AspectC2C: a Symmetric Aspect Extension to the C Language

Danfeng Zhang, Yao Guo, Xiangqun Chen

Key laboratory of High Confidence Software Technologies, Ministry of Education
Institute of Software, School of Electronics Engineering and Computer Science, Peking University
zhdf@os.pku.edu.cn, {yaoguo, cherry}@sei.pku.edu.cn

Abstract

By separating crosscutting concerns into modules, aspect-oriented programming (AOP) can greatly improve the maintainability, understandability and reusability of software. However, the asymmetric paradigm adopted by most AOP extensions could bring crosscutting concerns into the aspect code and thus limit the reusability of aspects.

Symmetric paradigms have been proposed to alleviate such limitations, but few extensions on such paradigm target at non-object-oriented languages, such as C. In this paper, we propose a symmetric aspect extension to the C language, called AspectC2C, and discuss implementation issues and benefits of this new extension comparing to the asymmetric ones.

Keywords Aspect-Oriented Programming, Symmetric Paradigm, Aspect Extension, C Language

1. Introduction

Recent research on aspect-oriented programming (Kiczales et al. 1997) have shown that, separating crosscutting concerns into modules can eliminate the scattered and tangled code in the base code, thus greatly improving the maintainability, understandability and reusability of legacy software (Tarr et al. 1999; Hannemann and Kiczales 2002; Rajan and Sullivan 2005).

Most aspect extensions nowadays use an *asymmetric paradigm*, in which "aspects" are composed (woven) into components that implement a "base" model. In this paradigm, while *aspects* can be woven into the base code, they can not be advised or instantiated. A *symmetric paradigm*, on the other hand, makes no distinction between components and aspects, and does not mandate a distinguished base model. It was shown that the symmetric paradigms are more suitable for independent development, and code written in these paradigms are more reusable than that in the asymmetric ones (Ossher and Tarr 2001; Harrison et al. 2002; Rajan and Sullivan 2005).

In this paper, we present our effort on implementing a symmetric aspect extension to the C language and discuss several issues while implementing it. This new extension is called AspectC2C[1], and a prototype weaver has been implemented to demonstrate the feasibility of AspectC2C. To the best of our knowledge, there is no current work on implementing similar symmetric paradigms on non-object-oriented languages such as C.

An example of refactoring a readers and writers problem into aspect-oriented program is used throughout this paper. The experiment shows that can overcome the limitations of symmetric paradigms in the context of C, and the code written in it is more reusable, comprehensible and suitable for independent development.

The rest of this paper is structured as follows: in the next section, we give a brief introduction to AspectJ. A readers and writers problem and how to refactor it in asymmetric paradigm is shown in section 3. We discuss the specification of AspectC2C and implementation issues in section 4. The benefits of AspectC2C are discussed in section 5. We conclude this paper in section 6.

2. Background

To make this paper self-contained, we will make a brief review of the most frequently used asymmetric aspect extension on Java, *AspectJ*, in this section.

In *AspectJ*, the crosscutting concerns are encapsulated into a new entity called *aspect*. *AspectJ* adds five constructs to Java: join point, pointcut, advice, inter-type declaration, and aspect. We present a simple example below to explain the basic concepts.

```
1 aspect trace {
2   pointcut traced():
3       execution(* foo(..));
4   before():traced() {
5       System.out.println("tracing");
6   }
7 }
```

A **join point** (line 3) defines points in the base code that are exposed for possible modification by aspects. Here, *execution(* foo(..))* refers to all executions of the function named "foo" with arbitrary parameter and return type. A **pointcut** (line 2) is a set of join points. An **advice** (line 4-

[1] AspectC2C implies that our aspect extension weaves plain C code to plain C code

6) serves as a *before*, *after*, or *around* method to provide an extension at each join point in a pointcut. An **aspect** (lines 1-7) is a class-like construct that modularize a concern in a system. Aspects also support the data abstraction and inheritance abilities like classes. **Inter-type declarations** (also known as "introduction") are declarations that cut across classes and their hierarchies, which are not used in this example. The precedence of aspects can be defined using *declare precedence* statement.

Using asymmetric paradigms, such as *AspectJ*, a program can be divided into two parts: base code and aspect code (Lamping 1999). A concern is *crosscutting* if its realization in pre-AOP designs leads to scattered or tangled code. *Scattered* means not local to a module but fragmented across a system. *Tangled* means intermingled with code for other concerns (Kiczales et al. 1997).

Most aspect extensions for OO languages are based on the asymmetric paradigm, which is similar to the *AspectJ* model presented above. Most aspect extensions for C follow similar asymmetric paradigms, such as AspectC (Coady et al. 2001), AspectC++ (Spinczyk et al. 2005), Arachne (Douence et al. 2005), ACC[2] (more tools can be found on the AOSD web site[3]).

Symmetric paradigms have been discussed and adopted for OO languages (Ossher and Tarr 2001; Harrison et al. 2002; Rajan and Sullivan 2005). However, to the best of our knowledge, there is no current work on implementing symmetric paradigms on non-object-oriented languages such as C.

3. Limitations of Asymmetric Paradigm for the C Language

In this section, we implement a well-known synchronization model, and then try to refactor it to aspect-oriented version using an asymmetric aspect extension to C.

3.1 The Readers and Writers Problem

The problem of guaranteeing exclusive access to a shared resource in a system of cooperating sequential process can be modeled as the "readers and writers problem", which is widely used for synchronization in operating systems and database systems.

There are two classes of processes in this problem: the first named *writers*, who must have exclusive access; the second named *readers*, who may share the resource with an unlimited number of other readers. The access between the readers and writers is exclusive.

According to different priorities of the readers and writers, this problem can be stated as two different ones: the readers first problem and the writers first problem. The best known solution to these problems was proposed by Courtois et al. (Courtois et al. 1971).

[2] http://research.msrg.utoronto.ca/acc/

[3] http://aosd.net/

3.1.1 Problem 1: the Readers First Problem

The readers first problem demands a higher priority of the readers: no readers should be kept waiting unless a writer has already obtained the permission to use the resource; i.e. no reader should wait simply because a writer is waiting for other readers to finish. The solution in C is shown in Figure 1. *Mutex* (mutual exclusion) lock, *semaphore*, and *P, V* operations in this solution are commonly used synchronizing primitives (Dijkstra 1968).

3.1.2 Problem 2: the Writers First Problem

The writers first problem demands a higher priority of the writers: once a writer is ready to write, he performs his "write" as soon as possible (whenever no readers or writers are accessing the resource). The solution in C is shown in Figure 2.

We observed the following features in the solutions:

1. The solution to Problem 2 contains duplicate code existing in the solution to Problem 1. Only the code in grey background in Figure 2 is new in this solution (*mutex1* and *w* corresponds exactly to the use of *mutex* and *w* in the solution to the readers first problem).
2. It is obvious that, the solutions of these two problems in the C language is full of tangled code that handles synchronization with the read and write operations. These tangled code can be encapsulated into aspects.

As there are many crosscutting concerns in this solution, we will first use an asymmetric aspect extension to refactor these solutions.

3.2 Refactoring with an Asymmetric Paradigm

A generic aspect extension to C with AspectJ like grammar is used in this part, because there is no asymmetric aspect extension to the C language that is as widely used and mature as AspectJ. It has the same five new constructs as in AspectJ, and a weaving sequence support is assumed, such as the *declare precedence* in AspectJ. However, due to the limitation of the C language, abstract and inheritance abilities of "aspects" are not supported.

The refactored code for the readers first problem and the writers first problem is shown in Figure 3.

In the refactored version, we use a more comprehensible way to encapsulate the concerns. It is straightforward to encapsulate all the concurrent code for the readers first problem into *readerfirst.aspect*, and all the concurrent code for the writers first problem into *writersfirst.aspect*. However, the refactored code in this manner may contain a lot of duplicate code that already exists in *readerfirst.aspect*, because of the duplicate code in the solutions we pointed out earlier.

To eliminate the duplicated code, the weaving sequence support is used to offer a more comprehensible solution. The solutions can then be divided into two meaningful aspects.

```
int readcount=0;
semaphore mutex=1;
extern semaphore w ;
P(mutex) ;
  readcount = readcount + 1;
  if (readcount == 1)
    P(w);
V(mutex);
read();
P(mutex)
  readcount = readcount - 1;
  if (readcount == 0)
    V(w);
V(mutex);
```

Reader.c

```
semaphore w=1;
P(w);
write();
V(w);
```

Writer.c

Figure 1. Readers First Problem

```
int readcount=0;
semaphore mutex1=1,mutex3=1,
  r=1;
extern semaphore w;
P(mutex3);
  P(r);
    P(mutex1) ;
      readcount=readcount+1;
      if (readcount == 1)
        P(w);
    V(mutex1);
  V(r);
V(mutex3);
read();
P(mutex1)
  readcount=readcount-1;
  if (readcount == 0)
    V(w);
V(mutex1);
```

Reader.c

```
int writecount=0;
semaphore mutex2=1, w=1;
extern semaphore r;
P(mutex2)
  writecount=writecount+1;
  if (writecount == 1)
    P(r);
V(mutex2);
P(w);
write();
V(w);
P(mutex2);
  writecount=writecount-1;
  if (writecount == 0)
    V(r);
V(mutex2);
```

Writer.c

Figure 2. Writers First Problem

```
1. void reader(){
2.    read();
3. }
4.
5. void read(){
6.  // reading
7. }
```

reader.c

```
1. void writer (){
2.    write();
3. }
4.
5. void write(){
6.  // writing
7. }
```

writer.c

```
1. aspect readerfirst() {
2.   semaphore mutex=1, w=1;
3.   int readcount=0;
3.   pointcut Read():
4.     call(void read());
5.   pointcut Write():
6.     call(void write());
7.   before():Read() {
8.     P(mutex);
9.       readcount++;
10.      if ( readcount==1)
11.        P(w);
12.    V(mutex);
13.  }
14.  after():Read() {
15.    P(mutex);
16.      readcount--;
17.      if(readcount==0)
18.        V(w);
19.    V(mutex);
20.  }
21.  before():Write() {
22.    P(w);
23.  }
24.  after():Write() {
25.    V(w);
26.  }
27.}
```

readerfirst.aspect

```
1. aspect writerfirst(){
2.   declare precedence:
3.     readerfirst;
4.   semaphore mutex2=1,
5.     mutex3=1, r=1;
6.   int writecount=0;
7.
8.   pointcut Reader():
9.     execution(void read());
10.  pointcut Read():
11.    call(void reader());
12.  pointcut Writer():
13.    execution(void write());
14.
15.  before():Reader() {
16.    P(mutex3);
17.      P(r);
18.  }
19.  before():Read() {
20.      V(r);
21.    V(mutex3);
22.  }
23.  before():Writer() {
24.    P(mutex2);
25.      writecount++;
26.      if (writecount==1)
27.        P(r);
28.    V(mutex2);
29.  }
30.  after():Writer() {
31.    P(mutex2);
32.      writecount--;
33.      if (writecount==0)
34.        V(r);
35.    V(mutex2);
36.  }
37.}
```

writerfirst.aspect

Figure 3. Refactor using an Asymmetric Paradigm

The first aspect *readerfirst.aspect* encapsulates the synchronization operation needed for exclusive access between readers and writers, and guarantees the readers' priority.

Writerfirst_1.aspect solves the writers first problem. It grantees the priority of the writers. By declaring *readerfirst.aspect* as its precedence in line 2 and 3, this aspect reuses *readerfirst.aspect*.

3.3 Limitations of Asymmetric Paradigm

Although the refactored version presented above removes the crosscutting concerns in the base code, and the aspects can be reused with a weaving sequence support, some limitations still remains in the asymmetric paradigm.

The main problem of the refactored version is that, crosscutting concerns still remain in the aspect code. For example, line 8, 12, 15, 19 in *readerfirst.aspect* use a mutex lock to guarantee the exclusive access of variable, so are line 24, 28, 31, 35 in *writerfirst.aspect*. Lines 7-26 in *readerfirst.aspect*, which make the readers prior than writers, and lines 15-36 in *writerfirst.aspect*, which make the writers prior than readers are almost the same, except for the pointcut declarations.

Based on this observation, we identify the following limitations of asymmetric paradigm which caused the problem above:

1. While the aspect code can advise base code, they can not be advised by other aspects. The advice code in them can not be used alone, they can take effect only when they are woven to the base code.
2. The advice code is reusable most of the time while the pointcuts are not. Encapsulating advice and pointcuts

in a single aspect construct limits the reusability of the advice code. The abstraction and inheritance abilities in AspectJ may solve this problem to some extent, but they are not supported in non-OO languages, such as C.

3. The aspects are not "instantiable". This means that the aspects can not be instantiated similar to "classes" in OO languages. Thus they can not be woven to the base code more than once, which limits the reusability of aspect code.

4. Implementation of AspectC2C

To solve the problems mentioned above, we present AspectC2C, a symmetric aspect extension to the C language.

4.1 AspectC2C Specification

In this part, we introduce the specification of an aspect extension to the C language, AspectC2C in detail. This specification includes concern encapsulation and binding specification.

4.1.1 Concern Encapsulation

In AspectC2C, there is no distinction between the base code and the aspect code as in asymmetric ones. The base code and the aspect code are both written in plain C code. They can be used as normal C code, as advices, or as the base code to be advised.

We use a simple trace example here to explain the usage of AspectC2C. To implement a trace aspect as the one we shown in section 2, we should implement two concerns, one represents the base code, and the other represents the aspect code, both in plain C code. As shown in Figure 4, they can be encapsulated into files *Base.c* and *Trace.c* (to save space, we eliminated the include statements).

We notice that, there is no information about how these concerns should crosscut each other. The binding information is specified in separate XML format files, which we introduce below.

4.1.2 Binding Specification

The binding files are used to specify where to add the advice code and what advice code to insert, which are usually expressed in aspects in asymmetric aspect extensions. To make the binding language more familiar to the developers of AspectJ like language, we used some terminologies from AspectJ.

The binding files follow the DTD (Document Type Definition) definition shown in Figure 5. We choose the XML format for better readability and because it is easier for the programmers to learn.

The binding files contain the following elements (words in *italic* are elements):

- *Bindings*: a bindings element may contain one or more *bind* elements.

```
<!ELEMENT bindings (bind+)>
<!ELEMENT bind (base+)>
<!ATTLIST bind advice CDATA #REQUIRED>
<!ELEMENT base (pointcut+, outfile?)>
<!ATTLIST base file   CDATA #REQUIRED>
<!ELEMENT pointcut ((advice, args?)+) >
<!ATTLIST pointcut function CDATA #REQUIRED
          type CDATA #REQUIRED>
<!ELEMENT advice EMPTY>
<!ATTLIST advice function CDATA #REQUIRED
          type CDATA #REQUIRED>
<!ELEMENT args (#PCDATA)>
<!ELEMENT outfile (#PCDATA)>
```

Figure 5. DTD Definition

- *Bind*: a bind element represents one single iteration of adding advice code to base code. A bind can have more than one *base* elements, but only one advice file is allowed to be defined as an attribute[4].
- *Base*: a base element specifies two information: one base file name as an attribute, and one or more *pointcut* elements. *Outfile* is optional, used to specify the output file name of this iteration. The default name is "temp.c".
- *Pointcut*: A pointcut element represents a pointcut and advice pair. One pair defines one *pointcut* and one or more *advices* that are applied on that *pointcut*. The attributes of pointcut specify the point in the base code exposed to the advices. The prototype weaver for AsepctC2C supports function call and function execution pointcut types for now. *Args* is optional, only used for passing parameters to the advice function.
- *Advice*: as in AspectJ, an advice specifies what code to insert into the pointcuts. The difference in our extension is that these actions are implemented in plain C, instead of being encapsulated in *aspect* in AspectJ. Three advice types are supported in our prototype weaver, "before", "after" and "around".

Using this format, we define the binding file *Binding.xml* for the trace example as shown in Figure 4. Specifically we show how to specify a before execution advice as the example shown in section 2. The "exec" ponitcut exposes two points in the base code for advices, and the "before" advice choose the first place to add function "trace" into file "Base.c".

4.2 Implementation Issues

We discuss three issues related to the implementation of the AspectC2C in this section. These issues are important because they are the solutions to the three limitations of asymmetric paradigm we introduced in section 3.3. Fur-

[4] This limitation is introduced by the name substitution mechanism we used, which we will discuss in next part

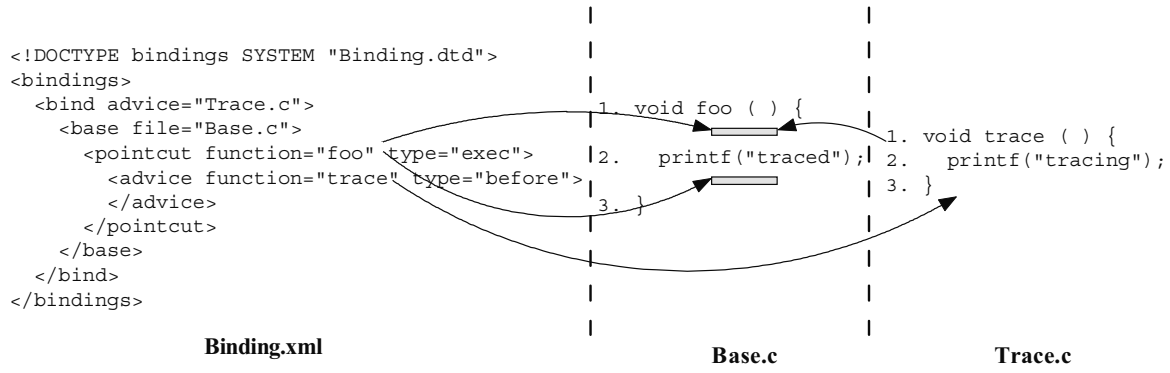

Figure 4. Example for CweaveC

thermore, the implementation of these three issues are different from that in OO languages.

4.2.1 Encapsulation of Concerns

In AspectC2C, both base code and advice code should be encapsulate into unified entities, to avoid the first problem in section 3.3.

For symmetric extensions targeting at OO languages, concerns are usually encapsulated into *classes* (Rajan and Sullivan 2005), or *hyperslice* in Hyper/J (Ossher and Tarr 2001), which still consists of normal *classes*. In non-OO languages, it would be straightforward to encapsulate them into C files. As we mentioned in section 4, a plain C file is defined to encapsulate a single concern in AspectC2C.

There are two possible code styles for the C file encapsulating a crosscutting concern.

The first style is to extract the data in the implementation to the files who use them. For example, a typical implementation of mutex lock only implement operations needed, and define the semaphores as a parameter of them. It does not maintain the data itself. The other files who call mutex lock must define the semaphore themselves, whose initial value is always 1.

Concerns written in this style can be supported in AspectC2C using the "introduction" mechanism most aspect extensions support. With this mechanism, the programmer who writes the binding files can introduce the variables needed to the base code. However, this will obstruct independent developments, for they should now care about which variables to introduce and what their initial value are.

The second style is to defined the data inside the implementation files. The code written in this style is similar to *classes* in OO languages, we call it **the OO style concern**.

The binding file programmer can ignore the implementation detail of concerns using this style. Concerns written in this style are recommended in AsepctC2C. However, the problem of this style is that the C files can not be instantiated. So if they are used more than once, one operation in one file may implicitly affect the others. We introduce a mechanism to avoid this problem in section 4.2.3.

4.2.2 Extracting of Binding Information

As we pointed out in section 3.3, encapsulating advice and pointcuts code into a single construct may limit the reusability of aspects.

In aspect extensions for OO languages, this requirement may be not so significant. For example, by using an abstract "aspect" with abstract "pointcuts", inheriting this abstract "aspect" and implementing the abstract "pointcuts" will make the abstract "aspect" and the advice code inside reusable. However, there are no abstraction and inheritance abilities in C.

In our extension, we use separate XML format files to specify the binding information, as we have shown in section 4.1. This design will make the advice code (in C files) more reusable.

4.2.3 Implementation of "instantiable" concerns

Rajan and Sullivan (Rajan and Sullivan 2003) have pointed out the importance of instance-level aspects for integrated system design. It will be more reusable for the aspect when it can be instantiated for many different base code.

As discussed earlier, the OO style concerns are more suitable for independent development. However, as the C files can not be instantiated, if they are used more than once, one operation in one file may implicitly affect the others. A *name substitution mechanism* is used in our extension to solve this problem in AspectC2C weaver.

AspectC2C weaver introduces global variables in the advice files automatically to the base files they advice with substituted names. For example, if there is a variable x whose initial value is 1 in the mutex lock implementation, and this concern is required by the binding file to advise functions in the file foo.c, then a global value named *newx* will be introduced to foo.c automatically, and all the reference to this variable in the advice code instrumented will refer to *newx*, not x anymore. The concerns are now "instantiable" using this mechanism. In our implementation, a random name is generated to avoid duplicate names.

For some cases, it may be necessary to share global variables among several files. Such requirements can be met with the well designed binding file format. As speci-

fied in 4.1.2, each *bind* must specify an advice file attribute *advice*. And each *base* element inside it must specify an base file attribute *file*. Our aspect weaver will generate a set of different names for global variables in the advice file, and introduce them into each base file advised (in the first file as declaration and as shared variable in other files). Thus, the variables are shared among all the base files in a single *bind* element.

4.3 Prototype Weaver Implementation

We have implemented an aspect prototype weaver for AspectC2C. This prototype is a source-to-source translator built on top of the ANTLR[5] parser generator tool, formerly known as PTTCS. The grammar for GNU C[6] is used to generate the abstract syntax tree (AST), and a code emitter grammar based on this AST is used to emit the final C code. The architecture of this weaver is shown in Figure 6. The current version of prototype weaver just operates on the AST and thus only static pointcuts are supported.

The code transformation is based on a sequence of transformation actions defined in binding files acting on the AST of the base program. However, before adding the advice tree to the base tree, name substitution should be applied as we mentioned above.

5. Refactoring of the Readers and Writers Problem using AsepctC2C

In this section, we review the problem of synchronization in the readers and writers problem introduced in section 3. We will show how to refactor it using AspectC2C.

5.1 Separation of Concerns in the Readers and Writers Problem

The first step to refactor the solutions for the readers and writers problem is to identify the concerns in it. Four concerns are identified in this problem: read, write, the mutual exclusion operation, and synchronization between different roles (readers and writers in this example). We show the implementation of each concern in Figure 7. To keep it simple, we eliminated the include statements.

In the refactored code, each concern is self-contained, which handles its own responsibility, ignoring any potential interactions between each another (the inter role concern uses P, V operation just for synchronization between two role). Each concern can be developed by different programmers, who just need to concentrate on their own duty.

5.2 Evolution 1: Add Mutex Lock to Counters

At this step, we first make *sync.c* in Figure 7 usable in a concurrent environment.

While implementing the inter role synchronization concern, *sync.c*, the programmer ignores the exclusive access to the variable *count*. However, in a concurrent environment, the variables should be locked by a mutex lock whenever they are accessed. We can accomplish this by adding the mutex lock operation to *sync.c* with the binding file shown in Figure 8.

This binding file adds a mutex lock to variable access in *sync.c* and generates an output file *tempsync.c*, which is specified by the *out* element.

This step shows that, in AspectC2C, the advice code (*sync.c*) that can not be advised in asymmetric paradigm before can now be advised. The concerns in AspectC2C are much more reusable than in asymmetric ones.

5.3 Evolution 2: The Readers First Problem

With the identified concerns, we can evolve our program to solve the readers first problem with a binding of implemented concern. All we have to do is using the synchronized *sync.c*, *tempsync.c*, generated in the last evolution step to make readers prior to the writers. The binding file could be implemented as in Figure 9.

We should notice that, the internal file *tempsyn.c* is used here and the global variables are shared between *reader.c* and *writer.c*, which shows the flexibility of our extension .

5.4 Evolution 3: The Writers First Problem

In this step, we need to make the writers prior than all the readers. The binding file could be implemented as Figure 10.

The "instantiable" concern is applied in this step. As we discussed in section 4.2.3, our name substitution mechanism will generate different names in different *bind* elements. After the weaving of the binding file in this step, reader.c and writer.c will share the semaphore *role* in sync.c with a new name. This binding will not interfere with the one in Figure 9, because it is in another bind element. The same concern can be woven more than once in AspectC2C, which improves the reusability of concerns. Also, by encapsulating binding information in binding files, the advice code ("tempsync.c") can be reused by different binding files.

5.5 Benefits of the AspectC2C

Through the three evolution steps above, we accomplished the process from four separate concerns to a better solution to the writers first problem. The benefits of AspectC2C regarding the limitations of asymmetric paradigms shown in section 3.3 are:

1. There is no distinction between base code and advice code in AspectC2C. Both of them are written in plain C code. They can be used as plain C code, as advice, or as base code to be advised.
2. The pointcut information is extracted from the aspect construct in the asymmetric model. Thus the advice code can be better reused, because they can bind to

[5] http://www.antlr.org/

[6] http://www.codetransform.com/gcc.html

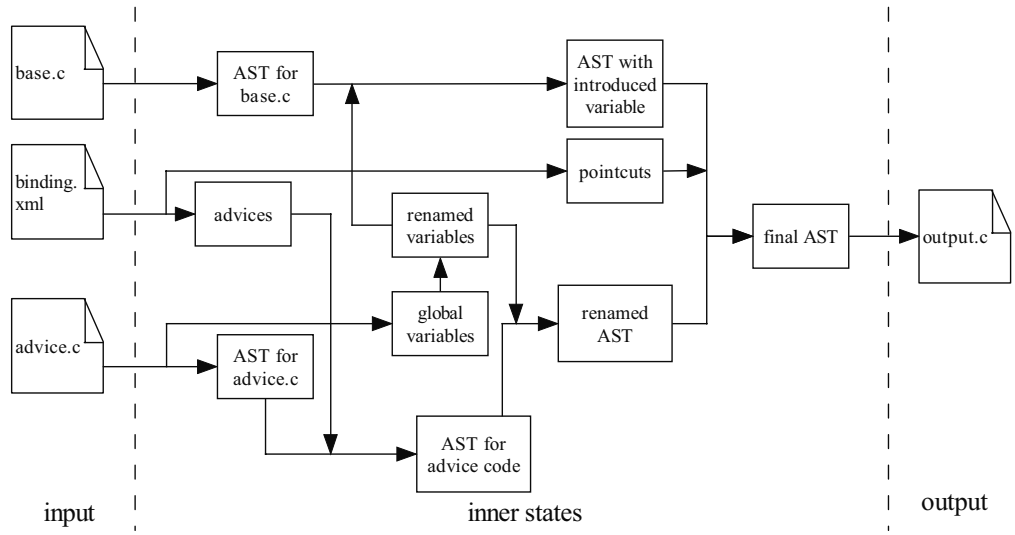

Figure 6. Implementation Overview

```
1. void reader(){
2.     read();
3. }
4.
5. void read(){
6.   // reading
7. }
```

reader.c

```
1. void writer(){
2.     write();
3. }
4.
5. void write(){
6.   // writing
7. }
```

writer.c

```
1. semaphore s=1;
2.
3. void mutexP(){
4.     P(s);
5. }
6.
7. void mutexV(){
8.     V(s);
9. }
```

mutex.c

```
1. int count=0;
2. semaphore role=1;
3.
4. void first_entry(){
5.     count ++;
6.     if (count==1)
7.         P(role);
8. }
9.
10.void last_exit(){
11.    count--;
12.    if (count==0)
13.        V(role);
14.}
15.
16.void lockrole(){
17.    P(role);
18.}
19.
20.void unlockrole(){
21.    V(role);
22.}
```

sync.c

Figure 7. Four Concerns in Reader and Writer Problem

```
<bind advice="mutex.c">
  <base file="sync.c">
    <pointcut function="first_entry" type="exec">
      <advice function="mutexP" type="before"/>
      <advice function="mutexV" type="after"/>
    </pointcut>
    <pointcut function="last_exit" type="exec">
      <advice function="mutexP" type="before"/>
      <advice function="mutexV" type="after"/>
    </pointcut>
    <outfile>tempsync.c</outfile>
  </base>
</bind>
```

Figure 8. Evolution 1 Binding file

```
<bind advice="tempsync.c">
  <base file="reader.c">
    <pointcut function="reader" type="exec">
      <advice function="first_entry" type="before"/>
      <advice function="last_exit" type="after"/>
    </pointcut>
    <outfile>tempreader.c</outfile>
  </base>
  <base file="writer.c">
    <pointcut function="write" type="call">
      <advice function="lockrole" type="before"/>
      <advice function="unlockrole" type="after"/>
    </pointcut>
    <outfile>tempwriter.c</outfile>
  </base>
</bind>
```

Figure 9. Evolution 2 Binding file

```
<bind advice="tempsync.c">
  <base file="tempwriter.c">
    <pointcut function="writer" type="exec">
      <advice function="first_entry" type="before"/>
      <advice function="last_exit" type="after"/>
    </pointcut>
    <outfile>finalwriter.c</outfile>
  </base>
  <base file="tempreader.c">
    <pointcut function="reader" type="exec">
      <advice function="lockrole" type="before"/>
    </pointcut>
    <pointcut function="read" type="call">
      <advice function="unlockrole" type="before"/>
    </pointcut>
    <outfile>tempreader.c</outfile>
  </base>
</bind>
<bind advice="mutex.c">
  <base file="tempreader.c">
    <pointcut function="reader" type="exec">
      <advice function="mutexP" type="before"/>
    </pointcut>
    <pointcut function="read" type="call">
      <advice function="mutexV" type="before"/>
    </pointcut>
    <outfile>finalreader.c</outfile>
  </base>
</bind>
```

Figure 10. Evolution 3

different base code by different bindings, the advice code itself in C needs not to be modified.

3. With the implementation of "instantiable" concerns in AspectC2C weaver, concerns written in C code can be reused more than once.

6. Conclusion

In this paper, we present a symmetric aspect extension, AsepctC2C, to the C language. A prototype weaver has also been implemented. The refactored example using this weaver shows that code written in AspectC2C is more reusable, comprehensible and suitable for independent development than the asymmetric ones.

Although we have presented a working prototype of AspectC2C, further studies are required before it can be utilized in a practical environment. Particularly, we are interested in pursing the following topics:

1. Extending the current prototype to include more advice and pointcut types, as well as dynamic join points. A dynamic aspect weaver for AspectC2C is also worthy of further study.
2. Although the XML format binding files are easy to follow, it may be prolonged to write. Content assist in contemporary IDEs will alleviate this problem, another solution is to develop a description language for binding files, which will greatly simplify the binding code.

References

Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system. In *proc. of European Software Eng. Conf. held jointly with Int'l Symp. Foundations of Software Eng. (ESEC/FSE)*, pages 88–98, 2001.

P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

E. W. Dijkstra. The structure of the "the" multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Sgura-Devillechaise, and M. Sdholt. An expressive aspect language for system applications with arachne. In *proc. of International Conference on Aspect Oriented Software Development (AOSD)*, pages 27–38, March 2005.

J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *proc. of Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, November 2002.

W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. *IBM Research Report*, RC22685(W0212-147), December 2002.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *proc. of European Conference on Object-Oriented Programming(ECOOP)*, June 1997.

J. Lamping. The role of the base in aspect oriented programming. In *proc. of the Workshop on Object-Oriented Technology*, pages 289–291, 1999.

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.

H. Rajan and K. Sullivan. Eos: Instance-level aspects for integrated system design. In *proc. of European Software Eng. Conf. held jointly with Int'l Symp. Foundations of Software Eng. (ESEC/FSE)*, pages 297–306, September 2003.

H. Rajan and K. J. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *proc. of International Conference on Software Engineering (ICSE)*, pages 59–68, May 2005.

O. Spinczyk, D. Lohmann, and M. Urban. Aspectc++: an aop extension for c++. *Software Developer's Jouranl*, pages 68–76, May 2005.

P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. n degrees of separation: multi-dimensional separation of concerns. In *proc. of International Conference on Software Engineering (ICSE)*, pages 107–119, May 1999.