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 a 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., [9, 10]). But these tools either require separate declarations of a program's structural properties (e.g., class dictionaries in [10]), or use potentially unsafe runtime reflection [9]. Furthermore, these tools focus on adapting code, but 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) [8] and its flagship tools, such as AspectJ [7] 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 [4] 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" [6]. 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 [1–3, 11] 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 inheritently evolvable. We illustrate the benefits of this property using a common design pattern: wrapper [5]. 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. While adding logging functionality is doable with AspectJ, MJ allows the "morphing" of a wrapper class in much 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 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. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. of the 16th ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 31–42, Tampa Bay, FL, USA, 2001. ACM Press.
2. 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.
3. 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.
4. B. Burke et al. *JBoss AOP Web site, http://labs.jboss.com/portal/jbossaop*. Accessed Apr. 2007.
5. E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
6. 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.
7. 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.
8. 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.
9. 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.
10. T. Skotiniotis, J. Palm, and K. J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *European Conf. on Object-Oriented Programming*, pages 477–500, Nantes, France, 2006. Springer Verlag Lecture Notes.
11. 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.