Domain-Specific Languages and Program Generation with Meta-AspectJ

Shan Shan Huang, David Zook
Georgia Institute of Technology
and
Yannis Smaragdakis
University of Oregon

Meta-AspectJ (MAJ) is a language for generating AspectJ programs using code templates. MAJ itself is an extension of Java, so users can interleave arbitrary Java code with AspectJ code templates. MAJ is a structured meta-programming tool: a well-typed generator implies a syntactically correct generated program. MAJ promotes a methodology that combines aspect-oriented and generative programming. A valuable application is in implementing small domain-specific language extensions as generators using unobtrusive annotations for syntax extension and AspectJ as a back-end. The advantages of this approach are twofold. First, the generator integrates into an existing software application much as a regular API or library, instead of as a language extension. Second, a mature language implementation is easy to achieve with little effort since AspectJ takes care of the low-level issues of interfacing with the base Java language.

In addition to its practical value, MAJ offers valuable insights to meta-programming tool designers. It is a mature meta-programming tool for AspectJ (and, by extension, Java): a lot of emphasis has been placed on context-sensitive parsing and error-reporting. As a result, MAJ minimizes the number of meta-programming (quote/unquote) operators and uses type inference to reduce the need to remember type names for syntactic entities.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*program synthesis, program transformation, program verification*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Design,Languages

Additional Key Words and Phrases: meta-programming, domain-specific languages, language extensions

1. INTRODUCTION

Meta-programming is the act of writing programs that generate other programs. Powerful meta-programming is essential for approaches to automating software development. Most domain-specific languages and many other software automation

Authors' address: College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA; email:{ssh,dzook}@cc.gatech.edu; 1202 University of Oregon, Eugene, OR 97403-1202, USA; email: yannis@cs.uoregon.edu; This is an extended version of [Zook et al. 2004] also containing material from [Huang and Smaragdakis 2006].

tasks are implemented as program generators. In this article we present Meta-AspectJ (MAJ): a meta-programming language extending Java with support for generating AspectJ [Kiczales et al. 2001] programs. MAJ offers a convenient syntax, while explicitly representing the syntactic structure of the generated program during the generation process. This allows MAJ to guarantee that a well-typed generator will result in a syntactically correct generated program. This is the hallmark property of *structured* meta-programming tools, as opposed to lexical or text-based tools. Structured meta-programming is desirable because it means that a generator can be released with some confidence that it will create reasonable programs, as long as the inputs satisfy predefined criteria.

MAJ is the reification (in Java/AspectJ) of a general, language-independent approach that we are advocating. We believe that combining generative techniques with aspect-oriented programming results in significant advantages over using either approach alone. MAJ is an example of a meta-aspect language—a meta-programming language capable of generating aspect-oriented programs. Even though MAJ is a modest combination of program generation with aspectization, it is useful for two general tasks. First, it can be used to implement program generators using the aspect language as the program-transformation back-end. Specifically, domain-specific constructs can be implemented in a meta-aspect language, by translating domain-specific abstractions into aspect programs. Second, a meta-aspect language can be used to enhance general-purpose aspect languages using generation. Specifically, limitations of the aspect language itself (e.g., recognizing previously unsupported kinds of joinpoints) can be addressed by writing programs to generate custom aspects in the base aspect language. Thus, a meta-aspect language enables the use of the base aspect language as an aspect-oriented "assembly language" [Shonle et al. 2003] to simplify what would otherwise be tedious tasks of recognizing patterns in an existing program and rewriting them.

Although MAJ is a general tool that can be used for all kinds of program generation tasks, we use it to illustrate some promising directions of the combined generational/aspect approach. MAJ is a perfect match for implementing domain-specific language extensions using unobtrusive annotations. As we have advocated in the past [Smaragdakis 2004], independently of MAJ, this is an approach to designing domain specific languages that makes engineering sense. From a Software Engineering standpoint, the main advantages of expressing a concept as a domain-specific language feature, as opposed to a library API, are conciseness, safety, and performance. Nevertheless, changing the syntax and semantics of a programming language has many disadvantages. From the usability perspective, new features may cause backward compatibility issues for legacy code, and independently developed extensions might interact very badly with each other. From the language implementor's perspective, the engineering overhead is high (e.g., modifying the language grammar, modifying or writing new code to handle compiler backend tasks such as code transformation). Extending the language only through unobtrusive annotations prohibits undisciplined language extension and has the advantage that the generator is used very much as a library. The main code base is not polluted, and the generator can be modularly removed from the software development pipeline and replaced with a library, if the programmer so decides. Much of the en-

gineering overhead of developing a new language feature from scratch is eliminated by using annotations as the only syntax "extension", and by leveraging AspectJ as a transformational backend.

Overall, the value and novelty of Meta-AspectJ can be described in two axes: its application value (i.e., the big-picture value for potential users and transferable to users of other meta-aspect languages), and its technical contributions (i.e., smaller reusable lessons for other researchers working on meta-programming tools). In terms of application value, MAJ is a useful meta-programming tool, not just for AspectJ but also for Java in general. Specifically:

—MAJ can be used to implement domain specific language features using AspectJ as a back-end. We present several example applications, in generating Enterprise Java Beans code, in connection pooling, etc. MAJ is the only tool for structured generation of AspectJ programs that we are aware of. Thus, to combine the benefits of generative programming and AspectJ, one needs to either use MAJ, or to use a text-based approach.

—For generating either AspectJ or plain Java code, MAJ is safer than any text-based approach because the syntax of the generated code is represented explicitly in a typed structure.

—Compared to plain Java programs that output text strings, generators written in MAJ can be simpler because MAJ allows writing complex code templates using quote/unquote operators, instead of low-level string operations.

In terms of technical value, MAJ offers several improvements over prior meta-programming tools for Java (e.g., JTS [Batory et al. 1998]). These translate to ease of use for the MAJ user, while the MAJ language design offers insights for meta-programming researchers:

—MAJ shows how to minimize the number of different quote/unquote operators compared to past tools, due to the MAJ mechanism for inferring the syntactic type (e.g., expression, declaration, statement, etc.) of a fragment of generated code. This property requires context-sensitive parsing of quoted code: the type of an unquoted variable dictates how quoted code should be parsed. As a result, the MAJ implementation is quite sophisticated and not just a naive precompiler. An additional benefit of this approach is that MAJ emits its own error messages, independently from the Java compiler that is used in its back-end.

—When storing fragments of generated code in variables, the user does not need to specify the types of these variables (e.g., whether they are statements, expressions, etc.). Instead, a special `infer` type can be used.

The above points are important because they isolate the user from low-level representation issues and allow meta-programming at the template level.

We next present an overview of the MAJ language design (Section 2), discuss example language extensions that we have built with MAJ (Section 3), analyze the value of combining generative and aspect-oriented programming (Section 4), describe in more depth the individual interesting technical points of MAJ (Section 5), consider the general framework of our approach (Section 6), and discuss related work (Section 7).

2. MAJ OVERVIEW

2.1 Background: AspectJ

Aspect-oriented programming (AOP) is a methodology that advocates decomposing software by aspects of functionality. These aspects are modular representations of "cross-cutting" concerns: their functionality spans multiple functional units (functions, classes, modules, etc.) of the software application. Tool support for aspect-oriented programming consists of machinery for specifying such cross-cutting concerns separately from the main application code and subsequently composing them with that code.

AspectJ [Kiczales et al. 2001] is a general purpose, aspect-oriented extension of Java. AspectJ allows the user to define aspects to be merged ("weaved") with the rest of the application code. The power of AspectJ comes from the variety of ways in which aspects can interface with existing Java code. With AspectJ, the user can specify code that gets executed during semantic actions (e.g., method executions, field references, exception throwing), can emit static errors upon matching a code pattern (e.g., a call to a method inside a certain class), and more. Complex enabling predicates can be used to determine whether an aspect applies at a certain point. Such predicates can include, for instance, information on the identity of the caller and callee, whether a call to a method is made while a call to a certain different method is on the stack, etc. For a simple example of the syntax of AspectJ, consider the code below:

```
aspect CaptureUpdateCallsToA {
  static int num_updates = 0;

  pointcut updates(A a): target(a) && call(public * update*(..));

  after(A a): updates(a) {        // advice
    num_updates++;                // update was just performed
  }
}
```

The above code defines an aspect that just counts the number of calls to methods whose name begins with "update", taking any number of arguments (indicated by ".."), on objects of type `A`. The "pointcut" definition specifies where the aspect code will tie together with the main application code. The exact code ("advice") will execute after each call to an "update" method.

AspectJ also features *inter-type declarations*, which allow new methods, fields, interfaces, or a superclass to be *introduced* to an existing class or interface through an aspect. For example, the following AspectJ aspect introduces a method `void foo() {...}` to a class `C`:

```
aspect AddFooToC {
  void C.foo() {...}
}
```

AspectJ provides a `declare parents` construct to allow introduction of interfaces or a superclass. For example, the following aspect adds the interface `IFoo` to `C`:

```
aspect AddIFooToC {
  declare parents: C implements IFoo;
}
```

The separately declared members/parents are weaved into the existing type by the AspectJ compiler.

2.2 MAJ Basics

MAJ offers two variants of code-template operators for creating AspectJ code fragments: `‘[...]` ("quote") and `#[EXPR]` or just `#IDENTIFIER` ("unquote"). (The ellipses, `EXPR` and `IDENTIFIER` are meta-variables matching any syntax, expressions and identifiers, respectively.) The quote operator creates representations of AspectJ code fragments. Parts of these representations can be variable and are designated by the unquote operator (instances of unquote can only occur inside a quoted code fragment). For example, the value of the MAJ expression `‘[call(* *(..))]` is a data structure that represents the abstract syntax tree for the fragment of AspectJ code `call(* *(..))`. Similarly, the MAJ expression `‘[!within(#className)]` is a quoted pattern with an unquoted part. Its value depends on the value of the variable `className`. If, for instance, `className` holds the identifier "`SomeClass`", then the value of `‘[!within(#className)]` is the abstract syntax tree for the expression `!within(SomeClass)`.

MAJ also introduces a new keyword `infer` that can be used in place of a type name when a new variable is being declared and initialized to a quoted expression. For example, we can write:

```
infer pct1 = ‘[call(* *(..))];
```

This declares a variable `pct1` that can be used just like any other program variable. For instance, we can unquote it:

```
infer adv1 = ‘[void around() : #pct1 { }];
```

This creates the abstract syntax tree for a piece of AspectJ code defining (empty) advice for a pointcut. Section 2.3 describes in more detail the type inference process.

Since AspectJ is an extension of Java, any regular Java program fragment can be generated using MAJ. Furthermore, the values of primitive Java types (`int`s, `float`s, `double`s, etc.) and their arrays can be used as constants in the generated program. The unquote operator automatically promotes such values to the appropriate abstract syntax tree representations. For example, consider the code fragment:

```
void foo(int n) {
  infer expr1 = ‘[ #n * #n ];
  infer expr2 = ‘[ #[n*n] ];
  ...
}
```

If `n` holds the value 4, then the value of `expr1` is `‘[ 4 * 4 ]` and the value of `expr2` is `‘[ 16 ]`. Similarly, if `nums` is an array of Java `int`s with value `{1,2,3}` then the code fragment

```
infer arrdcl = ‘[ int[] arr = #nums; ];
```

will set `arrdcl` to the value `‘[ int [] arr = {1,2,3}; ]`.

The unquote operator can also be used with an array of expressions of non-primitive types. We call this variant of the operator "unquote-splice". The unquote-splice operator is used for adding arguments in a quoted context that expects a variable number of arguments (i.e., an argument list, a list of methods, or a block of statements). For example, if variable `argTypes` holds an array of type names, then we can generate code for a pointcut describing all methods taking arguments of these types as:

```
infer pct2 = ‘[call(* *(#[argTypes])];
```

That is, if `argTypes` has 3 elements where `argTypes[0]` is `int`, `argTypes[1]` is `String`, and `argTypes[2]` is `Object`, then the value of `pct2` will be the abstract syntax tree for the AspectJ code fragment `call(* *(int, String, Object))`.

We can now see a full MAJ method that generates a trivial but complete AspectJ file:

```
void generateTrivialLogging(String classNm) {
  infer aspectCode =
   ‘[   package MyPackage;
        aspect #[classNm + "Aspect"] {
          before : call(* #classNm.*(..))
          { System.out.println("Method called"); }
        }
    ];
  System.out.println(aspectCode.unparse());
}
```

The generated aspect causes a message to be printed before every call of a method in a class. The name of the affected class is a parameter passed to the MAJ routine. This code also shows the `unparse` method that our abstract syntax types support for creating a text representation of their code.[1] The abstract syntax types of the MAJ back-end[2] also support other methods for manipulating abstract syntax trees. One such method, `addMember`, is used fairly commonly: `addMember` is supported by syntactic entities that can have an arbitrary number of members (e.g., classes, interfaces, aspects, or argument lists). Although the high-level MAJ operators (quote, unquote, unquote-splice) form a complete set for generating syntax trees, it is sometimes more convenient to manipulate trees directly using the `addMember` method. One such use of `addMember` is shown in Figure 1, which we explain in detail in Section 3.2.

2.3 Types and Inference

We saw earlier an example of the MAJ keyword `infer`:

[1] `unparse` is a primitive way of integrating quoted code into compilation. More sophisticated and cohesive options are possible, and these are simply a matter of engineering.

[2] We currently use modified versions of the AspectJ compiler classes for the MAJ back-end, but may choose to replicate these classes in a separate MAJ package in the future.

```
infer adv1 = `[void around(): #pct1 {} ];
```

The inferred type of variable `adv1` will be `AdviceDec` (for "advice declaration"), which is one of the types for AspectJ abstract syntax tree nodes that MAJ defines. Such types can be used explicitly both in variable definitions and in the quote/unquote operators. For instance, the fully qualified version of the `adv1` example would be:

```
AdviceDec adv1 = `(AdviceDec)[void around(): #(Pcd)pct1 {} ];
```

The full set of permitted type qualifiers contains the following names: `IDENT`, `Identifier`, `NamePattern`, `Modifiers`, `Import`, `Pcd`, `TypeD`, `VarDec`, `JavaExpr`, `Stmt`, `MethodDec`, `ConstructorDec`, `ClassDec`, `ClassMember`, `InterfaceDec`, `DeclareDec`, `AdviceDec`, `CompilationUnit`, `PointcutDec`, `Pcd`, `AspectDec`, `FormalDec`, and `AspectMember`. Most of the qualifiers' names are self-descriptive, but a few require explanation: `IDENT` is an unqualified name (no dots), while `Identifier` is a full name of a Java identifier, such as `pack.clazz.mem`. `NamePattern` can be either an identifier, or a wildcard, or a combination of both. `Pcd` is for a pointcut body (e.g., `call(* *(..))`, as in our example) while `PointcutDec` is for full pointcut declarations (with names and the AspectJ `pointcut` keyword).

Although MAJ allows the use of type qualifiers, these are never necessary for uses of quote/unquote, as well as wherever the `infer` keyword is permitted. The correct types and flavor of the operators can be inferred from the syntax and type information. The goal is to hide the complexity of the explicit type qualifiers from the user as much as possible. Nevertheless, use of types is still necessary wherever the `infer` keyword is not allowed, notably in method signatures and in definitions of member variables that are not initialized. For example, the following uses of `infer` are *not* allowed:

```
// using infer on null-initialized variable.
infer expr = null;

// using infer in method signature.
infer getExpr() { ... }
```

Section 5.2.1 details the reasons for these limitations on the placement of `infer`.

3. APPLICATIONS

Most domain-specific languages and several kinds of software automation tools (wizards, design-to-code editors, middleware systems, etc.) are implemented as program generators [Cordy et al. 1991]. We next present three example applications of MAJ. Although one can use MAJ as just a meta-Java tool for all kinds of program generation, we selected these examples to showcase the unique emphasis of MAJ. All three examples are accomplished much more easily by using AspectJ than by ad hoc program generation. Yet, none can be expressed in AspectJ alone.

A common pattern of using MAJ is for implementing domain-specific language extensions using unobtrusive annotations. Recently, the introduction of user-defined annotations in mainstream programming languages, such as C# and Java, has allowed specialized language extensions (e.g., for distributed computing, persistence,

or real-time programming) to be added without changing the base syntax. This is a desirable property as it clearly separates the base code from the domain-specific extensions and their implementation. From a software engineering standpoint, this means that the domain-specific language implementation (i.e., the generator) introduces a smaller dependency in the software development process. When the domain-specific extension interfaces with the base application only through optional annotations, the generator adds or transforms code in a modular way. Effectively, the generator becomes a convenient substitute for code the programmer himself/herself could have written. If for any reason the application programmer wants to stop using the generator (e.g., because the generator is no longer maintained, or because the generated code now needs to support radically different needs) he/she can do so without affecting the rest of the code.

Using MAJ, the programmer can easily express an extension to the Java language as a program that: a) reads annotations and type information from an existing program using Java reflection; b) outputs a customized AspectJ aspect that is responsible for transforming the original program according to the information in the annotation; c) executes the generated aspect by using the standard AspectJ compiler. This approach leverages the engineering sophistication of the AspectJ compiler implementation and its provisions for dealing correctly with different Java language features. If a programmer were to replicate the same effort by hand, he/she would likely need to reproduce much of the AspectJ compiler complexity.

3.1 Filling Interface Methods

Our first language extension is simple but represents a good exposition example to our approach, since it can be defined very quickly and it is hard to implement with alternate means.

The Java language ensures that a class cannot declare to "implement" an interface unless it provides implementations for all of its methods. Nevertheless, this often results in very tedious code. For instance, it is very common in code dealing with the Swing graphics library to implement an event-listener interface with many methods, yet provide empty implementations for most of them because the application does not care about the corresponding events. The example code below is representative:

```
private class SomeListener
                implements MouseListener, MouseMotionListener {
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }

  // the rest are not needed. Provide empty bodies.
  public void mouseClicked (MouseEvent event) {}
  public void mouseReleased (MouseEvent event) {}
  public void mouseEntered (MouseEvent event) {}
  public void mouseExited (MouseEvent event) {}
```

```
  public void mouseMoved (MouseEvent event) {}
}
```

Of course, the programmer could avoid providing the empty method bodies on a per-interface basis, by associating each interface with a class that by default provides empty implementations of all interface methods. Then a client class can inherit the empty implementations and only provide implementations for the methods it needs. This pattern is indeed supported in Swing code (through library classes called *adapters*). Since Java supports single inheritance for classes, however, the listener class cannot already have another superclass, and can only use a single adapter class. Instead, it would be nice to provide a simple Java language extension implemented as an annotation [Gosling et al. 2005]. For example, we can use `@Implements({IFACE1, IFACE2, ...})` to annotate any class that should implement interfaces `IFACE1`, `IFACE2`, etc. The implementation of the extension would be responsible for finding the unimplemented methods and supplying empty implementations by default (or implementations that just return a default primitive or null value, in the case of methods that have a return type). In this case, the above class could be written more simply as:

```
@Implements ({"MouseListener", "MouseMotionListener"})
public class SomeListener {
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }
}
```

Of course, this extension should be used carefully since it weakens the tests of interface conformance performed by the Java compiler. (E.g., a programmer could forget to implement a method that should provide a non-default implementation. Whereas without this extension, the Java compiler would complain to the programmer and remind him/her that an implementation is needed, no such reminder is available when this extension is applied.)

We implemented the above Java extension using MAJ. The code for the implementation is less than 200 lines long, with most of the complexity in the traversal of Java classes, interfaces, and their methods using reflection. The code processes a set of given Java classes and retrieves the ones with an `@Implements` annotation. It then finds all methods that are in any of the interfaces passed as arguments to the `@Implements` annotation and are not implemented by the current class. For each such method, an AspectJ inter-type declaration is generated in an aspect to add an appropriate method implementation to the class. For instance, the code to add the method to the class in the case of a `void` return type is:

```
infer newMethod = `[ public void #methodName (#formals) {} ];
aspectMembers.add(newMethod);
```

Finally, the class needs to be declared to implement the interfaces specified by the annotation. This is easily done by emitting the appropriate AspectJ code:

```
infer dec =
 `[declare parents: #[c.getName()] implements #[iface.getName()]; ];
```

The final aspect (slightly simplified for formatting reasons) generated for our earlier listener class example is:

```
public aspect SomeListenerImplementsAspect1 {
  void SomeListener.mouseClicked(MouseEvent e) {}
  void SomeListener.mouseEntered(MouseEvent e) {}
  void SomeListener.mouseExited(MouseEvent e) {}
  void SomeListener.mouseMoved(MouseEvent e) {}
  void SomeListener.mouseReleased(MouseEvent e) {}

  declare parents:
     SomeListener implements MouseListener;
  declare parents:
     SomeListener implements MouseMotionListener;
}
```

This aspect performs exactly the modifications required to the original class so that it correctly implements the `MouseListener` and `MouseMotionListener` interfaces.

Note that MAJ does not provide a pattern language for matching on the structure of Java classes. Thus, our implementation of this language extension uses the regular Java Reflection API [Gosling et al. 2005], and the resulting reflection code can be verbose. For example, the following code is used to collect all methods implemented by a class `c`:

```
{ ...
  Vector methods = new Vector();
  java.lang.reflect.Method[] cMethods = c.getDeclaredMethods();
  if ( cMethods != null ) {
    for ( int i=0; i<cMethods.length; i++) {
      // only add the non-abstract methods.
      if ( !Modifier.isAbstract(cMethods[i].getModifiers()))
        methods.add(cMethods[i]);
    }
  }
  ...
}
```

A pattern language for reflection is orthogonal to the scope of this work, though could be incorporated into our approach.

Default implementations for interface methods is a well-recognized problem with languages such as Java, where no multiple inheritance (of classes) is possible. In fact, a language extension similar to the one described above has been proposed in previous literature by Mohnen [Mohnen 2002]. (Techniques such as traits [Ducasse et al. 2006] and mixins [Bracha and Cook 1990] can certainly be used to provide alternative solutions, but these are more general language features, whereas Mohnen's extension specifically addresses the default implementation problem, and

thus permits a more appropriate comparison to our extension.) Mohnen's extension allows programmers to provide default implementations for interface methods right in the *body* of the interface definition. Classes can then choose to "inherit" a default implementation using an "`= default`" syntax. The MAJ extension can be extended to emulate all of the functionality offered by the Mohnen extension (e.g., by allowing an annotation in which the programmer specifies which default method implementations to incorporate into a class). Yet, a comparison of the MAJ extension to the Mohnen proposal highlights both the elegance of language extensions through annotations, and the power of combining generative programming with an aspect language. Compared to the Mohnen extension, annotation-based MAJ extension is unobtrusive—it does not require changes to the Java language specification, nor does it require changes to the base language implementation. A programmer may easily choose to incorporate or remove our extension, by simply running (or not running) the MAJ code that generates the AspectJ aspects. The MAJ extension is also significantly easier to implement, since it reuses AspectJ for all the complex code transformations. Similarly, the MAJ extension can be easily extended for more expressiveness—e.g., a more expressive annotation can allow the extension writer to provide *different* default implementations for the same method.

We invite the reader to consider how else this language extension might be implemented. With AspectJ alone, a different aspect would need to be written for *each* class/interface combination. This provides no savings in coding effort at all. Even generic aspects, a new feature in AspectJ 1.5, are not powerful enough to allow a general aspect that can be applied to all class/interface combinations. Alternatively, we could use more general transformation/generation tools, such as Stratego/XT [Bravenboer et al. 2005] or TyRuBa [Volder 1998]. Stratego/XT is a good representative of modern and powerful pattern-based program transformation tools. Using pattern-based transformations is often convenient, and even concise, for many tasks, although harder for tasks not supported well with pre-existing patterns. Furthermore, using Stratego/XT is a radical departure from our incremental approach and requires the programmer to think in terms of transformations. For instance, to use Stratego/XT, we would first need to develop a GLR grammar for Java (or understand well the internal constructs of an existing one), and then use Stratego/XT's rewrite system in order to perform the transformations. Both tasks have non-negligible learning curves and do not benefit from the mature engineering support (e.g., for newer versions of Java) that AspectJ enjoys. In contrast, our approach of using annotations in combination with MAJ yields a very simple implementation by letting AspectJ deal with most of the complexities of Java. Specifically, we did not have to deal with the low-level complexities of either Java source syntax or Java bytecode. For instance, we did not have to do any code parsing to find the class body or declaration that needs to be modified. Dealing with Java syntactic sugar, such as inner classes, was automatic. We did not need to do a program transformation to add the `implements` clauses or the extra methods to the class. Similarly, we did not need to worry about the valid syntax for adding an implemented interface if the class already implements another.

3.2 Automatically Preparing a Class for Distribution

In previous work [Tilevich et al. 2003] we presented the GOTECH generator for the domain of distributed computing. GOTECH accepts a Java program annotated with JavaDoc comments to describe what parts of the functionality should be remotely executable. It then transforms parts of the program so that they execute over a network instead of running on a local machine. The middleware platform used for distributed computing is J2EE (the protocol for Enterprise Java Beans—EJB). GOTECH takes care of generating code adhering to the EJB conventions and makes methods, construction calls, etc. execute on a remote machine. Internally, the modification of the application is performed by generating AspectJ code that transforms existing classes.

GOTECH has been one of the motivating examples in the design of MAJ. GOTECH has several shortcomings related to its use of JavaDoc comments and its text-based program generation using the XDoclet tool [Stevens et al.]. An approach using Java annotations and MAJ completely overcomes these shortcomings: the language extension is more robust and the code generation is safer. (In Section 5 we discuss extensively why MAJ is better than text-based program generation.) During the development of MAJ we have replicated the entire functionality of GOTECH using MAJ and have used it as a regression test. We next show in detail one of the program generation tasks of GOTECH, as reimplemented in MAJ. This is a good example, in that it is realistic yet short enough to be shown in full.

When a method needs to support Java remote invocation, all of its arguments must implement interface `java.io.Serializable` (if the argument type is not a primitive type). For every class that can be potentially accessed remotely, GOTECH finds all the types of its methods' arguments and if they do not already implement `java.io.Serializable`, it causes them to do so. The transformation is triggered by the `@makeRemote` annotation in the source code. For example, imagine the following class that needs to be prepared for remote invocation. We annotate it with `@makeRemote`:

```
@makeRemote
class SomeClass {
  public void meth1(Car c) { ... }
  public void meth2(int i, Tire t) { ... }
  public void meth3(float f, Seat s) { ... }
}
```

The implementation of `@makeRemote` is a piece of MAJ code that crawls the classpath for classes annotated with `@makeRemote`. Upon seeing `SomeClass`, it calls the MAJ code in Figure 1 which traverses all of `SomeClass`'s methods, and creates an aspect that makes each method argument type implement the interface `java.io.Serializable` (provided the argument type does not implement this interface already and it is not a primitive type).[3] In this case, the list of all argument

[3]This extension does not take into account classes for which special handling is needed in serialization, i.e., classes that need non-default implementations of `writeObject` and `readObject`. A more complex mechanism could handle those, unless the handling is truly class-specific, in which case MAJ would offer no leverage over manual AspectJ aspects.

```
import java.io.*;
import java.lang.reflect.*;
import org.aspectj.compiler.base.ast.*;
import org.aspectj.compiler.crosscuts.ast.*;

public class MAJGenerate {
 public static void genSerializableAspect(Class inClass, PrintStream out)
 {
  String aspectName = inClass.getName() + "SerializableAspect";

  // Create a new aspect
  infer serializedAspect = `[aspect #aspectName {}];

  // Add Serializable to every method argument type that needs it
  for (int meth = 0; meth < inClass.getMethods().length; meth++) {
    Class[] methSignature = inClass.getMethods()[meth].getParameterTypes();
    for (int parm = 0; parm < methSignature.length; parm++) {
      if (!methSignature[parm].isPrimitive() &&
          !Serializable.class.isAssignableFrom(methSignature[parm]))
        serializedAspect.addMember(
         `[ declare parents:
            #[methSignature[parm].getName()] implements java.io.Serializable;
         ] );
    } // for all params
  } // for all methods

  infer compU = `[ package gotech.extensions;
                   #serializedAspect
                 ];

  out.print(compU.unparse());
 }
}
```

Fig. 1. A routine that generates an aspect that makes method parameter types be serializable

types is `int`, `float`, `Car`, `Tire`, and `Seat`. The first two are primitive types, thus the MAJ program will generate the following AspectJ code:

```
package gotech.extensions;
aspect SerializableAspect {
  declare parents: Car implements java.io.Serializable;
  declare parents: Tire implements java.io.Serializable;
  declare parents: Seat implements java.io.Serializable;
}
```

This example is a good representative of realistic uses of MAJ, and demonstrates how MAJ could be used to implement extensions to AspectJ itself. The extension is implemented in the form of the `@makeRemote` annotation, and the extension

is a more flexible joinpoint that recognizes all non-primitive types in a method signature that do not implement `Serializable` already. This example illustrates that the cases where AspectJ alone is not sufficient are exactly those where complex conditions determine the existence, structure, or functionality of an aspect. Observe that most of the code in Figure 1 is not program generating code. Instead, it concerns the application logic for finding argument types and deciding whether a certain type should be augmented to implement interface `Serializable`. MAJ makes the rest of the code be straightforward.

3.3 Language support for object pooling

Our last example language extension addresses a common programming need, especially in server-side programming. Software applications often find the need for pooling frequently-used objects with high instantiation costs. We use the following database connection class as a running example:

```
public class DBConnection {
  public DBConnection(String dbURI,
                      String userName,
                      String password ) { ... }
  public void open() { ... }
  public void close() { ... }
}
```

The cost of an `open()` call is very high for a database connection. In applications concerned with performance, such as high-volume websites with lots of database requests, one often finds the need to pool database connections and keep them open, instead of repeatedly creating new ones and opening them.

Making a class such as `DBConnection` into a "pooled" class involves at the very least creating a pooling manager class that knows how to manage instances of the class being pooled. A different pooling manager class needs to be developed for each class being pooled, since the manager needs to have class-specific information such as how to instantiate a new instance of the class when the pool is running low (e.g., `DBConnection` objects are created by a constructor call, followed by an `open()` call), and how to uniquely identify objects of the same class that belong to different pools (e.g., `DBConnection` objects of different `dbURI`, `userName`, and `password` combinations need to be in different pools, and the pooling manager needs to understand which pool to fetch objects from when a request arrives). Each pooling manager class differs only in the way they instantiate and manage new objects. With current programming methodologies, the best way to reuse the shared features is by extending an abstract pooling manager class, with an abstract `instantiate(...)` method. Each concrete pooling manager class can then implement `instantiate(...)` according to the class' needs. However, this process can be easily mechanized using meta-programming combined with the ability to do reflection.

In addition to the need for a separate pooling manager class, if a large code base has been developed using the class in its un-pooled form, programmers need to find each constructor site for the original class, and change it to a request to the pooling manager. Likewise, when the object is no longer needed, programmers need to add

code to return the object to the pooling manager. These are again very tedious programming efforts that can be simplified by a language extension, as we will show next.

We expressed the pooling concept as a language feature that can be used transparently with any Java class, as long as some broad properties hold regarding its construction and instantiation interface. The rest of the application will be completely oblivious to the change. This facilitates the application of pooling after a large code base which uses the class in its non-pooled form has been developed. Using our extension, converting a class to a pooled class involves only 4 annotations: `@pooled`, `@constructor`, `@request`, and `@release`. For example, to convert the `DBConnection` class into a "pooled" class, and to adapt an existing code base to using the pooled functionality, the user only has the add the following annotations to the code:

```
@pooled(mgr=pooled.PoolTypes.BASIC, max=10, min=2)
public class DBConnection {
  @constructor
  public DBConnection(String dbURI,
                      String userName,
                      String password ) { ... }
  @request
  public void open() { ... }
  @release
  public void close() { ... }
}
```

The `@pooled` annotation indicates that the class `DBConnection` should be pooled. It accepts parameters that can be used to customize the pooling policy. `@constructor` annotates the constructor call whose parameters serve as unique identifiers for different kinds of `DBConnection` objects. In this example, `DBConnection` objects with different `dbURI`, `userName`, and `password` combinations should be maintained separately. `@request` annotates the method that signals for the request of a pooled object, and `@release` annotates the method call that signals for the return of the pooled object back to the pooling manager.

The implementation of this language extension using MAJ is less than 400 lines of code. The MAJ program searches for classes annotated with `@pooled`, and generates two Java classes and one AspectJ aspect to facilitate converting this class to be pooled. We next describe the generated code in more detail. The reader may want to consider in parallel how the same task could be accomplished through other means. Neither conventional Java facilities (i.e., classes and generics) nor AspectJ alone would be sufficient for expressing the functionality we describe below in a general way, so that it can be applied with little effort to arbitrary unmodified classes. For instance, none of these facilities can be used to create a proxy class with methods with identical signatures as those of an arbitrary Java class.

First, a pooling manager class, `PoolMgrForDBConnection`, is generated for `DBConnection`. The pooling manager class contains methods for requesting and releasing pooled `DBConnection` objects, as well as code to manage the expansion of the pool based on the `min` and `max` parameters.

In order to retrofit an existing code base to use `DBConnection` as a pooled class, we need to introduce proxy objects that will be used wherever an object of the original class would exist in the application code. This is necessary as different objects from the perspective of the client code will correspond to the same pooled object. We generate a proxy class as a subclass of the pooled class. In our example: `DBConnection_Proxy extends DBConnection`. All instances of the proxy class share a static reference to an instance of `PoolMgrForDBConnection`. Each proxy instance holds (non-static) references to the parameters to the `@constructor` constructor call, and the `DBConnection` object obtained from the pooling manager. The proxy class rewrites the `@request` and `@release` methods: the `@request` method is rewritten to obtain an object of `DBConnection` type from the pooling manager, using the unique identifiers kept from the constructor call, and the `@release` method returns the `DBConnection` object back to the pool, while setting the reference to this object to null. The MAJ code generating the proxy takes care to exactly replicate the signature of the original methods, including modifiers and `throws` clauses. For instance, the `@release` method in the proxy is generated as:

```
infer meth =
 `[ #mods #ret #[m.getName()] (#formals) #throwsClause
    {
      m_poolMgr.release(m_uniqueId, m_proxiedObj);
      m_proxiedObj = null;
    }];
```

All other methods simply delegate to the same method in the superclass.

The idea is to have variables declared to hold a `DBConnection` object, now hold a `DBConnection_Proxy` object. Therefore, to complete the *proxy* pattern, we need to change all the calls of `new DBConnection(...)` to `new DBConnection_Proxy(...)`. This tedious recoding effort is easily replaced by a generated aspect:

```
public aspect DBConnection_Proxy_Aspect {
  DBConnection around(String arg1, String arg2, String arg3) :
    call (DBConnection.new(String, String, String)) &&
    args(arg1) && args(arg2) && args(arg3) &&
    !cflowbelow(execution(public DBConnection
                          PoolMgrForDBConnection.request(..))) {
      return new DBConnection_Proxy(arg1, arg2, arg3);
    }
}
```

The aspect intercepts all the constructor calls of `DBConnection`, provided the call is not made from within the pooling manager class `PoolMgrForDBConnection` (this is the purpose of the `!cflowbelow(...)` construct), and returns an object instantiated by calling `new DBConnection_Proxy(...)`. This ensures that no one except the pooling manager instantiates new `DBConnection` objects, and all other requests for `DBConnection`, made through the `@request` methods, are turned into a request to the pooling manager, which simply returns a `DBConnection` object already opened.

In summary, a user can easily turn a class into a pooled class, *and* retrofit any existing code base to use this class in its new, pooled form. The client code does not need to be hand-edited at all, other than with the introduction of our 4 annotations.

An alternative implementation would be to not use the proxy pattern at all. Instead of returning a new `DBConnection_Proxy` object for each constructor call on `DBConnection`, we could generate an aspect that intercepts the constructor call, and simply returns the pooled `DBConnection` object itself. However, this approach has more restrictions—for instance, it may change the original program's semantics, since two `DBConnection` objects with originally different identities now have the same identity. Using the proxy pattern, we can maintain program semantics by keeping separate object identities.

Regardless of one's decision regarding the proxy pattern, it is important to point out that AspectJ alone cannot accomplish the tasks necessary in this example. In order to keep track of the different objects being pooled, the aspect needs to be able to explicitly refer to the uniquely identifying keys (arguments in constructor call for the pooled class) for pooled objects. For an aspect to be generic enough to handle different classes, it needs to be able to both intercept constructor calls taking varying argument lengths, *and* be able to refer to these arguments and use them to look up the pooled objects. While AspectJ allows interception of methods without specifying the exact argument types (by using "`..`"), it does not allow iterating over these arguments. Thus, using AspectJ alone, a different aspect needs to be written for each class wishing to be pooled.

4. DISCUSSION

The applications of the previous section raise the more general question of what is the value of combining generative and aspect-oriented programming. One can ask why a generator cannot just perform the required modifications to a program without an aspect-oriented language, using meta-programming techniques alone. Similarly, one can ask why an aspect-oriented language alone is not sufficient for the desired tasks. We next abstract away from the specifics of our previous examples and offer some more general observations.

4.1 Why Do we Need an Aspect-Oriented Language?

A generator or program transformer could completely avoid the use of an aspect language and instead manipulate the structure of a program directly. Nevertheless, aspect-oriented languages often give very convenient vocabulary for talking about program transformations, as well as mature implementations of such transformations. For example, AspectJ lets its users transform many points of the program, such as all references to a member, all calls to a set of methods, etc. If a generator was to reproduce this functionality without AspectJ, the generator would need to parse all the program files, recognize all transformation sites, and apply the rewrites to the syntax. These actions are not simple for a language with the syntactic and semantic complexity of Java. For instance, generator writers often need to use functionality similar to the AspectJ `cflow` construct. `cflow` is used for recognizing calls under the control flow of another call (i.e., while the latter is still on the execution stack). Although this functionality can be re-implemented from scratch by adding a run-time flag, this would be a tedious and ad hoc replication

of the AspectJ functionality. It is much better to inherit a general and mature version of this functionality from AspectJ. AspectJ is also particularly well-suited as a transformation toolkit because of its capability for inter-type declarations (of methods, fields, interfaces, and superclasses), which we used in the examples of the previous section.

We should point out that, although one can use AspectJ (and aspect languages in general) as a "bag of program transformation tricks", this is neither the intention nor the real nature of aspect languages. Aspect-oriented programming explicitly discourages thinking in terms of transformations, and aspect languages avoid exposing the underlying program manipulations they perform. Instead, the goal of aspect-oriented programming is to export a higher-level abstraction and only allow expressing changes in terms of the *behavior* of a program and not its *structure*. For instance, with AspectJ, the user cannot arbitrarily change the body of a method, or place code at specific syntactic locations, but can augment what happens when a method is called. This is a limitation, since more general transformation languages such as Stratego/XT [Bravenboer et al. 2005] allow more fine-grained, statement-, and even expression-level transformations. However, we consider this limitation to be an excellent property for generators. We believe that a generator needs to produce modular code that does not depend too heavily on the low-level specifics of the original code it transforms. This prevents the generator from being too brittle and from needing to interact in complex ways with all the language features of the host language.

4.2 Why Do we Need Meta-Programming?

Although aspect-oriented languages vary in their expressiveness and power, there are always useful operations that an aspect-oriented language alone cannot handle. For example, AspectJ cannot be used to create an interface isomorphic to the public methods of a given class (i.e., a new interface whose methods correspond one-to-one to the public methods of a class). This is an essential action for a tool like GOTECH that needs to create new interfaces (home and remote interfaces, per the EJB conventions) for existing classes. GOTECH was used to automate activities that were previously [Soares et al. 2002] shown impossible to automate with just AspectJ.

At the time that MAJ was first developed, AspectJ did not have the capabilities of matching on annotations in defining joint points. The current version of AspectJ, however, does allow joint point matching based on annotations. This adds to AspectJ some extra flexibility: instead of matching purely on a method signature, one can now specify a joint point of methods that all have a particular annotation, but with nothing else common in their signature that could have been expressed using AspectJ before. Although this is an improvement, the fundamental limitations of AspectJ that we discussed in previous sections still exist. Using AspectJ with annotations alone, none of the three example applications we showed can be achieved. For example, in the `@Implements` example, one still cannot introduce a method into a class based on this annotation alone—one still needs to programmatically distinguish which methods in the implemented interfaces are missing, and thus need to be introduced.

In general, using meta-programming allows us to go beyond the capabilities of

AspectJ by adding arbitrary flexibility in recognizing where aspects should be applied and customizing the weaving of code. For instance, AspectJ does not allow expressing joinpoints based on properties such as "all classes with native methods", "all methods in classes that extend a system class", etc. Methods embodying these properties are simple to recognize using Java reflection, and, consequently, flexible joinpoints such as these can be defined using MAJ, by generating appropriate AspectJ programs for the methods. Similarly, AspectJ does not allow aspects to be flexible with respect to what superclasses they add to the class they affect, whether added fields are private or public, etc. This information is instead hard-coded in the aspect definition. With a meta-program written in MAJ, the generated aspect can be adapted to the needs of the code body at hand.

5. META-ASPECTJ DESIGN AND IMPLEMENTATION

5.1 MAJ Design

We will next examine the MAJ design a little closer, in order to compare it to other meta-programming tools.

5.1.1 *Structured vs. Unstructured Meta-Programming.* Structured meta-programming tools represent quoted code fragments in a syntactically (and, sometimes, semantically) meaningful way. The Lisp language [Guy L. Steele 1990] is one of the oldest and best known representatives of structured meta-programming. Lisp provides operators quote (`'`) and unquote (`,`) for creating and splicing code fragments. Parsing and representing these code fragments is easy in Lisp: the surface syntax is uniformly represented by lists. Quote and unquote constructs were later added to languages with much richer syntax, e.g., MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003] for ML/OCaml, JTS [Batory et al. 1998], JSE [Bachrach and Playford 2001], and Maya [Baker and Hsieh 2002] for Java, and more. These tools overcome difficulties introduced by their rich syntax by either restricting the granularity of code that can be quoted/unquoted, or forcing programmers to explicitly state the type of code fragment being quoted/unquoted, or both. The common ground, though, among all these tools is that they maintain structural information about the quoted code fragments. Unstructured meta-programming tools, on the other hand, usually store code fragments as strings. Many text-based program generation tools are unstructured, e.g., XDoclet [Stevens et al.] or the C preprocessor.

Maintaining information about the syntax and semantics of quoted/unquoted fragments gives structured meta-programming tools the power to make certain guarantees about the well-formedness of code fragments being constructed. The MAJ operators ensure that all trees manipulated by a MAJ program are syntactically well-formed, although they may contain semantic errors, such as type errors or scoping errors (e.g., references to undeclared variables). That is, MAJ is based on a context-free grammar for describing AspectJ syntax. The MAJ expressions created using the quote operator correspond to words ("words" in the formal languages sense) produced by different non-terminals of this context-free grammar. Compositions of abstract syntax trees in ways that are not allowed by the grammar is prohibited. Thus, using the MAJ operators, one cannot create trees that do not correspond to fragments of AspectJ syntax. For instance, there is no way to create

a tree for an "if" statement with 5 operands (instead of 3, for the condition, then-branch, and else-branch), or a class with a statement as its member (instead of just methods and instance variable declarations), or a declaration with an operator in the type position, etc. The syntactic well-formedness of abstract syntax trees is ensured statically when a MAJ program is compiled. For example, suppose the user wrote a MAJ program containing the declarations:

```
infer pct1 = `[call(* *(..))];
infer progr = `[ package MyPackage;
                   #pct1
                 ];
```

The syntax error (pointcut in unexpected location) would be caught when this program would be compiled with MAJ.

The static enforcement of syntactic correctness for the generated program is a common and desirable property in meta-programming tools. It is often described as: the type safety of the generator implies the syntactic correctness of the generated program. The property is desirable because it increases confidence in the correctness of the generator under all inputs (and not just the inputs with which the generator writer has tested the generator). Thus, as a structured meta-programming tool, MAJ is superior to text-based tools in terms of safety.

Additionally, MAJ is superior in terms of expressiveness to text-based generation with a tool like XDoclet [Stevens et al.]. MAJ programs can use any Java code to determine what should be generated, instead of being limited to a hard-coded set of attributes and annotations. Compared to arbitrary text-based generation with plain Java strings, MAJ is more convenient. Instead of putting Java strings together with the "+" operator (and having to deal with low-level issues like explicitly handling quotes, new lines, etc.) MAJ lets the user use convenient code templates.

Of course, static type safety implies that some legal programs will not be expressible in MAJ. For instance, we restrict the ways in which trees can be composed (i.e., what can be unquoted in a quote expression and how). The well-formedness of an abstract syntax tree should be statically verifiable from the types of its component parts—if an unquoted expression does not have the right type, the code will not compile even if the run-time value happens to be legal. Specifically, it is not possible to have a single expression take values of two different abstract syntax tree types. For example we cannot create an abstract syntax tree that may hold either a variable definition or a statement and in the cases that it holds a definition use it in the body of a class (where a statement would be illegal).

5.1.2 *Qualifier Inference.* MAJ is distinguished from other meta-programming tools because of its ability to infer qualifiers for the quote/unquote operators, as well as the ability to infer types for the variables initialized by quoted fragments. Having multiple quote/unquote operators is the norm in meta-programming tools for languages with rich surface syntax (e.g., meta-programming tools for Java [Batory et al. 1998], C [Weise and Crew 1993], and C++ [Chiba 1995]). For instance, let us examine the JTS tool for Java meta-programming—the closest comparable to MAJ. JTS introduces several different kinds of quote/unquote operators: `exp{...}exp`, `$exp(...)`, `stm{...}stm`, `$stm(...)`, `mth{...}mth`, `$mth(...)`,

`cls{...}cls`, `$cls(...)`, etc. Additionally, just like in MAJ, JTS has distinct types for each abstract syntax tree form: `AST_Exp`, `AST_Stmt`, `AST_FieldDecl`, `AST_Class`, etc. Unlike MAJ, however, the JTS user needs to always specify explicitly the correct operator and tree type for all generated code fragments. For instance, consider the JTS fragment:

```
AST_Exp x = exp{ 7 + i }exp;
AST_Stm s = stm{ if (i > 0) return $exp(x); }stm;
```

This written in MAJ is simply:

```
infer x = `[7 + i];
infer s = `[if (i > 0) return #x ;];
```

The advantage is that the user does not need to tediously specify what flavor of the operator is used at every point and what is the type of the result. Consider a comparison with prior tools: JTS allows 17 different kinds of syntactic entities to be quoted/unquoted. Weise and Crew's programmable macros for C [Weise and Crew 1993] allow at least 6 different kinds. Both tools require programmers to know *all* the different AST types for these entities. MAJ allows 23 different syntactic entities to be quoted/unquoted, and requires its programmers to know *none* of the specific AST types used to represent these entities. MAJ instead infers this information. As we explain next, this requires sophistication in the parsing implementation.

5.2 MAJ Implementation

We have invested significant effort in making MAJ a mature and user-friendly tool, as opposed to a naive pre-processor. This section describes our implementation in detail.

5.2.1 *Qualifier Inference and Type System.* It is important to realize that although multiple flavors of quote/unquote operators are common in syntax-rich languages, the reason for their introduction is purely technical. There is no fundamental ambiguity that would occur if only a single quote/unquote operator was employed. Nevertheless, inferring qualifiers, as in MAJ, requires the meta-programming tool to have a full-fledged compiler instead of a naive pre-processor. (An alternative would be for the meta-programming tool to severely limit the possible places where a quote or unquote can occur in order to avoid ambiguities. No tool we are aware of follows this approach.) Not only does the tool need to implement its own type system, but also parsing becomes context-sensitive—i.e., the type of a variable determines how a certain piece of syntax is parsed, which puts the parsing task beyond the capabilities of a (context-free) grammar-based parser generator.

To see the above points, consider the MAJ code fragment:

```
infer l = `[ #foo class A {} ];
```

The inferred type of `l` depends on the type of `foo`. For instance, if `foo` is of type `Modifiers` (e.g., it has the value `` `[public] ``) then the above code would be equivalent to:

```
ClassDec l = `[ #(Modifiers)foo class A {} ];
```

If, however, `foo` is of type `Import` (e.g., it has the value `‘[import java.io.*;]`) then the above code would be equivalent to:

```
CompilationUnit l = ‘[ #(Import)foo class A {} ];
```

Thus, to be able to infer the type of the quoted expression we need to know the types of the unquoted expressions. This is possible because MAJ maintains its own type system (i.e., it maintains type contexts for variables during parsing). The type system is simple: it has a fixed set of types with a few subtyping relations and a couple of ad hoc conversion rules (e.g., from Java strings to `IDENT`s). Type inference is quite straightforward: when deriving the type of an expression, the types of its component subexpressions are known and there is a most specific type for each expression. We put restrictions on the placement of keyword `infer` to avoid recursion in the inference logic. The `infer` keyword can only be used in variable declarations where the variables are initialized with non-`null` expressions. Since Java disallows both the use of a variable in its own initialization and forward reference of variables yet to be declared, this restriction conveniently avoids recursion in type inference. Similarly, `infer` cannot be used in method signature declarations—methods in Java could be mutually-recursively defined, and having `infer` in such methods' signatures could yield recursion in the inference logic.

The types of expressions even influence the parsing and translation of quoted code. Consider again the above example. The two possible abstract syntax trees are not even isomorphic. If the type of `foo` is `Modifiers`, this will result in an entirely different parse and translation of the quoted code than if the type of `foo` is `Import` (or `ClassDec`, or `InterfaceDec`, etc). In the former case, `foo` just describes a modifier—i.e., a branch of the abstract syntax tree for the definition of class `A`. In the latter case, the abstract syntax tree value of `foo` is at the same level as the tree for the class definition.

5.2.2 *Parser Implementation.* The current implementation of the MAJ front-end consists of one common lexer and two separate parsers. The common lexer recognizes tokens legal in both the meta-language (Java), and the object language (AspectJ). This is not a difficult task for this particular combination of meta/object languages, since Java is a subset of AspectJ. For two languages whose token sets do not match up as nicely, a more sophisticated scheme would have to be employed.

We use ANTLR [Parr and Quong 1995] to generate our parser from two separate LL(k) grammars (augmented for context-sensitivity, as described below). One is the Java' grammar: Java with additional rules for handling quote and `infer`. The other is the AspectJ' grammar: AspectJ with additional rules for handling `infer`, quote and unquote. Java', upon seeing a quote operator lifts out the string between the quote delimiters (`‘[...]`) and passes it to AspectJ' for parsing. AspectJ', upon seeing an unquote, lifts out the string between the unquote delimiters and passes it to Java' for parsing. Thus, we are able to completely isolate the two grammars. This paves way for easily changing the meta or object language for future work, with the lexer caveat previously mentioned.

The heavy lifting of recognizing and type-checking quoted AspectJ is done in AspectJ'. To implement context-sensitive parsing we rely on ANTLR's facilities for guessing as well as adding arbitrary predicates to grammar productions and back-

tracking if the predicates turn out to be false. Each quote entry point production is preceded by the same production wrapped in a guessing/backtracking rule. If a phrase successfully parses in the guessing mode and the predicate (which is based on the types of the parsed expressions) succeeds, then real parsing takes place and token consumption is finalized. Otherwise, the parser rewinds and attempts parsing by the next rule that applies. Thus, a phrase that begins with a series of unquoted entities might have to be guess-parsed by a number of alternate rules before it reaches a rule that actually applies to it.

The parsing time of this approach depends on how many rules are tried unsuccessfully before the matching one is found. In the worst case, our parsing time is exponential in the nesting depth of quotes and unquotes. Nevertheless, we have not found speed to be a problem in MAJ parsing. The parsing time is negligible (a couple of seconds on a 1.4GHz laptop) even for clearly artificial worst-case examples with a nesting depth of up to 5 (i.e., a quote that contains an unquote that contains a quote that contains an unquote, and so on, for 5 total quotes and 5 unquotes). Of course, more sophisticated parsing technology can be used in future versions, but arguably parsing speed is not a huge constraint in modern systems and we place a premium on using mature tools like ANTLR and expressing our parsing logic declaratively using predicates.

5.2.3 *Translation.* The MAJ compiler translates its input into plain Java code. This is a standard approach in meta-programming tools [Batory et al. 1998; Batory et al. 2003; Visser 2002]. For example, consider the trivial MAJ code fragment:

```
infer dummyAspect = ‘[ aspect myAspect { } ];
infer dummyUnit = ‘[ package myPackage;
                     #dummyAspect
                   ];
```

MAJ compilation will translate this fragment to Java code using a library for representing AspectJ abstract syntax trees. The Java compiler is then called to produce bytecode. The above MAJ code fragment will generate Java code like:

```
AspectDec dummyAspect =
  new AspectDec(
    null, "myAspect", null, null, null,
    new AspectMembers(new AspectMember[] {null}));
CompilationUnit dummyUnit =
  new MajCompilationUnit(
   new MajPackageExpr(new Identifier("myPackage")),
   null, new Decs(new Dec[] { dummyAspect }));
```

The Java type system could also catch ill-formed MAJ programs. If the unquoted expression in the above program had been illegal in the particular syntactic location, the error would have exhibited itself as a Java type error. Nevertheless, MAJ implements its own type system and performs error checking before translating its input to Java. Therefore, the produced code will never contain MAJ type errors. (There are, however, Java static errors that MAJ does not currently catch, such as access protection errors, uninitialized variable errors, and more.)

5.2.4 *Error Handling.* Systems like JTS [Batory et al. 1998] operate as simple pre-processors and delegate the type checking of meta-programs to their target language (e.g., Java). The disadvantage of this approach is that error messages are reported on the generated code, which the user has never seen. Since MAJ maintains its own type system, we can emit more accurate and informative error messages than those that would be produced for MAJ errors by the Java compiler.

Recall that our parsing approach stretches the capabilities of ANTLR to perform context-sensitive parsing based on the MAJ type system. To achieve good error reporting we had to implement a mechanism that distinguishes between parsing errors due to a mistyped unquoted entity and regular syntax errors. While attempting different rule alternatives during parsing, we collect all error messages for failed rules. If parsing succeeds according to some rule, the error information from failed rules is discarded. If all rules fail, we re-parse the expression with MAJ type checking turned off. If this succeeds, then the error is a MAJ type error and we report it to the user as such. Otherwise, the error is a genuine syntax error and we report to the user the reasons that caused each alternative to fail.

5.2.5 *Implementation Evolution and Advice.* It is worth briefly mentioning the evolution of the implementation of MAJ because we believe it offers lessons for other developers. The original MAJ implementation was much less sophisticated than the current one. The system was a pre-processor without its own type system and relied on the Java compiler for ensuring the well-formedness of MAJ code. Nevertheless, even at that level of sophistication, we found that it is possible to make the system user-friendlier with very primitive mechanisms.

An important observation is that type qualifier inference can be performed even without maintaining a type system, as long as ambiguous uses of the unquote operator are explicitly qualified. That is, qualifying some of the uses of unquote allows having a single unqualified quote operator and the `infer` keyword. For instance, consider the code fragment:

```
infer s = `[if (i > 0) return #x ;];
```

The syntactic types of `s` and `x` are clear from context in this case. Even the early implementation of MAJ, without its own type system, could support the above example. Nevertheless, in cases where parsing or type inference would be truly dependent on type information, earlier versions of MAJ required explicit qualification of unquotes—for instance:

```
infer l = `[ #(Modifiers)foo class A {} ];
```

Even with that early approach, however, parsing required unlimited lookahead, making our grammar not LL(k).

6. GENERALIZING: BEYOND JAVA AND ASPECTJ

It is interesting to consider the Meta-AspectJ approach in a more general framework. What aspect languages are good candidates for combining with meta-programming in an approach similar to that of MAJ? Are there specific features of Java that MAJ exploits, or is the approach applicable in a variety of host languages? Are statically typed languages more natural targets than dynamic languages?

Our view is that there is no a priori limitation on the mechanisms, aspect languages, or mainstream (host) languages that can be successfully used as target languages for a meta-programming facility. In fact, there are some very broad tendencies that make the interplay of the AOP and meta-programming paradigms particularly useful. Of course, specific features of aspect languages or host languages can vary the amount of benefit one can expect, but this is a more general phenomenon, also applicable to AOP on its own.[4] Yet in principle, we expect an approach much like MAJ to be applicable in any programming language with a mature aspect-oriented programming mechanism (e.g., AspectC++ [Lohmann et al. 2004], or Eos for C# [Rajan and Sullivan 2003]).

The tendencies represented by AOP and meta-programming are those of high-level expressiveness and generality. The two tendencies are often in competition: a highly general mechanism typically ends up too undisciplined and low-level. In fact, aspect-oriented programming can be viewed as a disciplined outgrowth of powerful, yet low-level mechanisms such as Lisp macros and the CLOS meta-object protocol. At the same time, there are multiple capabilities, both syntactic (e.g., defining new keywords) and semantic (e.g., dynamically varying inheritance hierarchies and the object model itself) that aspect-oriented programming mechanisms typically do not support, in order to maintain their high-level character.

Philosophically, both aspect-oriented programming and meta-programming aim to address shortcomings of a host programming language. Aspect-oriented programming is used to express concerns that are not sufficiently modularized in a given language, or more generally in a dominant language paradigm. Meta-programming is often viewed as a catch-all way to overcome expressiveness limitations of a language. Indeed meta-programming is often seen as an undisciplined approach and derided as a "crutch" for languages with limited expressiveness.

It is these general features of AOP and meta-programming that make them ideal for a combination, much along the lines of MAJ. In a sense, our approach results in a conservative combination and not in a seamless merging of the two paradigms. Aspect-oriented programming mechanisms in any language address some of the language's limitations (typically relating to cross-cutting concerns). This raises the level of application for meta-programming mechanisms. A meta-programming system does not need to solve the problems already solved by an AOP mechanism. Instead, it can use the AOP mechanism as a target for code generation. This has the advantage of reusing mature, high-level, and often well-designed and declarative mechanisms. At the same time, an AOP mechanism also has its own expressiveness characteristics—e.g., a static AOP mechanism, such as AspectJ, is a full-fledged programming language. To remain disciplined and high-level, an AOP mechanism needs to enforce strict limitations. There will, thus, always be interesting cross-cutting concerns that cannot be nicely encapsulated by a given AOP mechanism. Meta-programming can then play the role of enriching the set of allowed behaviors

[4]For instance, aspect-oriented mechanisms have seen more success in statically typed languages, as they immediately yield some low-hanging fruit in terms of enabling patterns that the language cannot natively express (e.g., perform an action every time any method of a certain class is called). In a dynamic language, such flexibility is more likely to be natively supported, as the language has less stringent correctness checks and fewer performance requirements.

without resorting to tedious repetition, or interspersion of relevant code throughout a large program. Hopefully, the meta-programming approach will follow some of the same general lines as MAJ: It should be minimal, and it should support as much discipline as possible—e.g., offer some simple guarantees of well-formedness (like MAJ's structured generation) and high-level expressiveness where possible (like MAJ's `infer` keyword).

7. RELATED WORK

In this section we discuss work related to MAJ, to similar work in AOP, to annotation-based domain-specific languages, and to meta-programming.

In terms of philosophy, MAJ is a compatible approach to that of XAspects [Shonle et al. 2003], which advocates the use of AspectJ as a back-end language for aspect-orientation. Aspect languages in the XAspects framework can produce AspectJ code using MAJ.

Aspect-oriented languages are occasionally themselves implemented using annotations instead of as core syntax extensions. For example, a relatively new aspect-oriented programming approach is JBoss AOP [Burke et al.]. JBoss AOP allows programmers to specify the same kind of cross-cutting concerns as AspectJ. Nevertheless, instead of having to program in a different syntax, one can specify pointcuts and advice by annotating pure Java classes and methods (or, alternatively, by using XML). For the programmer, JBoss AOP has the same rigidities as AspectJ, and thus does not allow the more flexible meta-programming that we demonstrated. Internally, JBoss AOP uses JavAssist [Chiba and Nishizawa 2003] for bytecode weaving. JavAssist is a set of APIs for bytecode manipulation. It is an unstructured tool, which means that there are no guarantees on the syntactic or semantic correctness of the code that it generates.

The new EJB 3.0 standard is the prototypical example of use of Java annotations for language extension. EJB 3.0 takes away a lot of the boiler-plate code that programmers must write to make a regular Java class conform to the EJB standard. Our earlier GOTECH work [Tilevich et al. 2003] (in collaboration with Marc Fleury of the JBoss Group) pioneered the automation of generating EJBs. EJB 3.0 does not quite reach the same level of automation as GOTECH but its implementations (e.g., JBoss EJB) often emphasize other features, such as code generation directly in bytecode form.

It is interesting to compare MAJ to state-of-the-art work in meta-programming languages. Visser [Visser 2002] has made similar observations to ours with respect to concrete syntax (i.e., quote/unquote operators) and its introduction to meta-languages. His approach tries to be language-independent and relies on generalized-LR parsing (GLR). GLR is powerful for ambiguous grammars as it returns all possible parse trees. Our type system can then be used to disambiguate in a separate phase. In fact, since the original MAJ publication [Zook et al. 2004] Bravenboer et al. reimplemented parts of MAJ using this exact GLR-based approach [Bravenboer et al. 2005]. However, there are some significant differences between Bravenboer's implementation of MAJ and ours. For example, the Bravenboer implementation does not have the ability to "infer" the syntactic types of quoted code fragments. Programmers must explicitly specify the AST type of quoted fragments. Although

parsing technology is an interesting topic, we do not consider it crucial for MAJ. Our current approach with ANTLR is perhaps crude but yields a clean specification and quite acceptable performance.

Multi-stage programming languages, like MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003] represent state-of-the-art meta-programming systems with excellent safety properties. For instance, a common guarantee of multi-stage languages is that the type safety of the generator implies the type safety of the generated program. This is a much stronger guarantee than that offered by MAJ and other abstract-syntax-tree-based tools. It means that the meta-programming infrastructure needs to keep track of the contexts (declared variables and types) of the *generated program*, even though this program has not yet been generated. Therefore the system is more restrictive in how the static structure of the generator reflects the structure of the generated program. By implication, some useful and legal generators may be harder to express in a language like MetaOCaml. Current applications of multi-stage languages have been in the area of directing the specialization of existing programs for *performance* optimization. Multi-stage languages, however, are not particularly suited for code generation for *modularity* and *reusability*. Modularity and reusability of code has been our primary focus for developing MAJ. For example, in our pooling generator example, it is necessary to generate the names of methods and superclasses based on the inputs to the generator. This is not allowed using MetaOCaml—the type system of MetaOCaml cannot make the type guarantees that it currently makes and still allow generation of names without hygienic renaming.

A promising tool that does allow generation of names, yet still guarantees the type-correctness of generated code statically (at the compile time of the *generator*), is SafeGen [Huang et al. 2005]. SafeGen is a meta-programming tool for generating Java programs. A generator written in SafeGen takes legal Java programs as inputs, and generates new legal Java programs. SafeGen stipulates that the conditions directing control-flow, iteration, and name generation must be specified using first-order logic sentences, expressed over built-in predicates and functions that correspond to information available through Java reflection. For example, a generator in SafeGen may say, "for all methods in the input class, generate a method with the same name, return type, parameters, and exceptions". Note that this is exactly the sort of generation needed in the example applications we showed in section 3. The type checker for SafeGen constructs first order logic sentences whose validity indicates whether or not a particular type-correctness property holds in the *generated code*. The checking of validity is delegated to a theorem prover. Therefore, SafeGen is a viable alternative for many of the generation needs that MAJ fulfills. What SafeGen cannot do, however, is generate based on conditionals that are arbitrary functions. This is a necessity of its type system and a cost for having static type guarantees for the generated code. MAJ, on the other hand, has no such restriction, but does not provide the same level of safety guarantees. The powerful guarantees come at a cost: validity of first-order logic sentences is undecidable. SafeGen's type checker relies on the theorem prover, which might not terminate on certain queries. SafeGen places a time limit on the theorem prover, and maintains soundness by providing warnings on the time-terminated queries.

An interesting direction in meta-programming tools is that pioneered by *hygienic macro expansion* [Kohlbecker et al. 1986; Clinger 1991]. Hygienic macros avoid the problem of unintended capture due to the scoping rules of a programming language. For instance, a macro can introduce code that inadvertently refers to a variable in the generation site instead of the variable the macro programmer intended. Similarly, a macro can introduce a declaration (binding instance) that accidentally binds identifier references from the user program. The same problem exists in programmatic meta-programming systems, like MAJ. In past work, we described *generation scoping* [Smaragdakis and Batory 1999]: a facility for controlling scoping environments of the generated program, during the generation process. Generation scoping is the analogue of hygienic macros in the programmatic (not pattern-based, like macros) meta-programming world. Generation scoping does not offer any guarantees about the correctness of the target program, but gives the user much better control over the lexical environments of the generated program so that inadvertent capture can be very easily avoided. Adding a generation scoping facility to MAJ is an interesting future work direction.

Other interesting recent tools for meta-programming include Template Haskell [Sheard and Jones 2002]—a mechanism for performing compile-time computations and syntax transformation in the Haskell language. Closer to MAJ are tools, like JSE [Bachrach and Playford 2001], ELIDE [Bryant et al. 2002] and Maya [Baker and Hsieh 2002] that have been proposed for manipulating Java syntax. Most of these Java tools aim at syntactic extensions to the language and are closely modeled after macro systems. The MAJ approach is different both in that it targets AspectJ, and in that it serves a different purpose: it is a tool for programmatic meta-programming, as opposed to pattern-based meta-programming. In MAJ, we chose to separate the problem of recognizing syntax (i.e., pattern matching and syntactic extension) from the problem of generating syntax (i.e., quoting/unquoting). MAJ only addresses the issues of generating AspectJ programs using simple mechanisms and a convenient language design. Although meta-programming is a natural way to implement language extensions, uses of MAJ do not have to be tied to syntactic extensions at all. MAJ can, for instance, be used as part of a tool that performs transformations on arbitrary programs based on GUI input or program analysis. MAJ also has many individual technical differences from tools like JSE, ELIDE, and Maya (e.g., JSE is partly an unstructured meta-programming tool, Maya does not have an explicit quote operator, etc.).

Complementing MAJ with a facility for *recognizing* syntax (i.e., performing pattern matching on abstract syntax trees or even based on semantic properties [Brichau et al. 2002]) is a straightforward direction for future work. Nevertheless, note that the need for pattern matching is not as intensive for MAJ as it is for other meta-programming tools. The reason is that MAJ generates AspectJ code that is responsible for deciding what transformations need to be applied and where. The beauty of the combination of generative and aspect-oriented programming is exactly in the simplicity of the generative part afforded by the use of aspect-oriented techniques. Another promising direction for future work on MAJ is to make it an extension of AspectJ, as opposed to Java (i.e., to allow aspects to generate other aspects). We do not yet have a strong motivating example for this application, but

we expect it may have value in the future.

8. CONCLUSIONS

In this article we presented an approach of automating software engineering tasks by implementing small, unobtrusive language extensions for Java. We introduced Meta-AspectJ (MAJ): a tool for generating Java or AspectJ programs. The implementation of MAJ was largely motivated by practical concerns: although a lot of research has been performed on meta-programming tools, we found no mature tool, readily available for practical meta-programming tasks in either Java or AspectJ. MAJ strives for convenience in meta-programming but does not aspire to be a heavyweight meta-programming infrastructure that supports syntactic extension, pattern matching, etc. Instead, MAJ is based on the philosophy that generative tools have a lot to gain by generating AspectJ code and delegating many issues of semantic matching to AspectJ. Of course, this approach limits the ability for program transformation to the manipulations that AspectJ supports. Nevertheless, this is exactly why our approach is an aspect-oriented/generative hybrid. We believe that AspectJ is expressive enough to capture many useful program manipulations at exactly the right level of abstraction. When this power needs to be combined with more configurability, generative programming can add the missing elements. We demonstrated that, using MAJ, we can implement a number of annotation-based domain-specific languages that automate tedious and error-prone programming tasks.

Acknowledgments and Availability

This research was supported by the National Science Foundation under Grant No. CCR-0238289, as well as by a grant from LogicBlox Inc.

MAJ is available at `http://www.cc.gatech.edu/maj`.

REFERENCES

Bachrach, J. and Playford, K. 2001. The Java syntactic extender (JSE). In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 31–42.

Baker, J. and Hsieh, W. C. 2002. Maya: multiple-dispatch syntax extension in Java. In *Programming Language Design and Implementation (PLDI)*. ACM Press, 270–281.

Batory, D., Lofaso, B., and Smaragdakis, Y. 1998. JTS: tools for implementing domain-specific languages. In *International Conference on Software Reuse (ICSR)*. IEEE, Victoria, BC, Canada, 143–153.

Batory, D., Sarvela, J. N., and Rauschmayer, A. 2003. Scaling step-wise refinement. In *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 187–197.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *European Conference on Object-Oriented Programming and Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP)*. ACM Press, New York, NY, USA, 303–311.

Bravenboer, M., Vermaas, R., Vinju, J., and Visser, E. 2005. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Generative Programming and Component Engineering (GPCE) Conference*, R. Glück and M. Lowry, Eds. Lecture Notes in Computer Science, vol. 3676. Springer, Tallin, Estonia, 157–172.

Brichau, J., Mens, K., and Volder, K. D. 2002. Building composable aspect-specific languages. In *Generative Programming and Component Engineering (GPCE) Conference*. LNCS 2487. Springer.

Bryant, A., Catton, A., De Volder, K., and Murphy, G. C. 2002. Explicit programming. In *Aspect-Oriented Software Development (AOSD)*. ACM Press, 10–18.

Burke, B. et al. *JBoss AOP Web site, http://labs.jboss.com/portal/jbossaop*. Accessed Sep. 2007.

Calcagno, C., Taha, W., Huang, L., and Leroy, X. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE) Conference*. LNCS 2830. Springer, 57–76.

Chiba, S. 1995. A metaobject protocol for C++. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. SIGPLAN Notices 30(10). Austin, Texas, USA, 285–299.

Chiba, S. and Nishizawa, M. 2003. An easy-to-use toolkit for efficient java bytecode translators. In *Generative Programming and Component Engineering (GPCE) Conference*. Springer-Verlag, New York, NY, USA, 364–376.

Clinger, W. 1991. Macros that work. In *Principles of Programming Languages (POPL)*. ACM Press, 155–162.

Cordy, J. R., Halpern-Hamu, C. D., and Promislow, E. 1991. Txl: a rapid prototyping system for programming language dialects. *Computer Languages 16,* 1, 97–107.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. 2006. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst. 28,* 2, 331–388.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java Language Specification, Third Edition*. Addison-Wesley, Boston, Mass.

Guy L. Steele, J. 1990. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA.

Huang, S. S. and Smaragdakis, Y. 2006. Easy language extension with Meta-AspectJ. In *International Conference on Software Engineering (ICSE), Emerging Results*. 865–868.

Huang, S. S., Zook, D., and Smaragdakis, Y. 2005. Statically safe program generation with SafeGen. In *Generative Programming and Component Engineering (GPCE) Conference*. 309–326.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. Getting started with AspectJ. *Communications of the ACM 44,* 10, 59–65.

Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. 1986. Hygienic macro expansion. In *1986 ACM conference on LISP and functional programming*. ACM Press, 151–161.

Lohmann, D., Blaschke, G., and Spinczyk, O. 2004. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Generative Programming and Component Engineering (GPCE) Conference*, G. Karsai and E. Visser, Eds. LNCS, vol. 3286. Springer Verlag, 55–74.

Mohnen, M. 2002. Interfaces with default implementations in Java. In *Principles and Practice of Programming*. National University of Ireland, Maynooth, County Kildare, Ireland, Ireland, 35–40.

Parr, T. J. and Quong, R. W. 1995. ANTLR: A predicated LL(k) parser generator. *Software—Practice and Experience 25,* 7 (July), 789–810.

Rajan, H. and Sullivan, K. 2003. Eos: instance-level aspects for integrated system design. *SIGSOFT Softw. Eng. Notes 28,* 5, 297–306.

Sheard, T. and Jones, S. P. 2002. Template meta-programming for Haskell. In *ACM SIGPLAN workshop on Haskell*. ACM Press, 1–16.

Shonle, M., Lieberherr, K., and Shah, A. 2003. Xaspects: An extensible system for domain specific aspect languages. In *OOPSLA'2003, Domain-Driven Development Track*.

Smaragdakis, Y. 2004. A personal outlook on generator research. In *Domain-Specific Program Generation*, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer-Verlag. LNCS 3016.

Smaragdakis, Y. and Batory, D. 1999. Scoping constructs for program generators. In *Generative and Component-Based Software Engineering Symposium (GCSE)*. Earlier version in Technical Report UTCS-TR-96-37.

Soares, S., Laureano, E., and Borba, P. 2002. Implementing distribution and persistence aspects with AspectJ. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 174–190.

Stevens, A. et al. *XDoclet Web site, http://xdoclet.sourceforge.net/*. Accessed Sep. 2007.

Taha, W. and Sheard, T. 1997. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*. New York: ACM, 203–217.

Tilevich, E., Urbanski, S., Smaragdakis, Y., and Fleury, M. 2003. Aspectizing server-side distribution. In *Automated Software Engineering (ASE) Conference*. IEEE Press.

Visser, E. 2002. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE) Conference*. LNCS 2487. Springer, 299–315.

Volder, K. D. 1998. Type-oriented logic meta programming. Ph.D. thesis, Vrije Universiteit Brussel.

Weise, D. and Crew, R. F. 1993. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*. 156–165.

Zook, D., Huang, S. S., and Smaragdakis, Y. 2004. Generating AspectJ programs with meta-AspectJ. In *Generative Programming and Component Engineering (GPCE) Conference*. Springer-Verlag, 1–18.