AspeCt-oriented C Language Specification Version 0.9 *[†]

Weigang Gong and Hans-Arno Jacobsen

Middleware Systems Research Group
Department of Computer Science &
Department of Electrical and Computer Engineering
University of Toronto

August 9, 2010

— **Working Technical Draft**.
For updates and changes, please refer to `www.AspeCtC.net`.

Contents

1 Overview **4**

2 Change Log **5**

3 Join Point Model **6**

4 Pointcut **6**

4.1 Primitive Pointcut 7

4.2 Composite Pointcut 10

4.3 Named Pointcut 10

4.4 cflow() Pointcut 11

4.5 Matching Mechanism 13

4.5.1 Simple Character Matching 13

4.5.2 Wildcard Character Matching 13

5 Advice **14**

5.1 Single Advice 14

5.2 Proceed() 16

5.3 Preturn() 16

5.4 “this”: Reflective Information at Join Points 17

5.5 Multiple Advice 19

5.5.1 before/after advices 19

5.5.2 around advices 20

6 Static Crosscutting **22**

6.1 intype() Pointcut 22

6.2 introduce() Advice 23

7 Exception Handling **24**

7.1 try() Pointcut 25

7.2 catch() Advice 26

[*]This research is supported by NSERC.

[†]Starting with Version 0.5, we have changed the name of the language extension from AspectC to AspeCt-oriented C, — ACC for short.

7.3 throw() 27

8 Implementation **27**

8.1 Aspect Compilation 28

8.2 Syntax Analysis 29

8.3 Advice Weaving 29

9 Grammar **31**

9.1 Keywords 31

9.2 Grammar Rules 31

10 Usage **32**

10.1 General Usage 32

10.2 Use “tacc” 34

10.3 Command Line Options 34

1 Overview

AspeCt-oriented C is an implementation of aspect-oriented programming (AOP) for the C programming language. AspeCt-oriented C is an extension to C.

This specification introduces the AspeCt-oriented C programming model and the new language constructs. From here on forward we refer to AspeCt-oriented C as ACC.

The specification is implemented by the ACC compiler that weaves code written in ACC into ACC-unaware ANSI-C code, and generates C sources implementing the aspect-oriented program. These sources can be compiled by any ANSI-C compliant compiler such as gcc.

The current ACC language design adapts the ideas of aspect-oriented programming laid-out in the original paper by Kiczales *et al.* [2] to the C programming language. The ACC language loosely follows the AspectJ programming language design [3] and the partial ACC language design originally suggested by Coady *et al.* [1].

To this end ACC aims at being enabling technology. It is the necessary "evil" and investment in building research infrastructure to eventually lead to further explorations and investigations not possible today, as no stable ACC implementation exists.

Long-term research objectives of the ACC project include the investigation of

1. concern separation support and aspect-oriented language features tailored to the C language and the imperative style of programming
2. aspect-orientation in the context of software written in C, especially systems software and middleware systems, targeting small-scale, embedded systems (e.g., cell phones, PDAs, chip cards, sensor boards etc.)
3. techniques and tools for the development of highly customizable and easily configurable systems and middleware systems software product lines catering to the extensive world of C-based systems.

The ACC implementation and further information can be found on the project Web pages at `www.AspeCtc.net`.

This document is neither a tutorial on ACC, nor a research paper, it simply documents our current ACC implementation and offers the corresponding language specification.

However, this document should suffice to get you started developing ACC programs and sending us bug reports. The ACC distribution contains a lot of test cases illustrating the use of ACC constructs. The project Web pages — `www.AspeCtc.net` — also contain a few examples.

2 Change Log

Changes in ACC Version 0.9 relative to ACC Version 0.8:

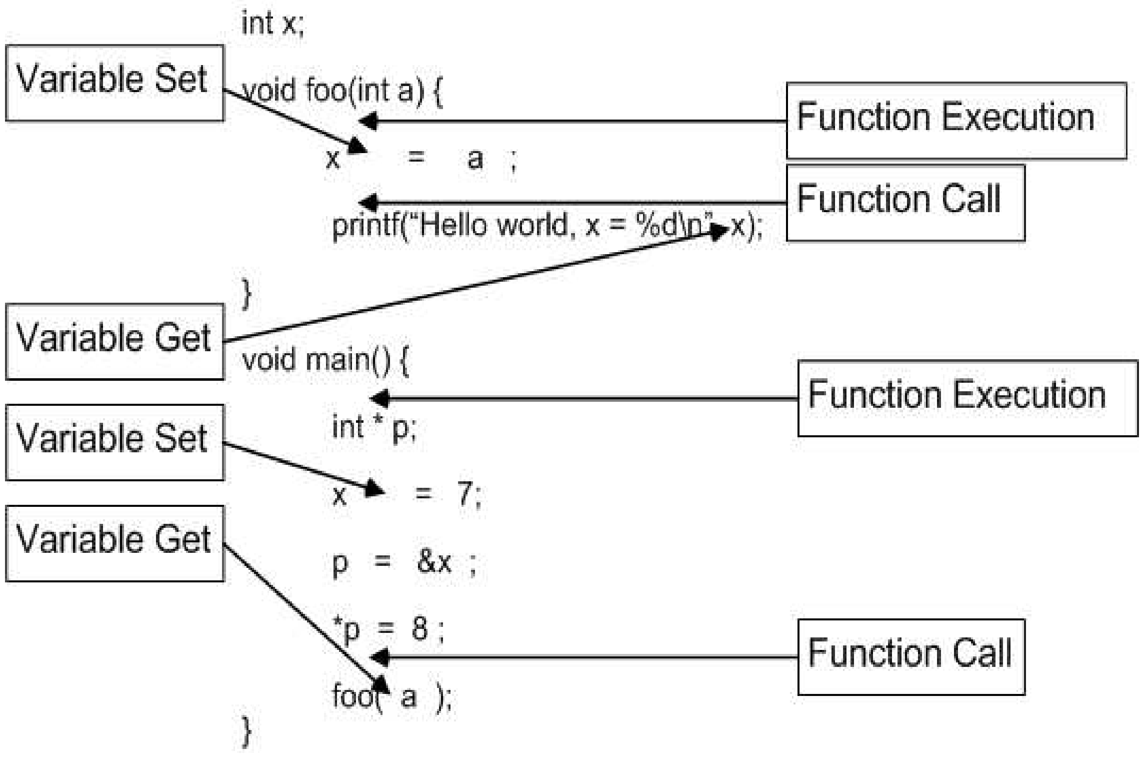
- Section 9.2: more precise description of grammar rules for “set” and “get” pointcut.

3 Join Point Model

A *join point* is a well-defined point in the execution context of a program. Currently, ACC supports the following join points:

1. call join point: the point when a function is called
2. execution join point: the point when a function is executed
3. set join point: the point when a variable is assigned a value [1]
4. get join point: the point when a variable's value is read

The above join points are illustrated in the below program.



```
int x;
void foo(int a) {
    x   =   a  ;
    printf("Hello world, x = %d\n", x);
}
void main() {
    int * p;
    x   =  7;
    p  =  &x  ;
    *p  =  8 ;
    foo(  a  );
}
```

4 Pointcut

A *pointcut* is a language extension representing one or more join points. Currently, ACC supports *primitive pointcuts*, *composite pointcuts*, and *named pointcuts*.

[1]Currently, ACC only supports set/get join points involving global variables with base types. In the future support for other type of variables, like local variables, or struct/union member fields will be considered.

4.1 Primitive Pointcut

A *primitive pointcut* represents one of the join points defined above. The following pointcuts are defined.

1. call(*function-signature*)

   A call pointcut picks out the join points of calling the function specified by *function-signature.*

2. callp(*function-signature*)

   A *callp* pointcut picks out the join points of calling the function specified by *function-signature* through dereferencing a function pointer.

3. execution(*function-signature*)

   An execution pointcut picks out the join points of executing the function specified by *function-signature.*

4. set(*variable-declaration*)

   A set pointcut picks out the join points of setting a value to the variable specified by *variable-declaration.*

5. get(*variable-declaration*)

   A get pointcut picks out the join points of getting the value of the variable specified by *variable-declaration.*

6. args(*a list of types or identifiers*)

   An args pointcut picks out the join points whose parameters' types match the specified types or the types of the specified identifiers.

7. infile("*file name*")

   An infile pointcut picks out the join points which appear in the file specified.

8. infunc(*identifier*)

   An infunc pointcut picks out the join points which appear in the function specified.

9. result(*type or identifier*)

   A result pointcut picks out the join points whose return type matches the specified type or the type of the specified identifier.

Semantics:

1. The *function-signature* must be a valid prototype of a function. It represents the function associated with the call or execution join point.

2. For the callp pointcut, the function name specified in the signature can not contain the wildcard character.

3. The callp pointcut captures function calls by dereferencing the following types of function pointers: global, local and function pointers passed as argument.

   For example, "`callp(void foo(int))`" captures the function calls shown below.

```
void (*gp)(int);
struct A {
        void (*sp)(int);
};
void foo(int a) {
        ...
}
void foo2(void (*ap)(int)) {
        void (*lp)(int);
        (*ap)() ; <— capture
        lp = foo;
        (*lp)(); <— capture
        {
          void (*llp)(int);
          llp = foo;
          (*llp)() ; <— not capture
        }
}
int main() {
        struct A sa;
        sa.sp = foo;
        (*sa.sp)(); <— not capture
        gp = foo;
        (*gp)(); <— capture
        foo2(foo);
}
```

4. For each parameter in the prototype, only its type can be specified.

   For example, "`call(int foo(int ))`" picks out any call to function "`foo`" accepting an `int` parameter and returning an `int`.

5. The *variable-declaration* represents the variable associated with a set or get join point. Wildcard characters can be used in the *variable-declaration*.

6. args or result pointcuts can be used to capture set and get join points of variables. ACC treats variable set and get join points in the same was as function call join points. Suppose the variable is declared as *T V*, a set join point of such a variable is treated as a call to a function whose prototype is *T V(T* , T)*; a get join point is treated as a call to a function whose prototype is *T V(T)*.

   For example, "`args(char)`" picks out a call or execution join point for a function having a "char" argument, or a get join point for a variable of type "char". "`result(int)`" picks out a call or execution join point for a function returning "int", or a set or get join point for a variable of type "int".

7. The identifiers specified in the args or result pointcuts must be declared as a parameter in the advice declaration. The main usage of args and result pointcuts is to expose program context to advice functions.

   For example, "`before(int x):  args(char, x)`" picks out any join points whose parameter types are "char" and "int", and the value of the second parameter is available for use inside the advice function, as follows:

```
before(int x) : execution ( void foo (char , int ) ) && args(char , x) {
        printf("inside before advice, param = %d\n",x );
}
```

8. There is a special format of args() or result() : "`args(* pointer-variable-name)` or "`result(* pointer-variable-name)` ". The meaning is that the parameter type or return type of a join point must match the type after dereferencing the pointer variable.[2] Using this format, advice functions can change the value of the argument passed into a function, as follows:

```
before(int * x) : execution ( void foo (char , int ) ) && args(char , *x) {
        *x = (*x) * 2;
        printf("inside before advice, argument value is doubled\n");
}
```

9. The file name specified in the infile pointcut must be enclosed in quotes, and it should be the name of the input file, not the generated file.

   For example, say the input main file is `t1mc.mc`, if a developer wants to pick out all join points appearing in this file, she must use "`infile("t1mc.mc")`".

10. The identifier specified in the infunc pointcut should be the function name where the join point occurs.

[2] "before(int *x):args(x)" is not the same as "before(int *x): args(*x)". The first matches a join point whose parameter type is "int *", but the latter matches a join point whose parameter type is "int".

4.2 Composite Pointcut

A *composite pointcut* defines a pointcut by composing pointcuts with the following operators: “&&” , “||”, “!” or “()”. The syntax is as follows:

1. pointcut_0 && pointcut_1: returns join points picked out by both pointcut_0 and pointcut_1
2. pointcut_0 || pointcut_1: returns join points picked out by either pointcut_0 or pointcut_1
3. !pointcut_0: returns join points not picked out by pointcut_0
4. (pointcut_0): returns join points picked out by pointcut_0

Semantics:

1. The pointcuts connected by the afore-mentioned operators can be any valid pointcut declaration.

4.3 Named Pointcut

To improve usability of pointcuts, developers can attach a name to a pointcut description, and the name can then be used in places where a pointcut is used. The syntax for attaching a pointcut name is as follows:

pointcut *pointcut-name* (*parameter-list*): *pointcut-description*;

The syntax for using a named pointcut is as follows:

identifier (*identifier-list*$_{opt}$)

For example, the following example shows how to declare a named pointcut and to use the name in two different advices:

```
pointcut callFoo() : call(void foo ()) ;
before() : callFoo() && infunc(main) {
        printf(“before calling foo in function main\n”);
}
before() : callFoo() && infunc(foo2) {
        printf(“before calling foo in function foo2\n”);
}
```

Semantics:

1. The *pointcut-name* can be any valid identifier.

2. The *parameter-list* can be empty, indicating there are no exposed arguments associated with the pointcut.

3. The *pointcut-description* can be any valid pointcut.

4. The *identifier* must be the name of a named pointcut.

5. The number of identifiers in the *identifier-list* must be the same as the number of parameters in the *parameter-list* where the named pointcut is declared.

6. The type of each identifier in the *identifier-list* must be the same as that of the corresponding parameter in the *parameter-list*.

7. The name of each identifier in the *identifier-list* should be declared as a parameter of the corresponding advice or named pointcut, such as:

```
pointcut FirstNamedPC(int z) : call(void foo (int)) && args(z);
before(int j) : FirstNamedPC(j) { ... }
pointcut SecondNamedPC (int w) : FirstNamedPC(w) ;
```

8. The developer can also expose the arguments or the return value by using a named pointcut, as follows:

```
pointcut callFoo(int w) : call(void foo (int )) && args(w);
before(int k) : callFoo(k) && infunc(main) {
        printf("before calling foo in function main, value = %d\n", k);
}
before(int p) : callFoo(p) && infunc(foo2) {
        printf("before calling foo in function foo2, value = %d\n", p);
}
```

4.4 cflow() Pointcut

ACC provides a `cflow()` pointcut to pick out all join points occurring in the dynamic execution context, i.e., the control flow, of other join points. Its syntax is `cflow( pointcut-definition )`.

For example, "`call(void foo(int)) && cflow(execution(void foo3()))`" only picks out the calls to function `foo` under the control flow of function `foo3`.

Given the following advice:

```
void around() : call (void foo(int)) && cflow(execution(void foo3())) {
        printf("skip call of foo in control flow of foo3\n");
}
```

If the advice is applied to the following C code:

```
void foo(int a) {
        printf("in foo\n\n");
}
void foo2() {
        printf("in foo2\n");
        foo(3);
}
void foo3() {
        foo2();
}
int main() {
        printf("call foo in main\n");
        foo(9);
        printf("———\n");
        printf("call foo2 in main\n");
        foo2();
        printf("———\n");
        printf("call foo3 in main\n");
        foo3();
}
```

The output is:

```
call foo in main

in foo

---------

call foo2 in main

in foo2

in foo

---------
```

```
call foo3 in main

in foo2

skip call of foo in control flow of foo3
```

Semantics:

1. The pointcut definition inside `cflow()` can be any valid pointcut definition except another `cflow()` pointcut.

4.5 Matching Mechanism

ACC provides two mechanisms for matching pointcuts with join points – simple character matching and wildcard character matching.

4.5.1 Simple Character Matching

When a plain string is specified in a pointcut's declaration, ACC uses simple case-sensitive string comparison for matching.

For example:

1. "`call(int foo(int))`" picks out any call to function "`foo`" accepting an `int` parameter and returning an `int`.
2. "`args(int, char))`" picks out any call or execution of functions[3] accepting an `int` and a `char` as parameters.
3. "`call(int foo(int)) && infunc(foo2)`" picks out any call of function "`foo`" inside function "`foo2`".

4.5.2 Wildcard Character Matching

ACC uses "$" and "..." as wildcard characters to enhance the matching capability: $ matches any type identifier or any length of continuous strings, including the empty string; ... matches any length item list, including the empty list.

For example:

[3]The args() pointcut could be used to pick out set or get join points. However, this specific args() pointcut is cannot do that, because the function derived from the set() join point takes a pointer type as the first argument, and the function derived from the get() join point takes only one parameter.

1. “`call(i$t f$oo(in$))`” picks out any call to functions which have a name starting with “`f`” and ending with “`oo`”, have a return type starting with “`i`” and ending in “`t`”, and accept one parameter having a type starting with “`in`”.
2. “`args(int, ..., char))`” picks out any call or execution of functions accepting an `int` and a `char` as the first and last parameters.
3. “`call(int foo(int)) && infunc(fo$o2)`” picks out any call of function “`foo`” inside functions whose name starts with “`fo`” and ends with “`o2`”.

Semantics:

1. Developers can use $$ to match one $ inside a target name.
2. ... can only be used when specifying parameter types for a function’s prototype.
3. When the name specified in args() or result() pointcuts has $, ACC searches an advice parameter having the exact same name. If found, the name is bound with the advice parameter, otherwise, ACC treats the name as a type name.

   For example,

   (a) “`before(int x$x): args(char, x$x)`” picks out any call or execution join point whose parameter types are ”char” and ”int”, because “`x$x`” matches an advice parameter having type “int”.

   (b) “`before(): args(char, x$x)`” picks out any call or execution join point whose first parameter type is ”char” and second parameter type’s name starts with “x” and ends with “x”.

5 Advice

5.1 Single Advice

An advice represents the code to be executed when a join point is matched by a pointcut defined inside the advice declaration. Currently, ACC supports the following types of advices:

1. before: code is executed before some join points
2. after: code is executed after some join points
3. around: code is executed instead of code at some join points

The general syntax for an advice declaration is:

type-specifier$_{opt}$ **before**|**after**|**around** (*parameter-type-list*$_{opt}$) : *pointcuts*
{ *function-body* }

Semantics:

1. For before/after advice, the *type-specifier* should not be specified. The ACC compiler uses “void” as the return type of the function generated from the advice.

2. For around advice, the *type-specifier* must be specified, and it becomes the return type of the function generated from the advice. Furthermore, the type-specifier must be the same as the return type of the matched functions.

3. The ACC compiler generate a unique function name for each advice.

4. If the *parameter-type-list* is specified, it becomes the parameter list of the generated function.

5. The information specified by the *pointcuts* is used to match join points.

6. For each parameter name in the *parameter-type-list*, the name must be used inside one pointcut among the *pointcuts*, like args() or result().

For example:

```
before() : execution ( void foo (int ) ) {
        printf(“before execution foo\n”);
}
```

The “before” advice indicates that a message is printed out before the execution of function “foo”.

```
int around() : call ( int foo (int ) ) {
        printf(“around call foo\n”);
        return 100;
}
```

This “around” advice indicates that a message is printed out, 100 is returned, and the calling of function “foo” is skipped.

```
after(int k) : execution ( void foo (int ) ) && args(k) {
        printf(“after execution foo, argument = %d\n”,k);
}
```

This “after” advice takes a parameter which exposes the argument value of function foo to the advice function.

5.2 Proceed()

Around advice can be used to skip code at an existing join point. However, sometimes developers still want to access the original join points inside the advice. This can be achieved by using `proceed()` inside the around advice. The `proceed()` call takes the original value of the arguments[4] and calls/executes the original function.

For example:

```
int around() : call ( int foo (int) ) {
        printf("around call foo\n");
        printf("value of foo = %d\n", proceed());
        return 0;
}
```

This shows that function foo() is accessed inside an around advice, and its return value is used by the advice.

5.3 Preturn()

When a join point is matched by pointcuts, the advices associated with the pointcuts are invoked. After the advice functions finish, the control flow of the function containing the join point, which is called parent function, continues to execute. Sometimes, developers want to exit the parent function immediately after the advice functions are invoked. This can be achieved by using `preturn()` inside the advice. `preturn()` allows an immediate return from the parent function of a pointcut-matched join point. It is used to skip the rest of the code inside the parent function. Its syntax is `preturn(integral-type-expression)`.

For example, suppose "t1.acc" contains the following advice,

```
void * around(): call ( void * malloc ($) ) {
        void * ret = proceed();
        if( ret == 0 ) {
                printf("malloc calls fails\n");
                preturn(1);
        }
        return ret;
}
```

[4]Note, the developer can use the `args()` construct to change the value of the original argument. If `proceed()` is called afterward, the new value is used to call/execute the original function.

and the core file is as follows:

```
int foo(int a) {
        char * p;
        p = (char *)malloc(1000);
        printf(" after malloc \n");
}
```

Whenever a `malloc()` function fails, i.e., its return value is 0, the advice function will emit a message and immediately return from function `foo`[5], which is the parent function of this `malloc()` call. Therefore, the `printf` statement in function `foo` will not be executed.

5.4 "this": Reflective Information at Join Points

Inside an advice , ACC provides a special pointer variable, `this`, to access reflective information about the current join point.[6] The following fields can be accessed by `this`:

1. arg(<integer-value>): a "void *" pointer pointing to the address of the memory holding the <integer-value>-th parameter.
2. argsCount: the number of parameters.
3. argType(<integer-value>): the type name of the <integer-value>-th parameter.
4. fileName: the name of the source file containing the join point.
5. funcName: the name of the function calling the join point.
6. kind: the join point kind, "call", "execution", "set" or "get".
7. retType: the return type name.
8. targetName: the callee function name of a call or execution join point.[7], or the variable name of a set or a get join point.

For example, a generic tracing aspect could be written as:

[5]When the parent function requires a return value, the value specified in `preturn()` is used to as the return value of the parent function, otherwise, the value is discarded. The value specified in `preturn()` should have an integral type, like int, char, or pointer type. In the future, ACC might support other types, such as float or double.

[6]`this` is similar to the `thisJoinPoint` variable in AspectJ.

[7]For an "execution" pointcut, the "targetName" is the same as "funcName".

```
before(): call($ $(...)) {
        printf("%s \"%s\" in function %s \n", this→kind, this→targetName, this→funcName);
        if ( this→argsCount == 0 ) printf("no parameter \n");
        else {
                for(int i = 1 ; i <= this→argsCount; i++) {
                    printf("arg[%d] = %s ", i, this→argType(i));
                    if(strcmp(this→argType(i), "int") == 0) {
                        printf(", value = %d ", *(int *)(this→arg(i)));
                    } else if(strcmp(this→argType(i), "double") == 0) {
                        printf(", value = %.2f ", *(double *)(this→arg(i)));
                    }
                    printf("\n");
                }
        }
        printf("return type = %s \n \n", this→retType);
}
```

When the advice is applied to the following C code:

```
char * foo(int a) {
        return "just a test ";
}
void foo2(int a, double b) {
        foo(3);
}
void foo3() {
        foo2(5, 2.2);
}
int main() {
        foo3();
}
```

The output is:

```
call "foo3" in function main
"foo3" parameter type:
no parameter
return type = void

call "foo2" in function foo3
"foo2" parameter type:
arg[1] = int , value = 5
arg[2] = double , value = 2.20
return type = void

call "foo" in function foo2
"foo" parameter type:
arg[1] = int , value = 3
return type = char*
```

5.5 Multiple Advice

When a join point is matched by pointcuts from multiple advices, the various types of advices are handled differently.

5.5.1 before/after advices

Advices are executed sequentially according to the matching sequence.

For example:

```
/* advice 1 */
before() : execution ( void foo (int ) ) {
        printf("before advice 1");
}

/* advice 2 */
before() : execution ( void foo (int )) {
        printf("before advice 2");
}
```

Since the execution join point of function foo is matched by both advice 1 & 2 and both are before advices, the two advices are executed in sequence. That is advice 1 is executed before advice 2.

5.5.2 around advices

- no proceed(): the first matched advice is executed, and the rest are skipped.
- has proceed(): the proceed() call inside the around advice invokes the next matched around advice if there is one; otherwise, the proceed() call invokes the original function.

For example:

```
/* advice 3 */
void around() : execution ( void foo (int) ) {
        printf("around advice 3");
}

/* advice 4 */
void around() : execution ( void foo (int)) {
        printf("around advice 4");
}
```

The execution join point of function foo is matched by both around advices 3 & 4. Since there is no proceed() inside the advices, the first matched advice, advice 3, is executed, and advice 4 is skipped.[8]

The situation changes, if a proceed() call is present in the advice. For example:

```
/* advice 5 */
void around() : execution ( void foo (int) ) {
        printf("around advice 5");
        proceed() ;
}

/* advice 6 */
void around() : execution ( void foo (int)) {
        printf("around advice 6");
        proceed() ;
}
```

[8]Even if there is a proceed() call inside advice 4, it is not executed, since the execution join point is already surrounded by advice 3 without proceed (i.e., defined as "around" advice of advice 3 that does not let the call proceed to either further advice or the surrounded code).

Since there is a proceed() call used inside the advices, the execution sequence is: advice 5 $\rightarrow$ advice 6 $\rightarrow$ foo.[9]

By using multiple around advice and proceed(), the developer can impose different advices for join points. This can achieve effects similar to multiple if–statements, like:

```
/* advice 7 */
void around(int x) : execution ( void foo (int)) && args(x) {
        if(x < 3) {
             printf("around advice 7");
             return;
        }else {
             proceed();
        }
}

/* advice 8 */
void around(int x) : execution ( void foo (int)) && args(x) {
        if(x < 9) {
             printf("around advice 8");
             return;
        }else {
             proceed();
        }
}

/* advice 9 */
void around(int x) : execution ( void foo (int)) && args(x) {
        if(x < 20) {
             printf("around advice 9");
             return;
        }else {
             proceed();
        }
}
```

[9] If there is no proceed() in advice 6, the original function foo() will not be executed.

The effects of applying advice 7, 8, & 9 is same as: whenever calling a function “foo” with parameter “x”,

```
if(x < 3) {
    printf(“around advice 7”);
}else if(x < 9) {
    printf(“around advice 8”);
}else if(x < 20) {
    printf(“around advice 9”);
}else {
    foo(x);
}
```

6 Static Crosscutting

In addition to expressing dynamic crosscutting represented by call/execution and set/get join points, ACC also provides mechanism to support static crosscutting, such as the addition of members to structs and unions.

6.1 intype() Pointcut

An intype pointcut picks out the struct or union type whose name matches the type name specified or which has been typedefed by a name matching the one specified. Its syntax is “`intype(`*identifier*`)`”.

For example, if the types are declared as follows:

```
struct X {              <— first struct
        int a;
};
typedef struct X MYX1;
typedef MYX1 MYX2;
typedef struct {        <— second struct
        int b;
} MYX3;
typedef MYX3 MYX4;
```

then the `intype()` pointcut has the following effects:

1. "`intype(struct X)`", "`intype(MYX1)`", and "`intype(MYX2)`" match the first struct.
2. "`intype(MYX3)`" and "`intype(MYX4)`" match the second struct.
3. "`intype(MYX$)`" matches both structs.

Semantics:

1. The *identifier* must be a struct or union type name, or a name assigned by a typedef for a struct or a union.
2. The wildcard character $ can be used inside the *identifier*.
3. The `intype()` pointcut must only be used within an `introduce()` advice.

6.2 introduce() Advice

An `introduce()` advice adds new data members to the struct or union type picked out by the `intype()` pointcut. Its syntax is:

introduce (): *pointcuts* { *function-body* }

Semantics:

1. The *pointcuts* must contain only `intype()` pointcuts, or contain composite or named pointcuts built from `intype()` pointcuts.
2. The *function-body* must contain valid struct or union member declarations.
3. The data member name declared inside the *function-body* must not collide with the existing member names in the matched type.
4. ACC simply copies the *function-body* and adds them to the end of the matched struct or union declaration.
5. If multiple `introduce()` advices are applied to the same type, the data members from each advice are added according to the matching sequence.

For example, for the types described above and the following advice declarations:

```
introduce() : intype(struct X) {          <— advice 1
        double b;
}
introduce() : intype(MYX1) {              <— advice 2
```

```
        double c;
    }
    introduce() : intype(MYX3) {                <— advice 3
        double c;
    }
    introduce(): intype(MYX3) || intype(MYX1) {           <— advice 4
        char * p;
    }
```

After the advices are applied, the types become:

```
    struct X {
        int a;
        double b;              <— from advice 1
        double c;              <— from advice 2
        char * p;              <— from advice 4
    };
    . . .
    typedef struct {
        int b;
        double c;              <— from advice 3
        char * p;              <— from advice 4
    } MYX3;
    . . .
```

7 Exception Handling

In addition to support dynamic and static crosscutting, ACC also provides mechanisms to add exception handling to C programs.[10]

[10]Currently, an "exception" in ACC is represented by any non-zero integer value. In the future, ACC might allow the developer to specify user-defined exceptions.

7.1 try() Pointcut

ACC provides the `try()` pointcut to set an exception handler for the exceptions thrown in the control flow of some join points. Its syntax is `try( pointcut-definition )`.

For example, "`try(execution(void foo3()))`" captures all exceptions occuring under the control flow of executing the function `foo3`.

Given the following advice:

```
catch(int e): try(execution(void foo3())) {
        printf("catch an exception = %d\n", e);
}

before(): call(void foo()) {
        printf("throw an exception before calling foo\n");
        throw(34);
}
```

If the advice is applied to the following C code:

```
void foo(int a) {
        printf("in foo\n\n");
}
void foo2() {
        printf("in foo2\n");
        foo();
}
void foo3() {
        foo2();
}
int main() {
        printf("call foo3 in main\n");
        foo3();
}
```

The output is:

```
call foo3 in main

in foo2

throw an exception before calling foo

catch an exception = 34
```

Semantics:

1. The pointcut definition inside `try()` can be any valid pointcut definition except another `try()` pointcut.
2. Only `catch()` advices can be specified for the `try()` pointcut.

7.2 catch() Advice

A `catch()` advice is invoked when an exception is captured by a `try()` pointcut. Its syntax is as follows:

catch (int *para-name*): *pointcuts* { *function-body* }

Semantics:

1. The *pointcuts* must contain only `try()` pointcuts, or contain composite or named pointcuts built from `try()` pointcuts.
2. There must be only one parameter, which is of "int" type. It stores the captured exception value, and can be used inside the advice function body. The parameter name should not be used by an *args()* pointcut in the *pointcuts*.
3. If multiple `catch()` advices are applied to the same `try()` pointcut, the first `catch()` advices take precedence. If there is another exception thrown from the first `catch()` advice, the second `catch()` advice takes effect, and so on.

In the above example, the `catch()` advice is replaced by the following three `catch()` advices:

```
catch(int e): try(execution(void foo3())) {
        printf("1st catch, catch an exception = %d\n", e);
        throw(35);
}
catch(int e): try(execution(void foo3())) {
        printf("2nd catch, catch an exception = %d\n", e);
        throw(36);
}
catch(int e): try(execution(void foo3())) {
        printf("3rd catch, catch an exception = %d\n", e);
}
```

The output of running the C code is:

```
call foo3 in main
in foo2
throw an exception before calling foo
1st catch, catch an exception = 34
2nd catch, catch an exception = 35
3rd catch, catch an exception = 36
```

7.3 throw()

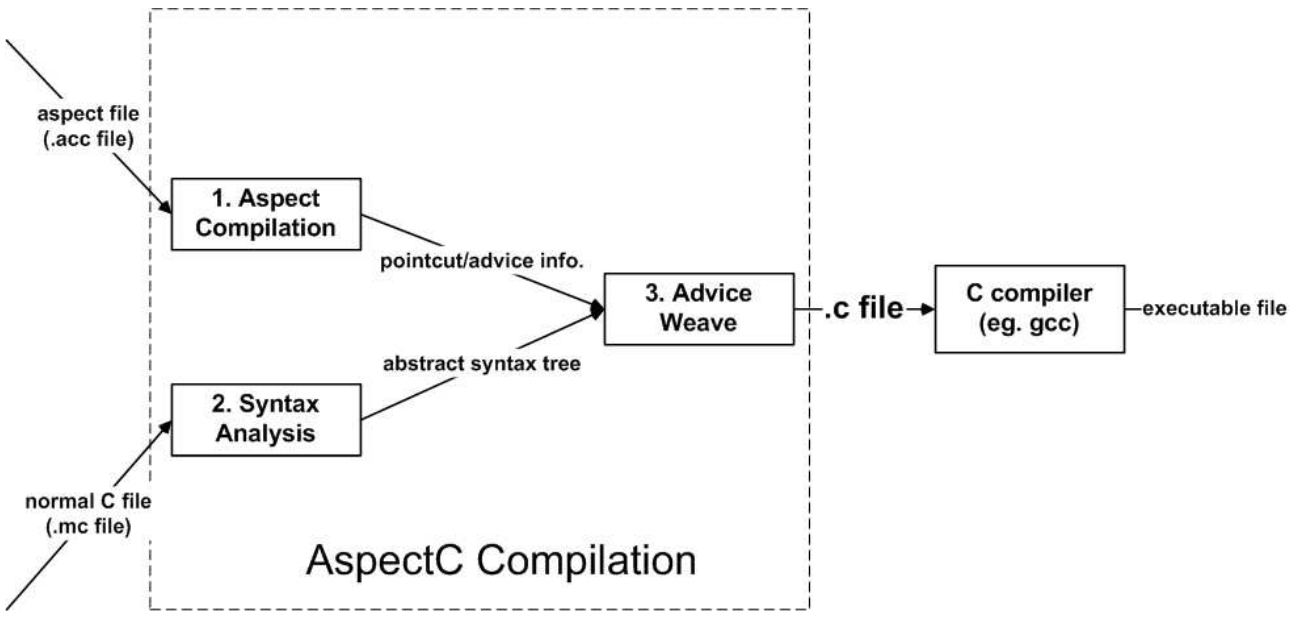
In an advice function, ACC uses `throw()` to throw an exception. The syntax is as follows:
`throw(non-zero-integer-value)`.

8 Implementation

ACC is implemented as a source-to-source translator. The inputs are ACC files and C source files[11]. The aspect files contain pointcut, advice, or normal C code. The outputs are normal C files with advice code inserted at the point specified by pointcuts. The output files can then be compiled by a C compiler.

There are 3 phases in the ACC compilation process: aspect compilation, syntax analysis and advice weaving. The compilation process is described by the following figure.

[11]Both kinds of input files need to be pre-processed by a C pre-processor or by gcc using the "-E" option.

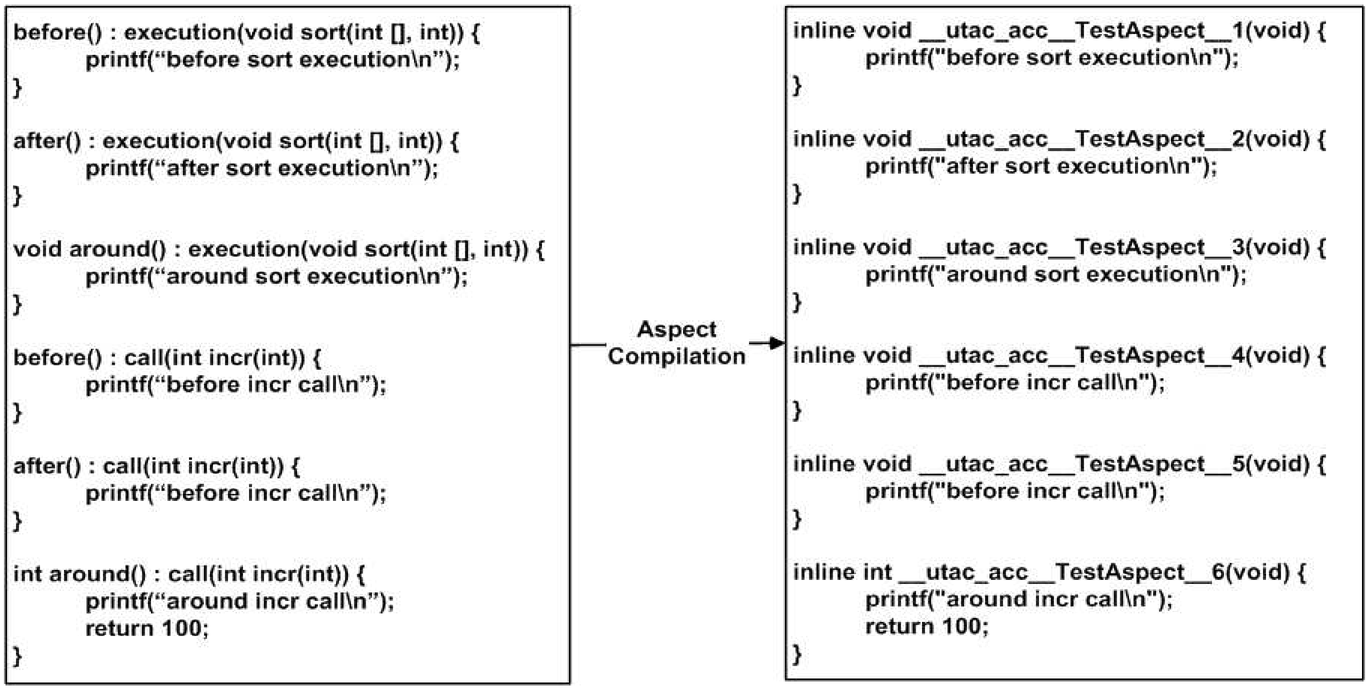


8.1 Aspect Compilation

In the aspect compilation phase, each advice is compiled to a unique C function. The advice parameters are compiled to parameters of the new C function. In the advice weaving phase, these parameters are bound to function arguments. Since before advice and after advice have no return type, the ACC compiler uses “void” as return type for the corresponding functions. For around advice, the ACC compiler uses the return type specified in the advice declaration as the return type of the function.

Another task in this phase it to collect information related to pointcut and advice, which is used in the advice weaving phase.

The following figure illustrates the kind of C functions generated from the advice in the aspect file.



```
before() : execution(void sort(int [], int)) {
        printf(“before sort execution\n”);
}

after() : execution(void sort(int [], int)) {
        printf(“after sort execution\n”);
}

void around() : execution(void sort(int [], int)) {
        printf(“around sort execution\n”);
}

before() : call(int incr(int)) {
        printf(“before incr call\n”);
}

after() : call(int incr(int)) {
        printf(“before incr call\n”);
}

int around() : call(int incr(int)) {
        printf(“around incr call\n”);
        return 100;
}
```

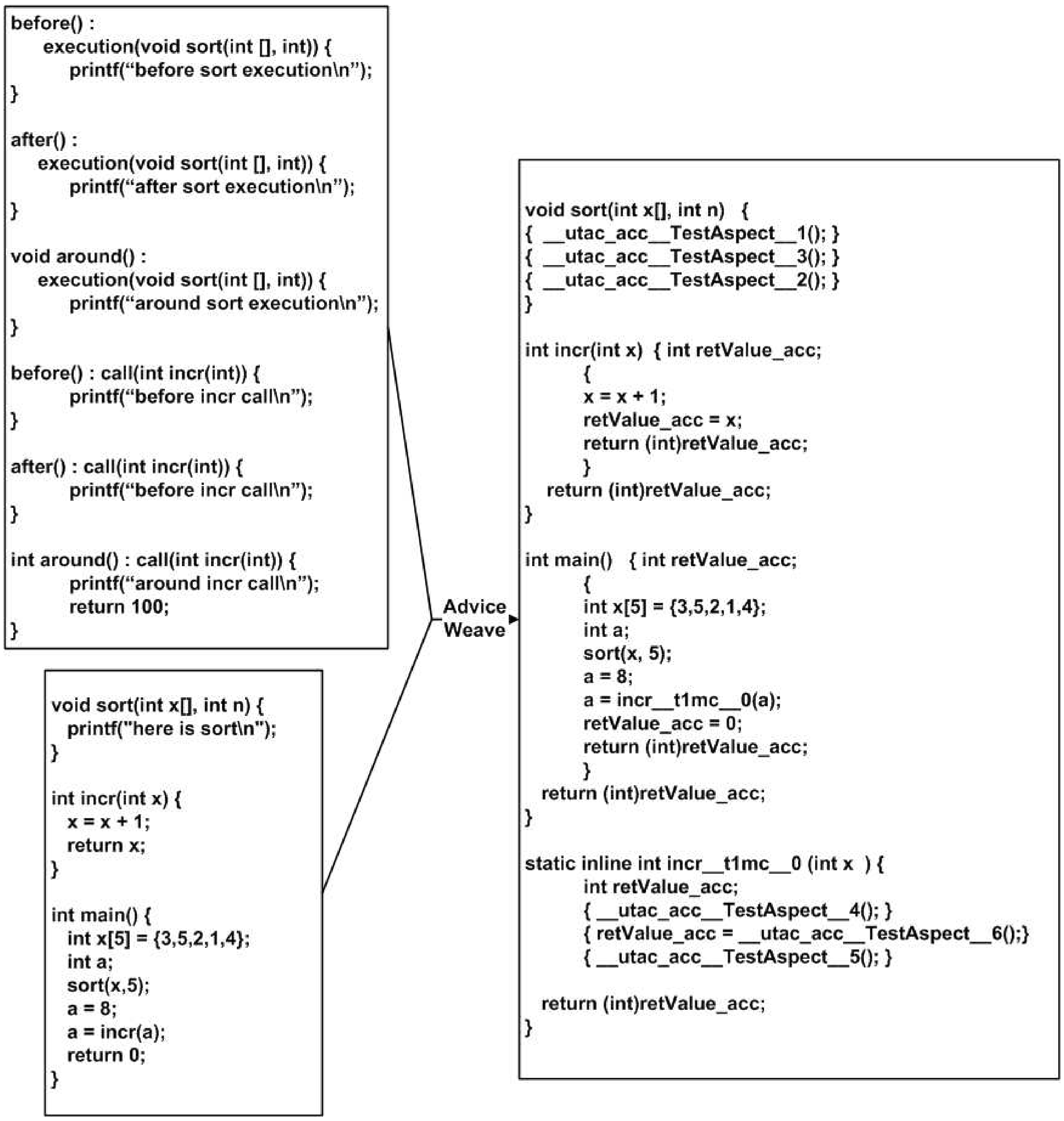
```
inline void __utac_acc__TestAspect__1(void) {
        printf("before sort execution\n");
}

inline void __utac_acc__TestAspect__2(void) {
        printf("after sort execution\n");
}

inline void __utac_acc__TestAspect__3(void) {
        printf("around sort execution\n");
}

inline void __utac_acc__TestAspect__4(void) {
        printf("before incr call\n");
}

inline void __utac_acc__TestAspect__5(void) {
        printf("before incr call\n");
}

inline int __utac_acc__TestAspect__6(void) {
        printf("around incr call\n");
        return 100;
}
```

8.2 Syntax Analysis

The main purpose of this phase is to collect information to facilitate join point matching by generating an abstract syntax tree (AST) for the C sources.

8.3 Advice Weaving

The last phase is to insert calls to advice functions in appropriate locations in the C sources. The following figure illustrates how calls are inserted into a C file.



```
before() :
    execution(void sort(int [], int)) {
        printf("before sort execution\n");
}

after() :
    execution(void sort(int [], int)) {
        printf("after sort execution\n");
}

void around() :
    execution(void sort(int [], int)) {
        printf("around sort execution\n");
}

before() : call(int incr(int)) {
        printf("before incr call\n");
}

after() : call(int incr(int)) {
        printf("before incr call\n");
}

int around() : call(int incr(int)) {
        printf("around incr call\n");
        return 100;
}
```

```
void sort(int x[], int n) {
  printf("here is sort\n");
}

int incr(int x) {
  x = x + 1;
  return x;
}

int main() {
  int x[5] = {3,5,2,1,4};
  int a;
  sort(x,5);
  a = 8;
  a = incr(a);
  return 0;
}
```

```
void sort(int x[], int n)   {
{  __utac_acc__TestAspect__1(); }
{  __utac_acc__TestAspect__3(); }
{  __utac_acc__TestAspect__2(); }
}

int incr(int x)  { int retValue_acc;
        {
        x = x + 1;
        retValue_acc = x;
        return (int)retValue_acc;
        }
   return (int)retValue_acc;
}

int main()   { int retValue_acc;
        {
        int x[5] = {3,5,2,1,4};
        int a;
        sort(x, 5);
        a = 8;
        a = incr__t1mc__0(a);
        retValue_acc = 0;
        return (int)retValue_acc;
        }
  return (int)retValue_acc;
}

static inline int incr__t1mc__0 (int x  ) {
        int retValue_acc;
        { __utac_acc__TestAspect__4(); }
        { retValue_acc = __utac_acc__TestAspect__6();}
        { __utac_acc__TestAspect__5(); }

  return (int)retValue_acc;
}
```

9 Grammar

In order for ACC to support aspect-oriented language extensions, the following keywords and grammar rules are added to the C language grammar.

9.1 Keywords

New keywords are : “args”, “after”, “around”, “before”, “call”, “callp”, “catch”, “cflow”, “execution”, “get”, “infile”, “infunc”, “introduce”, “intype”, “pointcut”, “preturn”, “proceed”, “result”, “set”, “throw”, “try”.

9.2 Grammar Rules

function-definition:

declaration-specifiers$_{opt}$ *declarator* **:** *pointcuts compound-statement*

declaration:

pointcut *declarator* **:** *pointcuts* **;**

pointcuts:

or-pointcuts

pointcuts && *or-pointcuts*

or-pointcuts:

unary-pointcut

or-pointcuts || *unary-pointcut*

unary-pointcut:

base-pointcut

! *base-pointcut*

base-pointcut:

args (*type-or-id-list* **)**

call (*func-jointpoint* **)**

cflow (*pointcuts* **)**

execution (*func-jointpoint* **)**

get (*declaration-specifiers declarator* **)**

identifier (*identifier-list*$_{opt}$)

infile (*string-literal*)

infunc (*identifier*)

intype (*type-name*)

result (*type-or-id*)

set (*declaration-specifiers declarator*)

try (*pointcuts*)

func-jointpoint:

declaration-specifiers declarator

type-or-id-list:

type-or-id

type-or-id-list type-or-id

type-or-id:

type-name

identifier

direct-declarator:

before

after

around

introduce

catch

10 Usage

10.1 General Usage

The ACC compiler takes C source files with and without ACC syntax as input. The files with ACC syntax should have the suffix ".acc" and those without AspectC syntax should have suffix ".mc". Furthermore, both types of files should be pre-processed by a C preprocessor before passing through the ACC compiler.[12]

[12]The file suffix could be changed by -af and -mf options.

The ACC compiler outputs ANSI-C compliant C source files to be processed by a C compiler. If any input file has an unknown suffix, the ACC compiler emits an error message and stops compilation.

Example 1:

Suppose there are neither #include nor macro directives used in the `a.acc` or the `b.mc` files:

```
>acc a.acc b.mc
```

The ACC compiler generates `a.c` and `b.c` C–source files for processing by a C compiler, like for example gcc.

```
>gcc a.c b.c
>./a.out
```

Example 2:

Suppose there are #include or macro directives used in the `a.acc` or the `b.mc` files. This requires that the source files have to be pre-processed before weaving and compilation.

1. since gcc does not recognize the `.acc` or the `.mc` suffix, the suffix must to be changed to `.c`.

```
>cp a.acc a_acc.c
>cp b.mc b_mc.c
```

2. pre-process the files by a pre-processor, and save the output in files with the by the ACC compiler required suffixes

```
>gcc -E a_acc.c > a_acc.acc
>gcc -E b_mc.c > b_mc.mc
```

3. perform the ACC compilation

```
>acc a_acc.acc b_mc.mc
```

4. the ACC compiler generates a_acc.c and b_mc.c C–source files for processing by a C compiler, like for example gcc.

```
>gcc a_acc.c b_mc.c
>./a.out
```

Makefile examples capturing the above steps are part of the ACC distribution.

10.2 Use “tacc”

In order to make it easy to use the ACC compiler (“acc”) and integrate aspect compilation into an existing building system of a C project, the ACC distribution also provides “tacc”. Users are encouraged to and should use “tacc” directly, instead of “acc”, because “tacc” will automatically conduct the steps of preprocessing, weaving[13] and compilation.

User can invoke “tacc” in the same was as invoking a C compiler, like gcc. For example, suppose there is an aspect file (`a.acc`) and a normal C file (`b.c`), using “tacc” compilation works as follows:

```
>tacc a.acc b.c
>./a.out
```

For details of “tacc”, please refer to `www.aspeCtc.net`.

10.3 Command Line Options

The following command line options are supported by the ACC compiler:

1. -a

   –aspectmatch

   The advices will also match join points inside aspect files.

2. -af=<file suffix> ,

   –aspect-suffix=<file suffix>

   Specifies the file suffix for the aspect file.

3. -h

   –help

   Display help information.

4. -m[=<file name>]

   –matchinfo[=<file name>]

   The join point-advice matching information is output.

5. -mf=<file suffix>

   –mainfile-suffix=<file suffix>

   Specifies the file suffix for the non-aspect file.

[13] “tacc” weaves aspects according to a set of rules, which are available on `www.aspeCtc.net`.

6. -n

   –no-line

   No #line directives are generated in output.

7. -t

   –thread-safe

   The code generated to support the `cflow()` pointcut is thread-safe.[14]

8. -v

   –version

   The compiler's version number is printed.

References

[1] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE*, 2001.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[3] AspectJ Team. AspectJ project web site. `http://www.eclipse.org/aspectj/`.

[14]The ACC compiler uses a GCC specific feature, – thread-local storage, – to ensure thread-safety for the generated code. The "__thread" keyword is used. However, since this is not a standard feature and not supported by all C compilers, ACC turns the option off by default. The default code generated for cflow pointcuts is not thread-safe. For more information about the thread-local storage feature, visit `gcc.gnu.org/`.