

AspectC Language Specification

Version 0.2 *

Weigang Gong and Hans-Arno Jacobsen

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

September 8, 2006

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

*This research is supported by NSERC.

Contents

1	Overview	4
2	Change Log	4
3	Join Point Model	5
4	Pointcut	5
4.1	Primitive Pointcut	5
4.1.1	Primitive Pointcuts on Pure Join Points	6
4.1.2	Primitive Pointcut on Mixed Join Points	6
4.2	Composite Pointcut	7
4.3	Named Pointcut	8
4.4	cflow() Pointcut	9
4.5	Matching Mechanism	10
4.5.1	Simple Character Matching	11
4.5.2	Wildcard Character Matching	11
5	Advice	12
5.1	Single Advice	12
5.2	Proceed()	13
5.3	“this”: Reflective Information at Join Points	14
5.4	Multiple Advice	16
5.4.1	before/after advices	16
5.4.2	around advices	16
6	Implementation	19
6.1	Aspect Compilation	19
6.2	Syntax Analysis	20
6.3	Advice Weaving	20
7	Grammar	22
7.1	Keywords	22
7.2	Grammar Rules	22

1 Overview

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

This specification introduces the ASPECTC programming model and the new language constructs.

The specification is implemented by the ASPECTC compiler that weaves code written in ASPECTC into ASPECTC-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 ASPECTC language design adapts the ideas of aspect-oriented programming laid-out in the original paper by Kiczales *et al.* [1] to the C programming language and loosely follows the ASPECTJ programming language design [2].

To this end ASPECTC 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 ASPECTC implementation exists.

Long-term research objectives of the AspectC 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 ASPECTC implementation and further information can be found on the project Web pages at www.AspectC.net.

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

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

2 Change Log

New features in ASPECTC Version 0.2, relative to ASPECTC Version 0.1:

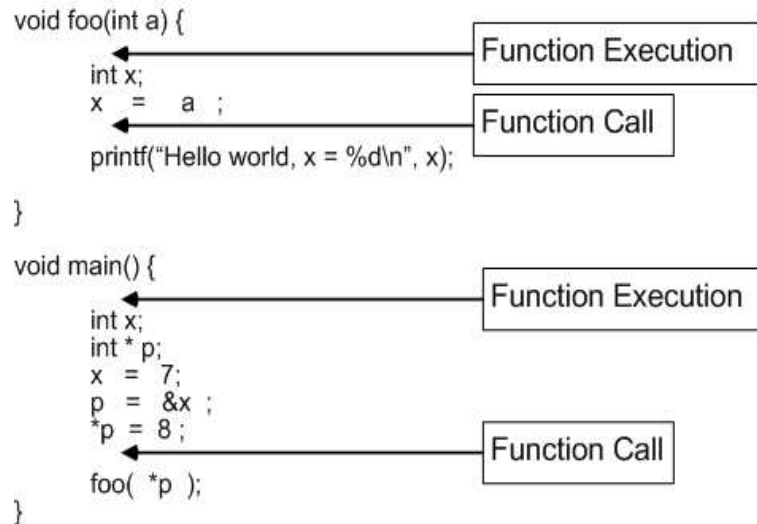
- Section 4.4: `cflow()` support.
- Section 5.3: `this` reflective join point information support.
- Section 5.4.2: multiple around advice with `proceed()`.

3 Join Point Model

A *join point* is a well-defined point in the execution context of a program. Currently, ASPECTC 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

Both join points are illustrated by the program below:



4 Pointcut

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

4.1 Primitive Pointcut

We distinguish between pure join points and mixed join points. A pure join point is one of the join points defined previously. A mixed join point refers to either one of the two join points defined previously.

4.1.1 Primitive Pointcuts on Pure Join Points

1. `call(function-signature)`

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

2. `execution(function-signature)`

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

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 each parameter in the prototype, only its type should be specified.

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

4.1.2 Primitive Pointcut on Mixed Join Points

1. `args(a list of types or identifiers)`

An `args` pointcut picks out call or execution join points whose parameters’ types match the specified types or the types of the specified identifiers.

2. `infile("file name")`

An `infile` pointcut picks out call or execution join points which appear in the file specified.

3. `infunc(identifier)`

An `infunc` pointcut picks out call or execution join points which appear in the function specified.

4. `result(type or identifier)`

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

Semantics:

1. 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 call or execution join point 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 );
}

```

2. There is a special format of args(): “args(* pointer-variable-name)”. The meaning is that the parameter type of a join point must match the type after dereferencing the pointer variable.¹ 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 c, int n ) ) && args(char , *x) {
    *x = (*x) * 2;
    printf(“inside before advice, argument value is doubled\n”);
}

```

3. The file name specified in the infile pointcut must be enclosed by quotes, and it should be the name of a generated file, not the input 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.c”)”, because “t1mc.c” is the name of the generated file.²

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

4.2 Composite Pointcut

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

1. pointcut₀ && pointcut₁: returns join points picked up by both pointcut₀ and pointcut₁
2. pointcut₀ || pointcut₁: returns join points picked up by either pointcut₀ or pointcut₁
3. !pointcut₀: returns join points not picked up by pointcut₀
4. (pointcut₀): returns join points picked up by pointcut₀

Semantics:

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

¹“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”.

²In future releases we plan to relax this point.

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-listopt )
```

For example, the following example shows how to declare a named pointcut and 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

ASPECTC 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

ASPECTC 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, ASPECTC 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 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

ASPECTC 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:

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 \$, ASPECTC searches an advice parameter having the exact same name. If found, the name is bound with the advice parameter, otherwise, ASPECTC 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, ASPECTC 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-specifieropt before|after|around ( parameter-type-listopt ) : pointcuts
{ function-body }
```

Semantics:

1. For before/after advice, the *type-specifier* should not be specified. The ASPECTC 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 ASPECTC 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("before execution foo, argument = %d\n",k);
}

```

This “before” 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 function call/execution inside the advice. This can be achieved by using `proceed()` inside the around advice. The `proceed()` call takes the original value of the arguments³ 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.

³Note, 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.

5.3 “this”: Reflective Information at Join Points

Within the advice code, ASPECTC provides a special pointer variable, `this`, to access reflective information about the current join point.⁴

The following string fields can be accessed by `this`:

1. `funcName`: the function name of the join point.
2. `kind`: the join point kind, either “call” or “execution”.

For example, the following advice prints out the name of every called/executed function in the control flow of `main`:

```
before(): cflow(execution(int main())) {
    printf("aspect: kind = %s, function = %s\n", this->kind, this->funcName);
}
```

When 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() {
    foo3();
}
```

The output is:

⁴`this` is similar to the `thisJoinPoint` variable in ASPECTJ.

aspect: kind = execution, function = main

aspect: kind = call, function = foo3

aspect: kind = execution, function = foo3

aspect: kind = call, function = foo2

aspect: kind = execution, function = foo2

in foo2

aspect: kind = call, function = foo

aspect: kind = execution, function = foo

in foo

5.4 Multiple Advice

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

5.4.1 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.

5.4.2 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.⁵

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() ;
}

```

Since there is a `proceed()` call used inside the advices, the execution sequence is: advice 5 → advice 6 → `foo`.⁶

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 {

```

⁵Even 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).

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

```

        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();
    }
}

```

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

```

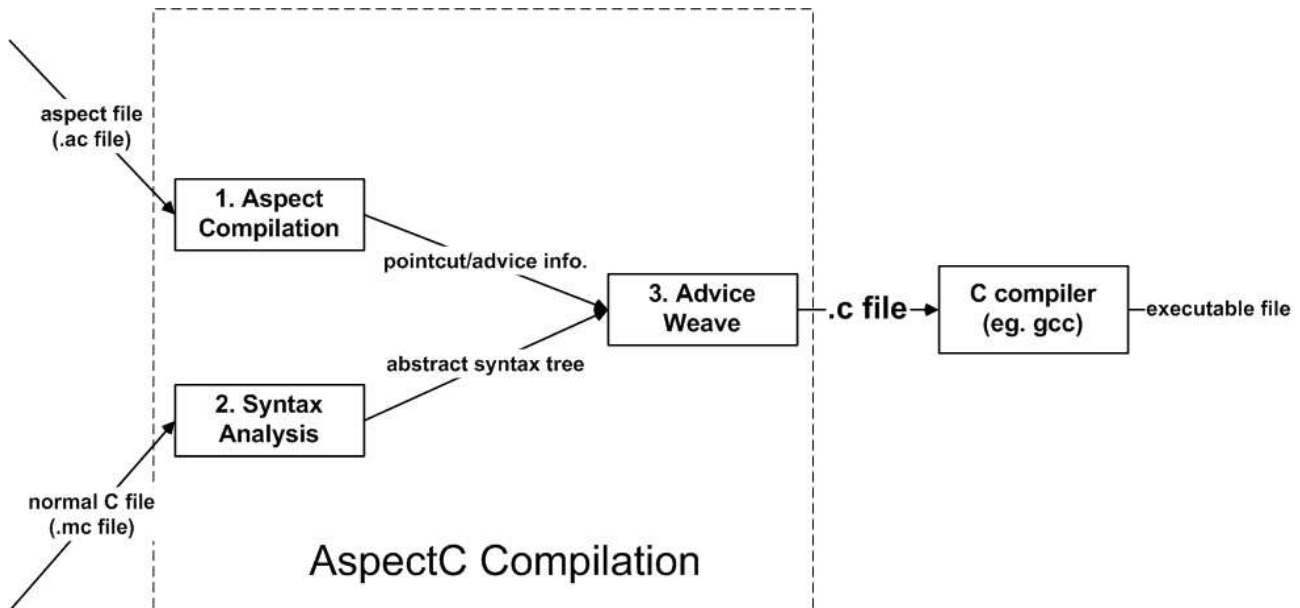
if(x < 3) {
    printf("around advice 7");
}else if(x < 9) {
    printf("around advice 8");
}else if(x < 20) {
    printf("around advice 9");
}else {
    foo(x);
}

```

6 Implementation

ASPECTC is implemented as a source-to-source translator. The inputs are ASPECTC files and C source files⁷. 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 ASPECTC compiler's compilation step: 1. aspect compilation; 2. syntax analysis; 3. advice weaving. The compilation process is described by the following figure.



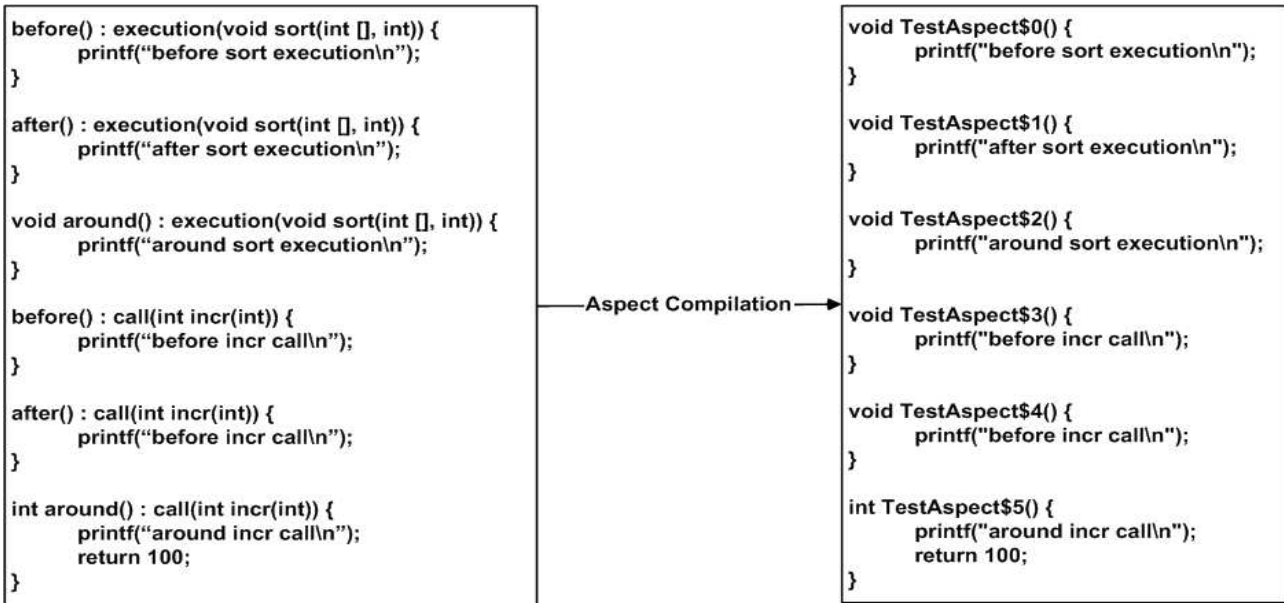
6.1 Aspect Compilation

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 ASPECTC compiler uses “void” as return type for the corresponding functions. For around advice, the ASPECTC compiler uses the return type specified in the advice declaration as the return type of the function.

Another task in this phase is 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.

⁷Both kinds of input files need to be pre-processed by a C pre-processor or by gcc using the “-E” option.



6.2 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.

6.3 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;
}

```

Advice
Weave →

```

void sort(int x[], int n) {
    { TestAspect$0(); }
    { TestAspect$2(); }
    { TestAspect$1(); }
    return ;
}
int incr(int x) {int $rtValue;
    (x=(x+1));
    $rtValue = x;
    return $rtValue;
}
int incr$SelectionSort$0 (int x ) ;
int main() {int $rtValue;
    int x[5] = {3,5,2,1,4};
    int a;
    sort(x, 5);
    (a=8);
    (a=incr$SelectionSort$0(a));
    $rtValue = 0;
    return $rtValue;
}
void TestAspect$3 () ;
void TestAspect$4 () ;
int TestAspect$5 () ;
int incr$SelectionSort$0 (int x ) {
    int $rtValue;
    { TestAspect$3(); }
    { $rtValue = TestAspect$5();}
    { TestAspect$4(); }
    return $rtValue;
}

```

7 Grammar

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

7.1 Keywords

New keywords are : “args”, “after”, “around”, “before”, “call”, “execution”, “infile”, “infunc”, “pointcut”, “proceed”, “cflow”, “result”.

7.2 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*)

execution (*func-jointpoint*)

infile (*string-literal*)

infunc (*identifier*)

result (*type-or-id*)

result (*type-or-id*)
identifier (*identifier-list_{opt}*)
cflow (*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

8 Usage

The ASPECTC compiler takes C source files with and without ASPECTC syntax as input. The files with ASPECTC syntax must have the suffix ".ac" and those without AspectC syntax must have suffix ".mc". Furthermore, both types of files must be pre-processed by a C preprocessor before passing through the ASPECTC compiler.⁸

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

For example:

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

The ASPECTC 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
```

⁸In the future we plan to relax the suffix restriction for C files.

References

- [1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.
- [2] AspectJ Team. AspectJ project web site. <http://www.eclipse.org/aspectj/>.