XAspects: An Extensible System for Domain-Specific Aspect Languages[*]

Macneil Shonle[†]
Department of Computer
Science and Engineering
UC San Diego
La Jolla, CA 92093-0114
mshonle@cs.ucsd.edu

Karl Lieberherr
College of Computer and
Information Science
Northeastern University
Boston, MA 02115
lieber@ccs.neu.edu

Ankit Shah
College of Computer and
Information Science
Northeastern University
Boston, MA 02115
ankit@ccs.neu.edu

ABSTRACT

Current general aspect-oriented programming solutions fall short of helping the problem of separation of concerns for several concern domains. Because of this limitation good solutions for these concern domains do not get used and the opportunity to benefit from separation of these concerns is missed. By using XAspects, a plug-in mechanism for domain-specific aspect languages, separation of concerns can be achieved at a level beyond what is possible for object-oriented programming languages. As a result, XAspects allows for certain domain-specific solutions to be used as easily as a new language feature.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.11 [**Software Engineering**]: Software Architectures—*languages, domain-specific architectures*

General Terms

Languages, Design.

Keywords

Aspect-oriented programming, generative programming, language extensions, domain-specific languages.

1. INTRODUCTION

The developing field of aspect-oriented programming explores the problem of separation of concerns. While object-oriented languages provide constructs to allow many concerns to be decomposed into separate classes and methods, there remain concerns that cannot be cleanly decomposed using constructs found in object-oriented languages. When a concern spans several components that concern is said to *crosscut* the components. For example, in order for a Java program to support logging of method calls, the profiling instrumentation code must be inserted before and after every method definition, either manually or by a tool. The profiling instrumentation, therefore, crosscuts the components that are profiled.

The goal of aspect-oriented programming is to identify crosscutting concerns and modularize them into *aspects*. A compiler for an aspect-oriented language takes these aspects together with the components and weaves them into a whole program. The most popular aspect-oriented programming language to date is AspectJ [32], which is a general purpose aspect language built on top of Java. In AspectJ, logging and other concerns can be written as separate aspects that are cleanly modularized, without the need for tools external to the compiler.

While general purpose aspect languages are useful, certain crosscutting concerns are best handled when using domain-specific aspect languages. A *domain-specific aspect language* is a custom language that allows special forms of crosscutting concerns to be decomposed into modularized constructs. Examples of domain-specific aspect languages include languages for dealing with coordination concerns, object marshaling concerns, and class graph traversal concerns.

Similarly, while some non-crosscutting concerns can be expressed as components in an object-oriented language these concerns might be better expressed in a domain-specific language: A domain-specific language sacrifices generality for closeness to a fixed problem area. In this paper, we will refer to these non-crosscutting languages as *domain-specific component languages*. One example of a domain-specific component language is a language for a parser generator tool.

In this paper, we present a system that integrates domain-specific aspect and component languages with AspectJ using a technique that allows domain-specific language implementations to leverage a powerful subset of AspectJ. The domain-specific languages are integrated by using a plug-in architecture that allows the constructs created by the languages to cooperate with each other while not being aware that the other plug-ins exist.

1.1 Background: The AspectJ Language

The AspectJ language supports the decomposition of crosscutting concerns with two concepts: (1) the join point model; and (2) inter-type declarations. These concepts are discussed below. A full definition of the AspectJ Programming Language can be found on the AspectJ website [32].

1.1.1 AspectJ's Join Point Model

AspectJ's join point model is concerned with dynamic events that occur during the execution of a program. Events recognized by AspectJ include: a call to a method or constructor, an execution of a method or constructor, a write to or read from a field, an execution of an exception handler. These events are called *join points* in AspectJ parlance.

Pointcuts. A *pointcut* represents a set of join points. Complex pointcuts can be built by using the set theory operators union, intersection, and complement on other pointcuts. More advanced operations can be applied to pointcuts to pick out those events that occur within another pointcut. For example, a pointcut could identify all writes made to any integer field named `numItems` during the execution of any method named `count`. Below, that pointcut is given the name `setOccuredInCount` and is defined from the two pointcuts `setOccured` and `inCount`:

```
pointcut setOccured(int rhs):
          set(int *.numItems) && args(rhs);
pointcut inCount():
          cflow(execution(* *.count(..)));
pointcut setOccuredInCount(int rhs):
          setOccured(rhs) && inCount();
```

Pointcut `setOccuredInCount` (through pointcut `inCount`) selects all join points that are executed directly or indirectly in the control-flow of the `count` method. The "`..`" means that the `count` method can have any number of arguments. AspectJ also includes the `withincode` operator to identify only the code in the count method *body* itself.

Advice. After the pointcuts have been specified, new behavior can be associated with the events by using AspectJ's *advice*. The join points identified by a pointcut can be advised in three places: before the execution of the join point; after the execution of the join point, or "around" the execution of the join point. Figure 1 shows an aspect that includes a pointcut definition and an after advice applied to it.

An around advice can be used to perform actions before, after, or instead of executing the join point. Because each

```
01 aspect SetWatcher {
02   pointcut setOccuredInCount(int rhs):
03           set(int *.numItems) && args(rhs)
04           && cflow(execution(* *.count(..)));
05
06   after(int newVal): setOccuredInCount(newVal){
07     System.out.println("New value: " + newVal);
08   }
09 }
```

Figure 1: A Simple AspectJ Aspect.

advice is lexically scoped an advice cannot access the local variables of the join point nor any other advice that may be associated with the join point: for example, an after advice cannot use the variables declared in a before advice, therefore an around advice is useful when a resource needs to be acquired before a join point occurs and then released afterwards. An around advice is needed when a join point execution needs to be conditional.

1.1.2 Inter-type Declarations

Along with modifying the behavior of the program dynamically, AspectJ can be used to modify the static structure of a program. Static modifications are performed through AspectJ's *inter-type declarations*. Methods, constructors or fields can be introduced to preexisting classes from aspect definitions.

Inter-type declarations can be used to express the static concerns of the program in a manner not allowed in Java. For example, an aspect can be created to define all of the image rendering functions of a family of classes in a single aspect instead of scattering these similar functions across multiple classes. Yet another aspect could be created to define all of the caching fields and functions of the family that is separate from the rendering aspect. Inter-type declarations and AspectJ's join point model can also be used to modify classes that exist only in Java bytecode form.

1.2 Where AspectJ Falls Short

Unfortunately the constructs of AspectJ are not sufficient enough to separate all concerns. The visitor pattern [9] is one such concern: Given a root object that contains (directly or indirectly) sub-objects of a particular class provide a mechanism for a "visitor object" to be supplied with a reference to each sub-object. The visitor pattern can be implemented using AspectJ (as shown in [11]) but any efficient implementation of it must depend on a solution where the program states the object structure twice: once for the class definitions themselves and then again to specify the relationship of the root object to all of its sub-objects. This structural redundancy is a weakness because whenever the relationship between the root class and the composed classes change the visitor methods also need to be changed. For example, if a new field is added such that it will need its own visit method defined then that visit method must be written and one of the existing visit methods must be modified to accommodate for the addition. While AspectJ allows the visitor pattern code to be encapsulated in an aspect, the structural properties still crosscut both the classes and the aspect. The end result is a system where concerns are encapsulated but still highly coupled[1].

DemeterJ & DAJ. The solution to the structure-awareness problem is solved by a domain-specific aspect language called DemeterJ [35]. DemeterJ supports the visitor pattern in a structure-shy way: Instead of writing the visit methods manually a compact path traversal language is used. The traversal language allows the program to describe what needs to be done, not how it should be done. For example, suppose we have a University object that contains Library objects, and Library objects that contain Book objects. To

---

[1]Reflection can be used to dynamically compute the structural properties of the program, but at the cost of runtime efficiency.

specify the traversal path starting at a University instance down to each Book instance the following could be written (the notation used here is from the DAJ language [34], the successor of DemeterJ):

```
declare strategy: everyBook:
                    "from University to Book";
```

How a University object contains Book objects is irrelevant to the traversal strategy. If the University class is changed by first decomposing it into Colleges and decomposing Libraries into a collection of Shelves the above strategy will still be valid. More complex traversal paths can be made by specifying constraints and computing the intersection of two paths:

```
declare strategy: everyBook:
  "intersect(from University to Book, skipDorms)";
declare strategy: skipDorms:
              "from * bypassing Dorm to *";
```

A traversal path can be forced to go through a particular type or field name by using the `via` operator and a traversal path can be forced to bypass a particular type or field name by using the `bypassing` operator. These path requirements can be composed and intersected with further path requirements.

A traversal is used in order to create the methods required to implement the visitor:

```
declare traversal: void visitBooks():
                    everyBook (MyBookVisitor);
```

The only visible change to the structure of the program after the above traversal is used is that the University class now has a `visitBooks` method: the `visitBooks` method is placed in the University class because the path specified by the `everyBook` strategy starts at it (as specified by the `from` keyword). Behind the scenes the DAJ compiler implements the `visitBooks` method by adding support methods in the correct locations as specified by the strategy.

The lack of constructs that support the visitor pattern is just one example of where the AspectJ language falls short of separating concerns. We conjecture that this gap between domain-specific aspect languages and general purpose aspect languages will always exist because any general solution cannot cover all possible ways an aspect can crosscut a system. By their nature concerns are related to the problem domain, and there is an infinite variety of forms that problem domains can take.

AspectJ has some support for particular domains via abstract aspect libraries: The pointcut sets described by an abstract aspect can be defined later by a concrete aspect. Only a limited set of problems has thus far created useful aspect libraries.

1.3 Domain-Specific Solutions

In order to address the problem of separation of concerns cleanly domain-specific solutions can be used. Domain-specific solutions (such as DemeterJ) allow decomposition on a level unachievable with object-oriented languages. Domain-specific aspect languages in particular can be used for concerns whose implementation in an object-oriented language is either scattered or the implementation duplicates informa-

tion of other concerns. The DemeterJ and DAJ languages are useful for structure-shy concerns[2].

By using domain-specific solutions productivity is increased, the problem solved can be more complex, and the resulting quality is improved.

1.4 Problems Using Multiple Domain-Specific Solutions

Most non-trivial problem domains have concerns that a single domain-specific solution cannot entirely express. In order to use multiple domain-specific solutions, the program must be fragmented into different pieces that are read by each tool. For example, if one of the concerns is parsing then a parser-generator tool can be used along with the plain Java source files.

The fragmentation of tools causes problems when each concern relies on the solutions provided by the other tools: While a parser can remain orthogonal to the rest of the sources, a traversal through the program cannot be orthogonal to the sources. In the next section we present how XAspects [1] addresses this problem.

2. A PLUG-IN ARCHITECTURE

The fragmentation problem can be solved by forcing each domain-specific solution to conform to a particular framework and by defining a common language with which the domain-specific tools can communicate. The XAspects tool provides a plug-in framework that integrates domain-specific languages such that the tools become extensions of the AspectJ language. Rules restricting what a plug-in can do are introduced in order to enhance the integration.

A simple plug-in will first be introduced before the discussion of the general architecture.

2.1 Example: Class Dictionary Plug-In

The ClassDictionary plug-in can be used in order to solve parsing and printing concerns for a family of related classes. A class dictionary is a grammatical representation of a set of classes that are composed from each other. Just as in BNF grammars, class dictionary productions have a left hand side and a right hand side: The left hand side is an identifier that defines a new class and is the name of the new grammatical rule. The right hand side is a sequence of terminals and non-terminals. The non-terminals are members of the class identified by the left hand side and the terminals are tokens used to make the string representations of the objects parsable and printable.

A sample class dictionary is pictured in Figure 2. The first class defined by the ClassDictionary aspect in Figure 2 is the CD class, which has three fields: (1) the CD's `album` title field, represented by a String; (2) the CD's `artist` field, represented by an Artist; and (3) the CD's `upc` field, represented by a UPCNumber. If a non-terminal is found in a production without a `<fieldname>` presented in angle brackets then the field name defaults to the name of the type with an initial lower-case letter. The Artist class presented in Figure 2 is an abstract class with

---

[2]In general, a concern C is X-shy if there is a C implementation that depends on minimal knowledge of X and if the C implementation can adapt to any changes in X that do not affect dependency semantics. Domain-specific aspect languages allow problems to be decomposed to support concern-shy representations of concerns.

```
01 aspect(ClassDictionary) CompactDisc {
02   CD = "cd" <album> String "by" Artist ","
03        "(" <upc> UPCNumber ")" ".";
04   Artist : SingleArtist | MultipleArtist;
05   SingleArtist = Name;
06   MultipleArtist = Name "," Names;
07   Name = String;
08   UPCNumber = String;
09   CommaList(S) ~ S { "," S };
10   Names = CommaList(Name);
11 }
```

Figure 2: A Class Dictionary Aspect, using EBNF Notation.

two concrete sub-classes: SingleArtist and MultipleArtist. The CommaList class is a grammatical rule that is parameterized by other grammatical rules[3]. The "{}" brackets used is the EBNF representation for zero or more repetitions of the enclosed terminals and non-terminals. A CD object might be printed with the following representation:

```
cd "The Steve Howe Album" by "S. Howe",
  ("07567-81559-25").
```

The generated CD class can parse the above as input through its static `parse` method.

Class dictionaries do not encapsulate crosscutting concerns because class dictionaries only generate new classes; they do not modify existing classes. For this reason the class dictionary language is a domain-specific component language. Class dictionaries are useful for separating structural concerns. For example, suppose the class structure was changed so that the Artist rule was a concrete class representing lists of Names:

```
        Artist = CommaList(Name);
```

Even though the class representation has changed, the textual representation of a CD is the same. Class dictionaries together with traversals give strong support for structure-shy programming. For example, had a traversal been performed to visit all Name instances then the code would be correct regardless of how Artist is structured.

The class dictionary in Figure 2 generates seven classes: Had this solution been written in Java seven separate source files would have to be used, plus the additional code for parsing CDs.

2.2   Compilation Phases

The XAspects compiler (xajc) and each plug-in tool communicate with each other during six phases:

Source Code Identification The XAspects compiler identifies in the source code all program text that belongs to the plug-in and provides that text to the plug-in.

Generation of External Interfaces The plug-in generates source files that define the external (i.e. program visible) components introduced by the language that the plug-in implements.

---

[3]CommaList(Name) is the syntax for instantiating the parameterized CommaList rule with Name.
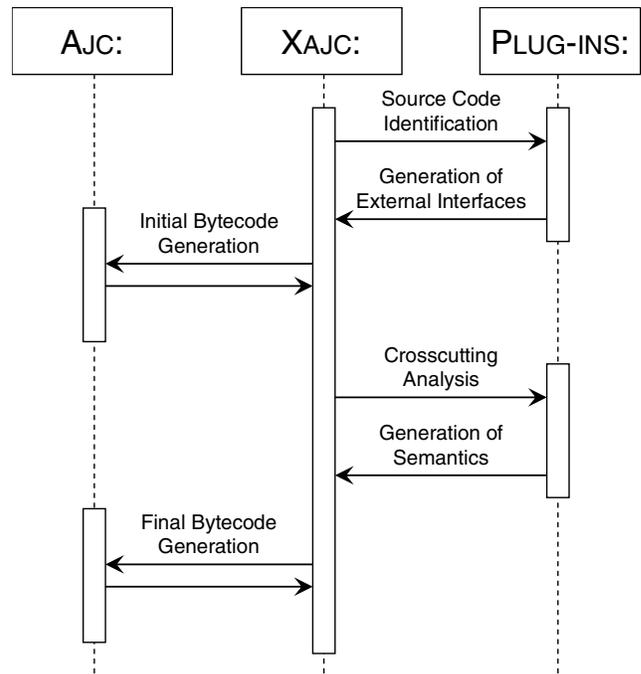

Figure 3: The Compilation Phases.

Initial Bytecode Generation The AspectJ compiler (ajc) generates bytecodes from the plug-in defined external interfaces and the remainder of the program.

Crosscutting Analysis The XAspects compiler provides each plug-in the binary of the program to perform reflection or other analysis on it.

Generation of Semantics The plug-in generates behavioral changes to the generated program in the form of AspectJ source code. (Structural changes are prohibited during this phase, only methods can be modified, and only aspect `private` methods can be introduced.)

Final Bytecode Generation Finally, the new behavioral code is woven into the existing code to create the complete program.

An overview of the compilation phases is depicted in Figure 3. Each of these steps are discussed in the following sections.

2.2.1   Source Code Identification

The input files to the XAspects compiler are Java and AspectJ source files that are allowed to include one minor syntactic extension: When the `aspect` keyword is followed by parentheses the parentheses must contain a valid type name for a class that can be found on the current class-path (the class is the plug-in itself). Line one of Figure 2 is an example of this form. All of the tokens contained in the {} brackets are scanned without interpretation by XAspects. The tokens have the following restrictions:

- The tokens must be valid Java tokens; and

- Any "{" bracket must be balanced by a "}" bracket, without any dangling brackets.

Non-Java tokens, such as "@" and "#," are not allowed in the aspect body. However, the plug-in can choose to express these characters by having the user contain them in string literals.

The tokens found in the brackets are the plug-in's *body*. For example, in Figure 2 the body of the ClassDictionary plug-in aspect is lines 2–10. After the source code specific to each plug-in is identified by the XAspects compiler, each plug-in used is instantiated and provided with the text of all bodies that belong to the plug-in. At this point the plug-in can choose to parse the body or delay parsing until the next phase.

2.2.2 Generation of External Interfaces

The XAspects compiler then queries each plug-in to provide a set of source files that describe the program visible interfaces generated by the plug-ins[4]. For example, the ClassDictionary plug-in at this point will provide the list of classes, methods and fields that were described by the grammar.

The external interfaces are described through AspectJ source code. As such, the external interfaces can express new classes or modifications to existing classes via inter-type declarations. The AspectJ source code generated needs to meet the following restrictions:

- All new classes generated must have a name that appeared as an identifier token in the aspect body. This restriction allows the user of the plug-in to have full control over the names generated. If this restriction were not in place different plug-ins could try to generate the same class names, with no mechanism available for the user to resolve the conflict.

- All new public fields or new public methods introduced to existing classes (via inter-type declarations) must have a name that appeared as an identifier token in the aspect body.

- All new classes generated from each aspect body must be generated in the same Java `package` in which the aspect body was defined.

The fields or methods defined in new classes do not have to be named by identifiers present in the aspect body because any further changes to that class will be done through external inter-type declarations or plug-ins. Therefore, any name clashes that arise can be handled in those changes.

2.2.3 Initial Bytecode Generation

Any of the public methods generated by the plug-ins in the previous phase may be stubs at this point (i.e. non-void functions that always return `null` or void functions that do nothing). These external interfaces are then woven with the rest of the AspectJ and Java source by using the AspectJ compiler.

2.2.4 Crosscutting Analysis

After the initial bytecode generation phase the structure of the program is complete and plug-ins cannot add new fields to any of the classes. Because the structure of the program is fixed the complete bytecodes for the program can be provided to each plug-in. Because the code generated by each plug-in can be translated into Java bytecodes (through AspectJ) the changes made by all plug-ins are in the same, well-defined language.

During this phase domain-specific aspect languages can perform structural analysis on the program through Java reflection on those bytecodes. For example, the Traversal plug-in (which supports DemeterJ style traversals) can generate concrete paths based on the traversal rules specified.

An alternative and more efficient way to perform the crosscutting analysis is to operate on the abstract syntax tree of the AspectJ program produced in the initial bytecode generation phase. The modified abstract syntax tree would then be used to generate code. This would avoid having to use the AspectJ compiler twice but this approach would need an AspectJ compiler that has an API to import and export abstract syntax trees. The current implementation of XAspects invokes the AspectJ compiler twice: The first compilation produces an executable that can be used (through reflection or bytecode parsing) to produce the class graph for the crosscutting analysis. The second compilation produces the final program.

2.2.5 Generation of Semantics

Each plug-in can provide semantic changes by using AspectJ's dynamic join-point model. The implementation of functions that were stubbed can be specified by using AspectJ's "around" advice. New private methods can also be introduced at this point by using inter-type declarations. Note that a private inter-type declaration does not introduce a private method to the class, rather it introduces a method to the class that is only visible by the generating aspect. For example, the Traversal plug-in can generate private methods to each of the classes it needs to traverse: these private methods cannot be called by other parts of the program, not even the classes of which the methods are members.

The AspectJ code generated during this phase must satisfy the restriction that only one AspectJ aspect can be generated at this phase for each domain-specific aspect defined. It must have the name of the identifier given to the domain-specific aspect (for example, the name of the domain-specific aspect in Figure 2 is "CompactDisc").

The AspectJ aspect generated should be careful with any use of the `declare precedence` rule. The `declare precedence` rule is a feature of AspectJ to manage conflicts between aspects that advise the same methods. If `declare precedence` might be necessary for the aspect, the user should be able to specify the clause, because only the user has enough information to decide how to resolve conflicts.

2.2.6 Final Bytecode Generation

Finally, the AspectJ source provided by the plug-ins are gathered by the XAspects compiler and the complete binary is created. The 1.1 release of the AspectJ compiler includes support for incremental compilation that may allow a future implementation of XAspects to compile faster because unchanged files will not have to be parsed twice.

The actions taken during these phases are discussed with concrete examples in the next section.

---

[4]Interface here means the API of the classes generated by the plug-in, which is not to be confused with the Java `interface` construct.

```
01 class MusicRegistry {
02   public void register(UPCNumber upc, CD cd) {
03     /* insert cd into the database */
04   }
05
06   public CD getCD(UPCNumber upc) {
07     /* find the CD with the given upc */
08   }
09 }
10
11 aspect(ClassDictionary) CompactDisc {
12  CD = "cd" <album> String "by" Names
13    "," "(" <upc> UPCNumber ")" ".";
14  Name = <name> String;
15  Names ~ Name { "," Name };
16  UPCNumber = <upc> String;
17 }
18
19 aspect(Coordination) MusicRegistryThreading
20    changes(MusicRegistry) {
21  declare selfex: register;
22  declare mutex: {register, getCD};
23 }
24
25 aspect(Traversal) MusicCollections {
26  declare strategy: everyMusicalAlbum:
27    "intersect(from AudioLib to CD, justMusic)";
28  /* skip books on CD */
29  declare strategy: justMusic:
30    "from * via MusicCollection to *";
31  declare traversal: void listAlbums():
32    everyMusicalAlbum (PrintCD);
33 }
34
35 aspect(TraversalAdvice) PrintCD {
36  void before(CD cd) {
37    System.out.println(cd.toString());
38  }
39 }
```

Figure 4: An XAspects Source File using Multiple Plug-Ins.

3. EXAMPLE: AN AUDIO LIBRARY

The following example demonstrates a program that uses three different domain-specific languages: (1) a class dictionary language; (2) a thread coordination language; and (3) a traversal language. The thread synchronization aspect language was taken from the COOL programming language, a part of the D framework [18]. The class dictionary and traversal languages were taken from the DemeterJ system. Figure 4 shows how these plug-ins can be used to implement an audio library. The code generated by each language is discussed in the following sections.

3.1 Class Dictionary Language

The ClassDictionary language is used for expressing structure shy input and output languages. The aspect body given to the ClassDictionary plug-in is lines 12–16 of Figure 4. After given the body, the ClassDictionary generates the following classes for the generation of external interface phase: CD, Name, Names, and UPCNumber. The CD class con-tains the fields `album`, `names`, and `upc`, along with a default constructor and a constructor parameterized by the CD's fields. The Name and UPCNumber classes contain the single String fields `name` and `upc`, respectively, along with both default and parameterized constructors. The Names class is a collection of one or more Name objects; it is a Java container typed in a way that a traversal can recognize that Name elements exist in it. (If a plain Java container was used a traversal could not know that the Object contained in it was of the desired type.)

Because a ClassDictionary is a domain-specific component language it does not need to perform any crosscutting analysis.

3.2 A Coordination Language

The Coordination language is used for expressing thread coordination concerns. In multithreaded programming an important concern is the synchronization policy: for a given object, or set of objects, locks must be used when various operations are being performed.

The two features of the Coordination plug-in we will examine are the `selfex` and `mutex` declarations. The `selfex` declaration on line 21 of Figure 4 states that the `register` method of MusicRegistry is non-reentrant: At most one call to `register` can occur across all threads at any time.

The "changes" declaration on line 20 is one use of a generalization of Java's "extends" and "implements" keywords. This generalization in XAspects recognizes that the relationship among classes go further than the "is-a" and "has-a" relationships. In this case, the "changes" declaration is being parameterized with the name of the MusicRegistry class. This declaration is passed to the plug-in along with the aspect body during the source code identification phase. The parenthesis are used to differentiate the new declaration words, which can be any valid identifier, from their arguments.

The `mutex` declaration on line 22 of Figure 4 states that calls to `getCD` and `register` cannot occur simultaneously. That is, that the methods are mutually exclusive.

The implementation of the Coordination aspect relies on AspectJ's inter-type declarations to introduce new locks to the object and on the AspectJ's join point model to change the semantics of existing methods. The AspectJ code in Figure 5 was generated by the Coordination aspect to implement the `selfex` behavior. By separating the locking policy and encapsulating it in an aspect both the component code and locking policy code are easier to understand and the programmer can reason about synchronization using language that is closer to the problem domain.

3.3 A Traversal Language

The Traversal Language uses two plug-ins: Traversal and TraversalAdvice. The `traversal` declaration on line 31 introduces a new `listAlbums` method to the AudioLib class. The `listAlbums` method calls new, private traversal methods to locate each CD instance accessible through the MusicCollection class. The visitor itself, PrintCD, is created through the TraversalAdvice plug-in, which is merely a wrapper for a Java class. The Traversal plug-in assumes that the structure of the classes it needs to traverse are tightly typed (for example, the traversal will not visit generic Objects). If the structure information is not available

```
01 aspect MusicRegistryThreading {
02   Cool_Lock MusicRegistry.registerlock =
03     new Cool_Lock();
04   Object MusicRegistry.cool_sync =
05     new Object();
06   pointcut in_register(MusicRegistry tar):
07     execution(* MusicRegistry.register(..)
08     && target(tar);
09   before(MusicRegistry tar): in_register(tar) {
10     tar._enter_register();
11   }
12   after(MusicRegistry tar): in_register(tar) {
13     tar._exit_register();
14   }
15   void MusicRegistry._enter_register(){
16     while (registerlock.othersRunning()) {
17       try { synchronized(cool_sync) {
18             cool_sync.wait(); } }
19       catch (InterruptedException e) {} }
20     registerlock.in();
21   }
22   /* ... */
23 }
```

Figure 5: Partial View of Self-Exclusion Code Generation

at compile-time it cannot insert the appropriate traversal methods.

3.4 Cooperation Between the Languages

An important issue in XAspects is how much information an aspect A needs to have about other aspects for A to function properly. Many aspects depend on each other by using common names and those common names might indicate constraints that must hold between aspects. Consider the Traversal aspect in Figure 4, which depends on the ClassDictionary aspect. The statement "`from AudioLib to CD`" in the Traversal aspect makes the assumption about the ClassDictionary aspect that there is a path from an AudioLib object to a CD object. If there is no such path it would be impossible to ever find a CD in the audio library. It is the responsibility of the plug-in designers to check for those dependencies.

All properties of the code that the Traversal aspect would need from any other aspects will be available in the first bytecodes generated. It is important that all visible interface changes (done using inter-type declarations) are performed by the plug-ins in the first compilation phase. An inter-type declaration might modify the class graph and the complete class graph is needed for computing the traversals in the crosscutting analysis phase. It is interesting to notice that the Traversal aspect is dependent on all aspects that create inter-type declarations for additional parameterless methods. So one aspect may be dependent on many other aspects. The addition of one inter-type declaration might break many traversals and requires the testing of all traversals whose traversal graph changes because of the new inter-type declaration. The alternative of not using the traversal aspect would even be worse because then we would have to maintain all navigation code manually rather than using an automatic tool.

We can generalize from this traversal example to other aspects: All properties of the code that an aspect would need from "the other aspects" will be available in the first bytecodes generated.

A slightly modified version of the Audio Library that the current XAspects tool can handle, along with other XAspects examples, is available through [1] (via the resources link).

4. ADVANTAGES OF XASPECTS

The XAspects system is unique because it separates conceptually two different semantic provisions of a domain-specific language: the language's external interface and structural modifications (structural meaning) and the language's generated semantics (behavioral meaning). The separation of these two concepts allows for the plug-ins to passively cooperate with each other.

The XAspects infrastructure essentially reduces the problem for a domain-specific language to the problem of reading in Java code and producing changes and additions to that code. Dependency issues among domain-specific languages are reduced because each plug-in only has to worry about the properties of Java classes, and not the properties of constructs of other domain-specific languages that created those classes.

It could be argued that it is not feasible to introduce many little domain-specific languages with which the programmers have to work. However, the inherent complexity of a project may require that the programmers address the issues of composition of aspect languages on a conceptual level. A conscious language design that addresses the aspect composition issues systematically can simplify the programmers's job: Without such an approach, the programmers will have to do this at the lower level of a general purpose aspect language, which results in more code that actually needs to be written, some of which will be redundant with the structure of the program and could have been generated automatically.

Another weakness of general aspect-oriented programming is the difficulty of using the language correctly. A novice can easily create an infinite loop in AspectJ if an advice happens to apply to itself. More subtle misapplications of advice can also occur that are not so easily observed. If a custom aspect language is created for a specific domain these misapplications can be checked by the plug-in. While a reusable AspectJ aspect might be able to implement some of the functionalities of an XAspects plug-in, it cannot perform some of the compile time checks needed to ensure that the aspect is being used correctly. Lieberherr et al. [14] discusses extensions that could be made to AspectJ's static property checker to make it more useful.

5. IMPLEMENTATION DETAILS

The XAspects compiler is a wrapper around the `ajc` compiler for AspectJ. This section describes some interesting implementation details of the system.

Self Implementation. The XAspects system is written in part using a minimal, earlier version of itself using the ClassDictionary and Traversal plug-ins. Because the ClassDictionary language is made for parsing LL grammars it is useful for implementing the syntax of domain-specific lan-

guages. After an AST is generated by the ClassDictionary provided `parse` method it can be traversed using constructs created by the Traversal language. In general, a plug-in can be implemented by writing a class dictionary for the input language and traversing it.

Compile Time Reflection Implementation. The bytecodes provided during the crosscutting analysis phase discussed in §2.2.4 largely represent the final bytecodes of the program except for the presence of private methods or changes to existing methods. Each plug-in is given these bytecodes in the form of a set of files. Each plug-in can choose to load the bytecodes and perform analysis on them using the Java reflection interfaces. The Traversal plug-in makes extensive use of this information in order to determine the class graph of the program.

More advanced analysis can be performed by the plug-in to compute properties of the program beyond its static class graph. For example, a plug-in could analyze the looping constructs in a manner similar to the Reverse Graphics system [21], a domain-specific aspect language from Xerox. In the Reverse Graphics system, a family of image processing filters are composed from each other such that each filter is easy to understand and debug yet inefficient because each filter iterates over the entire image each time, creating multiple temporary copies. This inefficiency is solved with the introduction of an aspect that allows the loops to be fused together but without sacrificing the code clarity: The task of fusing the loops can be performed by the compiler instead of the programmer having to perform it by hand.

The compile-time reflection and bytecode analysis tasks of a complex aspect language are simpler because the object and execution model of the Java language is preserved, even with the additions created by other plug-ins. This preservation is important because there are many libraries already available for computing with Java bytecodes or Java reflection.

6. RELATED WORK

XAspects builds on Crista Lopes's work on domain-specific aspect languages. She proposed COOL and RIDL [19, 17] which she developed further in her thesis [18]. COOL and RIDL were also the crystallization point for AspectJ. XAspects combines all those innovations into an integrated system using the powerful features of AspectJ to implement COOL and RIDL and many other aspect languages using a plug-in approach.

Czarnecki and Eisenecker [8] also propagate the idea of domain-specific aspect languages. In section 5.5, an aspectual DSL is defined as a language that influences the semantics of other languages. The section goes on "Implementing modularly composable DSLs requires a common language implementation platform providing all the infrastructure for implementing language plug-ins." XAspects uses AspectJ and the XAspects compilation architecture as the implementation platform.

In section 8.7 of [8] the point is made that the model of modular language extensions is very appropriate for AOP. A modular language extension can be plugged into whatever configuration of language extensions we currently use. Advantages of modular language extensions are enumerated (they all apply to XAspects): declarative representation, simpler analysis and reasoning, domain-level error checking, domain-level optimizations.

In section 11.6 the point is made that Intentional Programming [25] is a suitable technology for implementing modular language extensions. We find that an XAspects-based approach that builds on AspectJ is more promising because it is easier to implement crosscutting abstractions when AspectJ is available as the underlying "assembly" language.

Jim Hugunin's paper [13] addresses domain-specific aspect languages. In the section "Aspects in specialized domains" he writes: "While aspect libraries can provide some domain-specific support in a general purpose AOP language such as AspectJ, it is likely that some domains will be important enough to warrant the creation of domain-specific AOP languages." XAspects provides one approach to facilitate the creation and implementation of domain-specific languages.

Jim Hugunin later says that further into the future, work is needed to determine the right way to enable programmers to extend AspectJ without operating on the compiler. XAspects offers one way to extend AspectJ without operating on the compiler and by only relying on the public interface of AspectJ. This is achieved by relying on AspectJ's around methods and Java Reflection. It remains to be seen how general this approach is.

Multi-Dimensional Separation of Concerns (MDSC) [28] is the result of Subject-Oriented Programming [12] and Subject-Oriented Design [6]. MDSC is a generalization of aspect-oriented programming that deals not only with programming but also with design and analysis. The Hyper/J tool from IBM that implements MDSC uses hyperslices and hypermodules. Hyper/J could be simulated in XAspects by using a Hyper/J plug-in that would be allowed to create new Java packages. Hyperslices define reusable aspect behavior and hypermodules provide the glue code. The multidimensional concern ideas are useful for designing a system in XAspects. Ossher and Tarr [23] discusses integrating features with different domain models, a methodology that is also useful for XAspects.

The Generic Model Editor (GME) environment [10] is concerned with domain-specific modeling. GME uses multiple weavers, while XAspects uses a single weaver. Other relevant, but different, tools for domain-specific component languages are ExCIS [3], JTS [4], and DiSTiL [26].

Java syntax extenders, such as Maya [2], are similar to XAspects because they provide a mechanism for compile-time execution of external code generators. However, XAspects is different than such macro systems because XAspects allows the use of AspectJ's aspect weaver. The support of a weaver enables many crosscutting concerns to be both encapsulated and separated in a way that a macro system cannot handle.

ELIDE [5] is also related to Maya and XAspects: ELIDE allows user-defined modifiers to be applied to classes, fields, and methods. The trasformations the modifiers produce can create new classes or change the members of the class to which the modifier was applied. ELIDE's technique of creating a partial order based on dependency requirements may be useful to XAspects if a family of plug-ins need to be tightly integrated and share information.

XAspects is a limited reflective compiler. OpenJIT [20] is a compiler that allows for language customization and optimization based on computational reflection. XAspects

uses reflection in a principled way by giving access to the binary or abstract syntax tree of an intermediate stage.

The Concern Manipulation Environment (CME) [33] supports aspect-oriented software developers (end users), tool providers, and researchers. It is a competitor to XAspects that is much larger in scale. The rest of the paragraph is quoted with small changes from [33]: CME will offer end users a suite of tools for use in creating, manipulating, and evolving aspect-oriented software, across the full software lifecycle. For AOSD tool providers and researchers, it supplies a flexible, open set of components and frameworks on which to build a wide range of tools and paradigms that support aspect-oriented software development, and to do so much more easily than is possible at present. Moreover, the CME provides a common platform in which different AOSD tools can interoperate and integrate, ultimately providing end-users with a rich environment in which to perform aspect-oriented software engineering, and providing other tool builders and researchers with more powerful building blocks from which to create new tools and paradigms.

We believe that the Demeter technology in XAspects will be useful to CME. This includes the meta-model behind XAspects and the AP Library [15, 16] which plays an important role in concern composition.

7. FUTURE WORK

XAspects provides not only a framework for domain-specific component and aspect languages, it is also a framework for creating new language constructs. For example, the MultiJava language [7] has attractive features that we believe can be entirely implemented using the plug-in architecture. To implement multimethods in XAspects a plug-in will need to generate the multiple dispatch tables on its own. This requirement can be satisfied by the compile time reflection and AspectJ join-point model capabilities of the system.

More aspect languages can be found and modified to fit the XAspects framework. Many aspect-oriented solutions have proven useful because they solve concerns fit for particular domains. Unfortunately, these solutions are not used as often due to the burden of integrating them with other solutions. XAspects allows for these solutions to get used without sacrificing the use of a general purpose aspect language.

One future project would be to reimplement a subset of Java Web Services [36] (JAX*) using XAspects. JAX* uses many sublanguages and some of them are aspectual. For example, JAX* contains JAXB (Java Architecture for XML Binding) and JAXP (Java API for XML Processing). It seems that the two do not work ideally together but the intent is similar: We need schemas and their translation to Java (this is the purpose of JAXB) and we need a high-level language for processing the Java objects. JAXP uses XPath as the navigation language and XPath has similar capabilities as traversal specifications.

An interesting application would be to use XAspects to reengineer the Web Services Pack. How would JAXB, JAXP, JAX-PRC, JAXM, JAXR look like if reimplemented in XAspects? We already have a beginning: A subset of JAXB corresponds to the ClassDictionary plug-in and a subset of JAXP corresponds (at least in intent) to the Traversal plug-in.

XAspects needs a more general join point model than AspectJ. The different aspect languages (and class dictionaries) touch each other either through a common terminology or through an intermediate glue language. Techniques from composing diverse ontologies might be useful [30].

The QuO project at BBN [31, 24, 29] uses multiple aspect languages. It would be an interesting exercise to reengineer QuO in the context of XAspects. For example, we would need an aspect to express regions for certain system measurements to formulate how the application should behave in the different regions.

Future development of XAspects will be reported on the XAspects home page [1].

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] XAspects Home Page. http://www.ccs.neu.edu/research/demeter/xaspects. Continuously updated.

[2] J. Baker and W. C. Hsieh. Maya: Multiple-Dispatch Syntax Extension in Java. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281. ACM Press, 2002.

[3] D. Batory, D. Brant, M. Gibson, and M. Nolen. ExCIS: An Integration of Domain-Specific Languages and Feature-Oriented Programming. www.isis.vanderbilt.edu/sdp, Nov. 2001.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.

[5] A. Bryant, A. Catton, K. D. Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 10–18. ACM Press, 2002.

[6] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–339. ACM Press, 1999.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the conference on Object-oriented programming,*

systems, languages, and applications, pages 130–145. ACM Press, 2000.

[8] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, New York, NY, 1994.

[10] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, October 2001.

[11] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[12] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, Oct. 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[13] J. Hugunin. The Next Steps For Aspect-Oriented Programming Languages. www.isis.vanderbilt.edu/sdp, Nov. 2001.

[14] K. Lieberherr, D. H. Lorenz, and P. Wu. A Case for Statically Executable Advice: Checking the Law of Demeter With AspectJ. In M. Aksit, editor, *Second International Conference on Aspect-Oriented Software Development*, Boston, 2003. ACM Press.

[15] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. http://www.ccs.neu.edu/research/demeter/AP-Library/.

[16] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of Object Structures: Specification and Efficient Implementation. *ACM Trans. Prog. Lang. Syst.*, 2003. to appear.

[17] C. V. Lopes. Graph-based optimizations for parameter passing in remote invocations. In L.-F. Cabrera and M. Theimer, editors, *4th International Workshop on Object Orientation in Operating Systems*, pages 179–182, Lund, Sweden, August 1995. IEEE, Computer Society Press.

[18] C. V. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, Graduate School of the College of Computer Science, Northeastern University, Boston, MA, 1997.

[19] C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In R. Pareschi and M. Tokoro, editors, *European Conference on Object-Oriented Programming*, pages 81–99, Bologna, Italy, 1994. Springer Verlag, Lecture Notes in Computer Science.

[20] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT A Reflective Java JIT compiler. In *OOPSLA '98*, 1998.

[21] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A Case-Study for Aspect-Oriented Programming. Technical Report SPL97-009, Xerox Palo Alto Research Center, February 1997.

[22] D. Orleans and K. Lieberherr. AP Library: The Core Algorithms of AP. Technical report, Northeastern University, May 1999. http://www.ccs.neu.edu/research/demeter/AP-Library.

[23] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, October 2001.

[24] R. E. Schantz and D. C. Schmidt. Research Advances in Middleware for Distributed Systems: State of the Art. *IFIP World Computer Congress*, August 2002.

[25] C. Simonyi. The death of computer languages. Technical report, Mircosoft Research, 1995. ftp://ftp.research.microsoft.com/pub/tech-reports/Summer95/TR-95-52.doc.

[26] Y. Smaragdakis and D. Batory. DiSTiL: A transformation library for data structures. In *Domain-Specific Languages (DSL) Conference*, pages 257–270, 1997.

[27] J. Sung. Aspectual Concepts. Technical Report NU-CCS-02-06, Northeastern University, June 2002. Master's Thesis, http://www.ccs.neu.edu/home/lieber/theses-index.html.

[28] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering*, pages 107–119, Los Angeles, 1999. ACM.

[29] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. Loyall, and R. E. Schantz. Total Quality of Service Provisioning in Middleware and Applications. *The Journal of Microprocessors and Microsystems*, 26(9–10), March 2003.

[30] G. Wiederhold and J. Janninck. Composing diverse ontologies. In *Proc. 8 th IFIP working group on databases working conference on database semantics*, Rotorua(NZ), 1999. http://www-db.stanford.edu/SKC/publications/ifip99.html.

[31] J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Systems, John Wiley and Sons, Inc.*, 3(1):19 pages, January 1997.

[32] The AspectJ website. http://www.eclipse.org/aspectj.

[33] The Concern Manipulation Environment website. http://www.research.ibm.com/cme.

[34] The DAJ website. http://www.ccs.neu.edu/research/demeter/DAJ.

[35] The DemeterJ website. http://www.ccs.neu.edu/research/demeter.

[36] Java Web Services, Sun Microsystems. http://java.sun.com/webservices. Continuously updated.