return super.meth(a2);
  }
}
```

The intent is that class `CallWithMax<C>`, for some `C`, imitates the interface of `C` for all single-argument methods that return `int`, yet adds an extra formal parameter to each method. The corresponding method of `C` is then called with the greater of the two arguments passed to `CallWithMax<C>`. It is easy to define, use, and deploy such a generic transformation without realizing that it is not always valid: not all types `Y` will support the `compareTo` method. MJ detects such errors when compiling the above code, independently of instantiation. In

this case, the fix is to strengthen the pattern with the constraint `<Y extends Comparable<Y>>`:

```
<Y extends Comparable<Y>>[meth]for(public int meth (Y) : X.methods)
```

Additionally, the above code has an even more insidious error. The generated methods in `CallWithMax<C>` are not guaranteed to correctly override the methods in its superclass, `C`. For instance, if `C` contains two methods, `int foo(int)` and `String foo(int,int)`, then the latter will be improperly overridden by the generated method `int foo(int,int)` in `CallWithMax<C>` (which has the same argument types but an incompatible return type). MJ statically catches this error.

4 A Comparison to AOP

Morphing can be used to address some of the same issues as AOP. To be sure, morphing only relates to a small but central part of AOP functionality: aspect advice of structural program features, such as method before-, after-, and around-advice. Particularly, the logging and synchronization examples shown in previous sections are frequent use cases for AOP languages. Thus, it is worth delineating the similarities and distinct differences between morphing and AOP. We next compare MJ, the only reference morphing language, to AspectJ [8], a representative AOP tool for Java.

4.1 How Functionality is Added

Both MJ and AspectJ allow functionalities that cross-cut multiple class definitions to be defined in a modular way. For example, the method logging functionality can be defined in one MJ class, `MethodLogger`. However, the way such functionalities are added into a base class definition is one of the main differences between MJ and AspectJ. With MJ, cross-cutting functionality is added "into" a base class through explicit parameterization of the morphing class. The new functionality only exists in the parameterized morphing class, whereas the definition of the base class itself does not change. For example, parameterized morphing class `MethodLogger<java.io.File>` has the functionality that all `int`-returning methods are logged. However, the definition of `java.io.File` itself remains unchanged. In AspectJ, an aspect definition states the classes a functionality should be added to.[3] In this way, the new functionality is weaved with the code of the original class. The program cannot simultaneously use the separate notions of "original class" and "class with the cross-cutting functionality". One way to view the semantics of AspectJ is as changing the original class's definition. For instance, given an AspectJ aspect that adds logging code to each `int`-returning method of `java.io.File`, the class `java.io.File` itself can be thought of as changed after aspect application. Indeed this also happens to be the way current AspectJ compilers implement the semantics of weaving.

[3] In the case of generic aspects in AspectJ 5, the affected classes can be specified through parameterization of the aspect.

We view explicit parameterization in MJ as an important feature for two reasons. First, the ability to leave the original class definition untouched is an important one. For example, a programmer should be able to use both synchronized and unsynchronized versions of a data structure in the same program, depending on his needs. This is indeed the case with the MJ class `SynchronizeMe`, shown in Section 2. With AspectJ, however, the programmer must choose one or the other. AOP purists may hold the view that cross-cutting functionality enhancements, by definition, should be applied to all classes that need them. But as shown through the synchronization example, this is a very rigid requirement. Furthermore, if indeed all instantiations of a class should have a particular cross-cutting functionality, it should be possible to extend MJ with a global search-and-replace tool, replacing all instances of a class with the explicitly parameterized version of a morphing class. This is part of future work, however. Our current research focuses on the fundamental core of morphing, and we expect that usability enhancements will come later.

Secondly, explicit parameterization provides a way to clearly document and control the semantics of a program. This is particularly true when multiple, separately-defined functionality enhancements need to be added to a class. One of the much researched topics in AOP is aspect interaction. When one defines an aspect in AspectJ, there is no good way to specify the order of its application relative to all other aspects, some of which may be unknown to the aspect developer. Furthermore, an addition of another aspect unknown to the developer can change the program semantics in unexpected ways. This is an undesirable characteristic in terms of modularity. MJ, on the other hand, allows explicit control of functionality addition through instantiation order. The type-instantiation order gives a clear meaning as to where and how functionality is added.

4.2 Modular Type Safety and Trade-offs in Expressiveness

The other main difference between MJ and AspectJ is MJ's guarantee of modular type safety. In order to make such guarantees, we limited our attention to some specific features instead of adding maximum expressiveness to the language. Pattern matching in MJ is simple and high level by design. A programmer can only inspect classes at the level of method and field signatures: MJ pattern matching applies to reflection-level structural elements of a type. In contrast, AspectJ allows a programmer to match on a program's dynamic execution characteristics, using keywords such as `cflow` (for control flow) and `cflowbelow` in pointcuts.

Though MJ limits its pattern matching to the type signature level, it does allow matching using subtype-based semantic conditions, in contrast to the purely syntactic matching of signatures AspectJ offers. For instance, using pattern-matching type variables, MJ allows one to express a pattern that matches all methods that return *some* subtype of `java.lang.Comparable`. This is a pattern not expressible through AspectJ. The combination of pattern matching and the static `for` construct in MJ provides a controlled but useful kind of programmability in defining where a certain functionality should be introduced.

Although we make comparisions only to AspectJ, the arguments in this section generalize to other AOP tools, as well. AspectJ is representative in the way it applies aspects, and is perhaps the most expressive aspect language today.

5 Future Work: When Plain Morphing Isn't Enough

We have demonstrated through examples that morphing with MJ is particularly useful for defining classes whose structure must mirror the structure of another class or interface. However, there are some useful cases that MJ, with just pattern matching and static iteration, cannot express in a modularly type safe way. For example, it is often the case that each field in a class has its own getter and setter methods. These getter and setter methods must be manually defined by the developer. This seems to be the perfect use-case for morphing. We ought to be able to define a morphing class that defines a getter and a setter method for each field in its type parameter:

```
public class AddGetterSetter<class C> extends C {
  <F>[f] for( F f : C.fields )
  public F get#f () { return f; }

  <F>[f] for( F f : C.fields )
  public void set#f ( F newF ) { f = newF; }
}
```

Note that `AddGetterSetter` uses the MJ language construct `#`, which concatenates a constant prefix to a pattern matching name variable. For each field `SomeField` in `C`, `AddGetterSetter<C>` declares a getter method, `getSomeField`, and a setter method, `setSomeField`. However, this class is not modularly type safe. We cannot establish that the names of the methods being declared, `getSomeField` and `setSomeField`, whatever `SomeField` may be, do not conflict with methods in the superclass `C`. For example, `AddGetterSetter<Foo>` would not be a well-typed class if `Foo` is defined as follows:

```
public class Foo {
  int up;

  public int setup ( int i ) { ... }
}
```

`AddGetterSetter<Foo>` contains the method `void setup(int)`, which incorrectly overrides the method `int setup(int)` in its superclass `Foo`: `setup` in `AddGetterSetter<Foo>` has the same argument type as `setup` in `Foo`, but a non-covariant return type.

One possible way to express such functionality while keeping modular type safety is to put an additional condition on each element in the static iteration. We need to be able to express that we only want to iterate over those fields `f` of `C` for which a `set#f` (or `get#f`) method does not already exist in `C` itself. We

are currently considering an extension to the pattern language. For example, we could change the static iteration block defining the setter method to:

```
<F>[f] for( F f : C.fields;
            no set#f(F) : C.methods )
public void set#f ( F newF ) { f = newF; }
```

Note the extra clause: `no set#f(F) : C.methods`. This clause serves as the extra conditional needed to ensure that no conflicting method already exists in `C`.

This addition to MJ's pattern matching language might seem simple and innocuous at first, but the combination of iteration along with conditionals on types and their structures can easily yield undecidable type systems. We must take care to restrict the conditionals so that our type checker can still provide useful feedback to the programmers. Striking this balance between the need for this additional expressiveness and the tractability of the type system is our future focus.

6 Conclusion

Overall, we consider MJ and the idea of morphing to be a significant step forward in supporting software evolution. Morphing can be viewed as an aspect-oriented technique, allowing the extension and adaptation of existing components, and enabling a single enhancement to affect multiple code sites (e.g., all methods of a class, regardless of name). Yet morphing can perhaps be seen as a bridge between AOP and generic programming. Morphing allows expressing classes whose structure evolve consistently with the structures they mirror. Morphing strives for smooth integration in the programming language, all the way down to modular type checking. Thus, reasoning about morphed classes is possible, unlike reasoning about and type checking of generic aspects, which can typically only be done after their application to a specific code base. Morphing does not introduce functionality to unsuspecting code. Instead, it ensures that any extension is under the full control of the programmer. The result of morphing is a new class or interface, which the programmer is free to integrate in the application at will. We thus view morphing as an exciting new direction in supporting software evolution.

References

1. *Java Collections Framework Web site, http://java.sun.com/j2se/1.4.2/docs/guide/collections/.* Accessed Apr. 2007.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 31–42, Tampa Bay, FL, USA, 2001. ACM Press.
3. J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 270–281, Berlin, Germany, 2002. ACM Press.

4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proc. Fifth Intl. Conf. on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
5. B. Burke et al. *JBoss AOP Web site, http://labs.jboss.com/portal/jbossaop*. Accessed Apr. 2007.
6. E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
7. S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In E. Ernst, editor, *To Appear: Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, July 2007.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
9. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. of the 11th European Conf. on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
10. J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Intl. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.
11. T. Skotiniotis, J. Palm, and K. J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *European Conference on Object-Oriented Programming*, pages 477–500, Nantes, France, 2006. Springer Verlag Lecture Notes.
12. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, pages 216–238. Springer-Verlag, 2004. LNCS 3016.