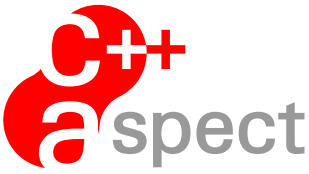

Documentation:
AspectC++ Language Reference

pure-systems GmbH

Matthias Urban

and Olaf Spinczyk

Version 2.0, July 7, 2016

[1]os@aspectc.org
www.aspectc.org

[2]aspectc@pure-systems.com
www.pure-systems.com
Agnetenstr. 14
39106 Magdeburg
Germany

Contents

1 About

This document is intended to be used as a reference book for the AspectC++ language elements. It describes in-depth the use and meaning of each element providing examples. For experienced users the contents of this document are summarized in the AspectC++ Quick Reference. Detailed information about the AspectC++ compiler **ac++** can be looked up in the AspectC++ Compiler Manual.

AspectC++ is an aspect-oriented extension to the C++ language[1]. It is similar to AspectJ[2] but, due to the nature of C++, in some points completely different. The first part of this document introduces the basic concepts of the AspectC++ language. The in-depth description of each language element is subject of the second part.

2 Basic Concepts

2.1 Pointcuts

Aspects in AspectC++ implement crosscutting concerns in a modular way. With this in mind the most important element of the AspectC++ language is the pointcut. Pointcuts describe a set of join points by determining on which condition an aspect shall take effect. Thereby each join point can either refer to a function, an attribute, a type, a variable, or a point from which a join point is accessed so that this condition can be for instance the event of reaching a designated code position or the allocation of a variable with a certain value. Depending on the kind of pointcuts, they are evaluated at compile time or at runtime.

2.1.1 Match Expressions

There are two types of pointcuts in AspectC++: *code pointcuts* and *name pointcuts*. Name pointcuts describe a set of (statically) known program entities like types, attributes, functions, variables, or namespaces. All name pointcuts are based on match expressions. A match expression can be understood as a search pattern. In such a search pattern the special character "%" is interpreted as a wildcard for names or parts of a signature. The special character sequence "..." matches any number of parameters in a function signature or any number of scopes in a qualified name. A match expression is a quoted string.

[1] defined in the ISO/IEC 14882:1998(E) standard
[2] http://www.eclipse.org/aspectj/

Example: match expressions (name pointcuts)

`"int C::%(...)"`
: matches all member functions of the class C that return an `int`

`"%List"`
: matches any namespace, class, struct, union, or enum whose name ends with `List`. In case of a matched namespace or class the match expression also matches entities inside the namespace resp. class. For more information see section 3.2.

`"% printf(const char *, ...)"`
: matches the function `printf` (defined in the global scope) having at least one parameter of type `const char *` and returning any type

`"const %& ...::%(...)"`
: matches all functions that return a reference to a constant object

Match expressions select program entities with respect to their definition scope, their type, and their name. A detailed description of the match expression semantics follows in section 3 on page 17. The grammar which defines syntactically valid match expressions is shown in appendix B on page 48.

2.1.2 Pointcut Expressions

The other type of pointcuts, the code pointcuts, describe an intersection through the set of the points in the control flow of a program. A code pointcut can refer to a call or execution point of a function, to a call of a built-in operator or and to write and read points of attributes and variables. They can only be created with the help of name pointcuts because all join points supported by AspectC++ require at least one name to be defined. This is done by calling predefined pointcut functions in a pointcut expression that expect a pointcut as argument. Such a pointcut function is for instance **within**(*pointcut*), which filters all join points that are within the functions or classes in the given pointcut.

Name and code pointcuts can be combined in pointcut expressions by using the algebraic operators “&&”, “||”, and “!”.

Example: pointcut expressions

`"%List" && !derived("Queue")`
: describes the set of classes with names that end with “List” and that are not derived from the class `Queue`

`call("void draw()") && within("Shape")`
: describes the set of calls to the function `draw` that are within methods of the class `Shape`

2.1.3 Types of Join Points

According to the two types of pointcuts supported by AspectC++ there are also two coarse types of join points: name join points and code join points. As diagramed in figure 1 both of these have sub join point types. The types `Any`, `Name`, `Code` and `Access` are abstract types and exist just for categorizing the other join point types.

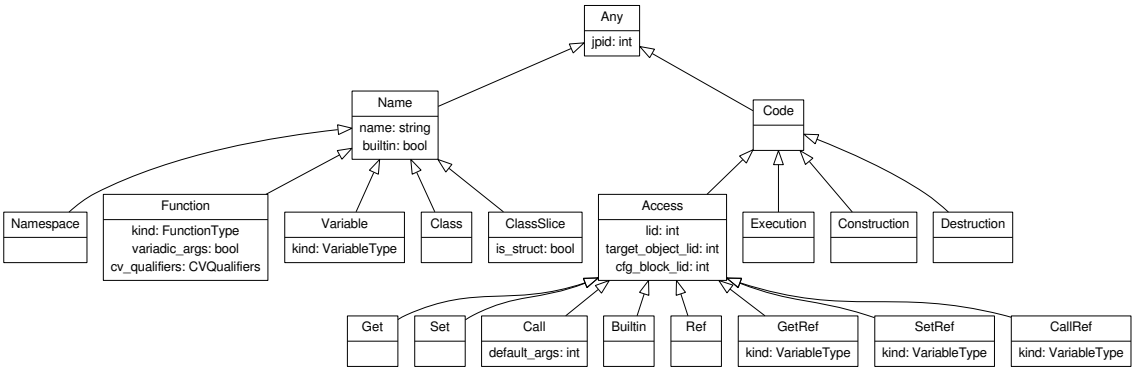

Figure 1: join point type hierarchy

Figure 1 is extracted from the AspectC++ project repository hierarchy, that can be found in appendix C.

Based on a short code fragment the differences and relations between the types of join points shall be clarified.

```
1  class Shape { /*...*/ };
2  void draw(Shape& shape) { /*...*/ }
3
4  namespace Circle {
5    typedef int PRECISION;
6
7    class S_Circle : public Shape {
8      PRECISION m_radius;
9      public:
10       void radius(PRECISION r) {
11         m_radius = r;
```

```
12        }
13        ~S_Circle() { /*...*/ }
14    };
15
16    void draw(PRECISION r) {
17      S_Circle circle;
18      circle.radius(r);
19      draw(circle);
20    }
21  }
22
23  int main() {
24    Circle::draw(10);
25    return 0;
26  }
```

Code join points are used to form code pointcuts and name join points (i.e. names) are used to form name pointcuts. Figure 2 shows join points of the code fragment above and how they correlate. Built-in constructors, destructors and uncalled operators are not shown. Additionally appendix D shows the contents of the project repository[3] for the code fragment.

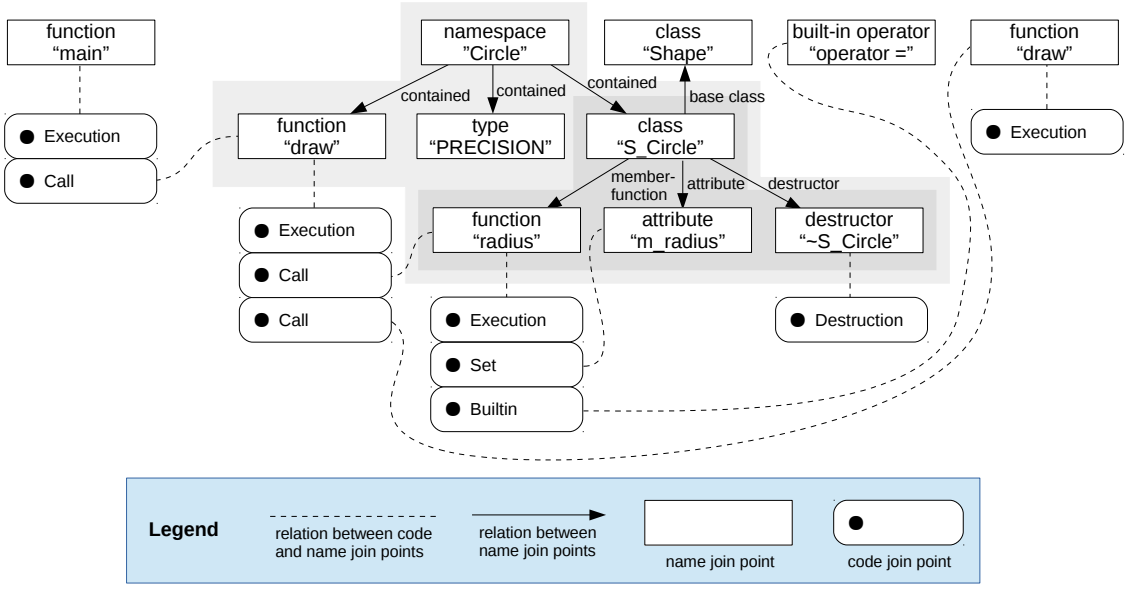

Figure 2: join points and their relations

[3]The AspectC++ project repository is a file, that contains the internal AspectC++ model as xml-tree. The actual style and format of the content may change at any time. For more information see the AspectC++ Compiler Manual.

Every **execution** join point is associated with the name of an executable function. Pure virtual functions are not executable. Thus, advice code for execution join points would never be triggered for this kind of function. However, the call of such a function, i.e. a **call** join point with this function as target, is absolutely possible. Furthermore there are no execution join points for built-in operator functions.

Every **call** or **builtin** join point is associated with two names: the name of the source and the target function (in case of builtin this is the global built-in operator function) of a function call. As there can be multiple function calls within the same function, each function name can be associated with a list of **call** join points and **builtin** join points. The same holds for **set** and **get** join points, which represent write resp. read operations on data members or global variables. Each of these join points is associated with the name of the function that contains the join point and the name of the accessed attribute or global variable. A **construction** join point means the class specific instruction sequence executed when an instance is created. In analogy, a **destruction** join point means the object destruction.

2.1.4 Pointcut declarations

AspectC++ provides the possibility to name pointcut expressions with the help of pointcut declarations. This makes it possible to reuse pointcut expressions in different parts of a program. They are allowed where C++ declarations are allowed. Thereby the usual C++ name lookup and inheritance rules are also applicable for pointcut declarations.

A pointcut declaration is introduced by the keyword `pointcut`.

Example: pointcut declaration

```
pointcut lists() = derived("List");
```

`lists` can now be used everywhere in a program where a pointcut expression can be used to refer to `derived("List")`

Furthermore pointcut declarations can be used to define pure virtual pointcuts. This enables the possibility of having re-usable abstract aspects that are discussed in section 2.4. The syntax of pure virtual pointcut declarations is the same as for usual pointcut declarations except the keyword `virtual` following `pointcut` and that the pointcut expression is “0”.

Example: pure virtual pointcut declaration

```
pointcut virtual methods() = 0;
```

`methods` is a pure virtual pointcut that has to be redefined in a derived aspect to refer to the actual pointcut expression

2.2 Slices

A *slice* is a fragment of a C++ language element that defines a scope. It can be used by advice to extend the static structure of the program. For example, the elements of a class slice can be merged into one or more target classes by introduction advice. The following example shows a simple class slice declaration.

Example: class slice declaration

```
slice class Chain {
  Chain *_next;
public:
  Chain *next () const { return _next; }
};
```

2.3 Advice Code

To a code join point so-called advice code can be bound. Advice code can be understood as an action activated by an aspect when a corresponding code join point in a program is reached. The activation of the advice code can happen before, after, or before and after the code join point is reached. The AspectC++ language element to specify advice code is the advice declaration. It is introduced by the keyword `advice` followed by a pointcut expression defining where and under which conditions the advice code shall be activated.

Example: advice declaration

```
advice execution("void login(...)") : before() {
  cout << "Logging in." << endl;
}
```

The code fragment `:before()` following the pointcut expression determines that the advice code shall be activated directly **before** the code join point is reached. It is also possible here to use `:after()` which means **after** reaching the code

join point respectively `:around()` which means that the advice code shall be executed instead of the code described by the code join point. In an **around** advice the advice code can explicitly trigger the execution of the program code at the join point so that advice code can be executed **before** and **after** the join point. There are no special access rights of advice code regarding to program code at a join point.

Beside the pure description of join points pointcuts can also bind variables to context information of a join point. Thus for instance the actual argument values of a function call can be made accessible to the advice code.

Example: advice declaration with access to context information

```
pointcut new_user(const char *name) =
  execution("void login(...)") && args(name);

advice new_user(name) : before(const char *name) {
  cout << "User " << name << " is logging in." << endl;
}
```

In the example above at first the pointcut `new_user` is defined including a context variable `name` that is bound to it. This means that a value of type `const char*` is supplied every time the join point described by the pointcut `new_user` is reached. The pointcut function `args` used in the pointcut expression delivers all join points in the program where an argument of type `const char*` is used. Therefore `args(name)` in touch with the **execution** join point binds `name` to the first and only parameter of the function `login`.

The advice declaration in the example above following the pointcut declaration binds the execution of advice code to the event when a join point described in `new_user` is reached. The context variable that holds the actual value of the parameter of the reached join point has to be declared as a formal parameter of `before`, `after`, or `around`. This parameter can be used in the advice code like an oridinary function parameter.

Beside the pointcut function `args` the binding of context variables is performed by `that`, `target`, and `result`. At the same time these pointcut functions act as filters corresponding to the type of the context variable. For instance `args` in the example above filters all join points having an argument of type `const char*`.

2.3.1 Introductions

The second type of advice supported by AspectC++ are the introductions. Introductions are used to extend program code and data structures in particular. The following example extends two classes each by an attribute and a method.

Example: introductions

```
pointcut shapes() = "Circle" || "Polygon";

advice shapes() : slice class {
  bool m_shaded;
  void shaded(bool state) {
    m_shaded = state;
  }
};
```

Like an ordinary advice declaration an introduction is introduced by the keyword `advice`. If the following pointcut is a name pointcut the slice declaration following the token ":" is introduced in the classes and aspects described by the pointcut. Introduced code can then be used in normal program code like any other function, attribute, etc. Advice code in introductions has full access rights regarding to program code at a join point, i.e. a method introduced in a class has access even to private members of that class.

Slices can also be used to introduce new base classes. In the first line of the following example it is made sure that every class with a name that ends with "Object" is derived from a class `MemoryPool`. This class may implement an own memory management by overloading the `new` and `delete` operators. Classes that inherit from `MemoryPool` must redefine the pure virtual method `release` that is part of the implemented memory management. This is done in the second line for all classes in the pointcut.

Example: base class introduction

```
advice "%Object" : slice class : public MemoryPool {
  virtual void release() = 0;
}
```

2.3.2 Advice Ordering

If more than one advice affects the same join point it might be necessary to define an order of advice execution if there is a dependency between the advice codes (“aspect interaction”). The following example shows how the precedence of advice code can be defined in AspectC++.

Example: advice ordering

```
advice call("% send(...)") : order("Encrypt", "Log");
```

If advice of both aspects (see 2.4) `Encrypt` and `Log` should be run when the function `send(...)` is called this order declaration defines that the advice of `Encrypt` has a higher precedence. More details on advice ordering and precedence can be found in section 8 on page 45.

2.4 Aspects

The aspect is the language element of AspectC++ to collect introductions and advice code implementing a common crosscutting concern in a modular way. This put aspects in a position to manage common state information. They are formulated by means of aspect declarations as a extension to the class concept of C++. The basic structure of an aspect declaration is exactly the same as an usual C++ class definition, except for the keyword `aspect` instead of `class`, `struct` or `union`. According to that, aspects can have attributes and methods and can inherit from classes and even other aspects.

Example: aspect declaration

```
aspect Counter {
  static int m_count;

  pointcut counted() = "Circle" || "Polygon";

  advice counted() : slice struct {
    class Helper {
      Helper() { Counter::m_count++; }
    } m_counter;
  };
```

```
  advice execution("% main(...)") : after() {
    cout << "Final count: " << m_count << " objects"
         << endl;
  }
};
... and at an appropriate place
#include "Counter.ah"
int Counter::m_count = 0;
```

In this example the count of object instantiations for a set of classes is determined. Therefore, an attribute `m_counter` is introduced into the classes described by the pointcut incrementing a global counter on construction time. By applying advice code for the function `main` the final count of object instantiations is displayed when the program terminates.

This example can also be rewritten as an abstract aspect that can for instance be archived in an aspect library for the purpose of reuse. It only require to reimplement the pointcut declaration to be pure virtual.

Example: abstract aspect

```
aspect Counter {
  static int m_count;
  Counter() : m_count(0) {}

  pointcut virtual counted() = 0;
  ...
};
```

It is now possible to inherit from `Counter` to reuse its functionality by reimplementing `counted` to refer to the actual pointcut expression.

Example: reused abstract aspect

```
aspect MyCounter : public Counter {
  pointcut counted() = derived("Shape");
};
```

2.4.1 Aspect Instantiation

By default aspects in AspectC++ are automatically instantiated as global objects. The idea behind it is that aspects can also provide global program properties and therefore have to be always accessible. However in some special cases it may be desired to change this behavior, e.g. in the context of operating systems when an aspect shall be instantiated per process or per thread.

The default instantiation scheme can be changed by defining the static method `aspectof` resp. `aspectOf` that is otherwise generated for an aspect. This method is intended to be always able to return an instance of the appropriate aspect.

Example: aspect instantiation using `aspectof`

```
aspect ThreadCounter : public Counter {
  pointcut counted() = "Thread";

  advice counted() : ThreadCounter m_instance;

  static ThreadCounter *aspectof() {
    return tjp->target()->m_instance;
  }
};
```

The introduction of `m_instance` into `Thread` guarantees that every thread object has an instance of the aspect. By calling `aspectof` it is possible to get this instance at any join point which is essential for accessing advice code and members of the aspect. For this purpose code in `aspectof` has full access to the actual join point in a way described in the next section.

2.5 Runtime Support

2.5.1 Support for Advice Code

For many aspects access to context variables may not be sufficient to get enough information about the join point where advice code was activated. For instance a control flow aspect for a complete logging of function calls in a program would need information about function arguments and its types on runtime to be able to produce a type-compatible output.

In AspectC++ this information is provided by the members of the class `JoinPoint` (see table below).

types:	
`Result`	result type
`That`	object type
`Target`	target type
`AC::Type`	encoded type of an object
`AC::JPType`	join point types
static methods:	
`int args()`	number of arguments
`AC::Type type()`	typ of the function or attribute
`AC::Type argtype(int)`	types of the arguments
`const char *signature()`	signature of the function or attribute
`unsigned id()`	identification of the join point
`AC::Type resulttype()`	result type
`AC::JPType jptype()`	type of join point
non-static methods:	
`void *arg(int)`	actual argument
`Result *result()`	result value
`That *that()`	object refered to by `this`
`Target *target()`	target object of a call
`void proceed()`	execute join point code
`AC::Action &action()`	`Action` structure

Table 1: API of class `JoinPoint` available in advice code

Types and static methods of the `JoinPoint` API deliver information that is the same for every advice code activation. The non-static methods deliver information that differ from one activation to another. These methods are accessed by the object `tjp` resp. `thisJoinPoint` which is of type `JoinPoint` and is always available in advice code, too.

The following example illustrates how to implement a re-usable control flow aspect using the `JoinPoint` API.

Example: re-usable trace aspect

```
aspect Trace {
  pointcut virtual methods() = 0;

  advice execution(methods()) : around() {
```

```
        cout << "before " << JoinPoint::signature() << "(";
        for (unsigned i = 0; i < JoinPoint::args(); i++)
          printvalue(tjp->arg(i), JoinPoint::argtype(i));
        cout << ")" << endl;
        tjp->proceed();
        cout << "after" << endl;
      }
    };
```

This aspect weaves tracing code into every function specified by the virtual pointcut redefined in a derived aspect. The helper function `printvalue` is responsible for the formatted output of the arguments given at the function call. After calling `printvalue` for every argument the program code of the actual join point is executed by calling `proceed` on the `JoinPoint` object. The functionality of `proceed` is achieved by making use of the so-called actions.

2.5.2 Actions

In AspectC++ an action is the statement sequence that would follow a reached join point in a running program if advice code would not have been activated. Thus `tjp->proceed()` triggers the execution of the program code of a join point. This can be the call or execution of a function as well as the writing or reading of attributes or variables. The actions concept is realized in the `AC::Action` structure. In fact, `proceed` is equivalent to `action().trigger()` so that `tjp->proceed()` may also be replaced by `tjp->action().trigger()`. Thereby the method `action()` of the `JoinPoint` API returns the actual action object for a join point.

3 Match Expressions

Match expressions are a used to describe a set of statically known program entities in a C++ source code. These program entities correspond to name join points. Therefore a match expression always returns a name pointcut. There can be match expressions for namespaces, classes, functions or variables.

3.1 Commonly Used Matching Mechanisms

This section describes matching mechanisms that are used in match expressions listed in sections 3.2 to 3.4.

The grammar used for match expression parsing is shown in appendix B on page 48. The following subsections separately describe the name, scope, and type matching mechanisms. All of them are used in match expressions of functions and variables, while match expressions of namespaces and classes only uses name and scope matching.

3.1.1 Name Matching

Name matching is trivial as long as the compared name is a normal C++ identifier. If the *name pattern* does *not* contain the special wildcard character %, it matches a name only if it is exactly the same. Otherwise each wildcard character matches an arbitrary sequence of characters in the compared name. The wildcard character also matches an empty sequence.

Example: simple name patterns

`Token`	only matches `Token`
`%`	matches any name
`parse_%`	matches any name beginning with `parse_` like `parse_declarator` or `parse_`
`parse_%_id%`	matches names like `parse_type_id`, `parse_private_identifier`, etc.
`%_token`	matches all names that end with `_token` like `start_token`, `end_token`, and `_token`

3.1.2 Scope Matching

Restrictions on definition scopes can be described by *scope patterns*. This is a sequence of name patterns (or the special *any scope sequence* pattern `...`), which are separated by `::`, like in `Puma::...::`. A scope pattern always ends with `::` and should never start with `::`, because scope patterns are interpreted relative to the global scope anyway[4]. The definition scope can either be a namespace or a class.

A scope pattern matches the definition scope of a compared function or type if every part can successfully be matched with a corresponding part in the qualified name of the definition scope. The compared qualified name has to be relative to the global scope and should not start with `::`, which is optional in a C++ nested-name-specifier. The special `...` pattern matches any (even empty) sequence

[4] This restriction is also needed to avoid ambiguities in the match expression grammar: Does "`A :: B :: C(int)`" mean "`A ::B::C(int)`" or "`A::B ::C(int)`"?

of scope names. If no scope pattern is given, a compared namespace, class, function or variable has to be defined in the global scope to be matched.

Example: scope patterns

`...::`	matches any definition scope, even the global scope
`Puma::CCParser::`	matches the scope `Puma::CCParser` exactly
`...::%Compiler%::`	matches any class or namespace, which matches the name pattern `%Compiler%`, in any scope
`Puma::...::`	matches any scope defined within the class or namespace `Puma` and `Puma` itself

3.1.3 Type Matching

C++ types can be represented as a tree. For example, the function type `int(double)` is a function type node with two children, one is an `int` node, the other a `double` node. Both children are leaves of the tree.

The types used in match expressions can also be interpreted as trees. As an addition to normal C++ types they can also contain the % wildcard character, name patterns, and scope patterns. A single wildcard character in a type pattern becomes a special *any type node* in the tree representation.

For comparing a type pattern with a specific type the tree representation is used and the *any type node* matches an arbitrary type (sub-)tree.

Example: type patterns with the wildcard character

`%`	matches any type
`void (*)(%)`	matches any pointer type that points to functions with a single argument and a `void` result type
%*	matches any pointer type

Matching of Named Types

Type patterns may also contain name and scope patterns. They become a *named type node* in the tree representation and match any union, struct, class, or enumeration type if its name and scope match the given pattern (see section 3.1.1 and 3.1.2).

Matching of "Pointer to Member" Types

Patterns for pointers to members also contain a scope pattern, e.g. `% (Puma::CSyntax::*)()`. In this context the scope pattern is mandatory.

The pattern is used for matching the class associated with a pointer to member type.

Matching of Qualified Types (`const`/`volatile`)

Many C++ types can be qualified as `const` or `volatile`. In a type pattern these qualifier can also be used, but they are interpreted restrictions. If no `const` or `volatile` qualifier is given in a type pattern, the pattern also matches qualified types[5].

Example: type patterns with `const` and `volatile`

`%`	matches any type, even types qualified with `const` or `volatile`
`const %`	matches only types qualified by `const`
`% (*)() const volatile`	matches the type of all pointers to functions that are qualified by `const` and `volatile`

Handling of Conversion Function Types

The result type of conversion functions is interpreted as a special *undefined* type in type patterns as well as in compared types. The *undefined* type is only matched by the *any type* node and the *undefined type* node.

Ellipses in Function Type Patterns

In the list of function argument types the type pattern `...` can be used to match an arbitrary (even empty) list of types. The `...` pattern should not be followed by other argument type patterns in the list of argument types.

Matching Virtual Functions

The *decl-specifier-seq* of a function type match expression may include the keyword `virtual`. In this case the function type match expression only matches virtual or pure virtual member functions. As `const` and `volatile`, the `virtual` keyword is regarded as a restriction. This means that a function type match expression without `virtual` matches virtual and non-virtual functions.

[5] Matching only non-constant or non-volatile types can be achieved by using the operators explained in section 4.9 on page 39. For example, `!"const %"` describes all types which are not constant.

Example: type patterns with `virtual`

`virtual % ...::%(...)`	matches all virtual or pure virtual functions in any scope
`% C::%(...)`	matches all member functions of C, even if they are virtual

Matching Static Functions

Matching static functions works similar as matching virtual functions. The *decl-specifier-seq* of a function type match expression may include the keyword `static`. In this case the function type match expression only matches static functions in global or namespace scope and static member functions of classes. As `const` and `volatile`, the `static` keyword is regarded as a restriction. This means that a function type match expression without `static` matches static and non-static functions.

Example: type patterns with `static`

`static % ...::%(...)`	matches all static member and non-member functions in any scope
`% C::%(...)`	matches all member functions of C, even if they are static

Argument Type Adjustment

Argument types in type patterns are adjusted according to the usual C++ rules, i.e. array and function types are converted to pointers to the given type and `const`/`volatile` qualifiers are removed. Furthermore, argument type lists containing a single `void` type are converted into an empty argument type list.

3.2 Namespace and Class Match Expressions

For namespaces and classes the matching process is special because it consists of two steps.

First, each namespace and class is compared with a given match expression. A match expression that matches a namespace or class begins with the optional scope part and ends with the required name part. In course of this step the matching name join points are collected in a temporary pointcut.

Example: scope and name parts of a namespace or class match expression

```
"Puma::...::Parser%"
```

This match expression describes the following requirements on a compared namespace or class:

scope: the scope in which the namespace or class is defined has to match `Puma::...::`

name: the name of the namespace or class has to match the name pattern `Parser%`

For more information about these parts see sections Scope Matching (3.1.2) and Name Matching (3.1.1).

In the second step the temporary pointcut will be extended by contained name join points yielding the result pointcut. The extension rules are as follows:

- If a namespace *N* is matched, the resulting pointcut additionally contains the following name join points:
  all functions, variables, (nested) classes, member functions, data members, constructors and destructors that are anywhere and arbitrary nested inside *N*.

- If a class *C* is matched, the resulting pointcut additionally contains the following name join points:
  all member functions, data members and constructors of *C* as well as the destructor of *C* that are directly located inside *C*. So name join points that are nested inside a member function, a data member or a nested class are **not** added to the pointcut.

The following list contains example match expressions and the results after the first as well as after the second step.

	after step one	**result**
`Token`	only matches namespaces or classes with the name `Token` that are directly inside the global namespace	*step one extended as described in step two*
`....::Token`	matches `Token` at arbitrary location	*step one extended as described in step two*
`%`	matches any namespace or class that is directly located in the global namespace but not the global namespace itself	matches any namespace except the global namespace, any class that is arbitrary nested in a non-global namespace, any class directly located in the global namespace and all functions, member functions, variables, data members, constructors and destructors that are contained in one of the just mentioned entities
::	matches the global namespace	matches any function, variable, (nested) class, member function, data member, constructor or destructor

	after step one	result
`OOStuBS::CGA%`	matches any namespace or class inside `OOStuBS` beginning with `CGA` like `OOStuBS::CGA`, `OOStuBS::CGA_Screen` or `OOStuBS::CGA_Stream`. Note that this matches `OOStuBS` only inside the global namespace.	*step one extended as described in step two*
`%::Smtp%Bldr%`	matches namespaces and classes like `SmtpBldr`, `SmtpClientBldr` or `SmtpServerBldrCreator`, that are nested in exact one namespace or class.	*step one extended as described in step two*
`%Node`	matches any namespace or class ending with `Node` like `ModelNode`, `GraphNode` and `Node`	*step one extended as described in step two*

Please note that local classes inside functions or member functions are never matched.

3.3 Function Match Expressions

For function (or member function) matching a match expression is internally decomposed into the function type pattern, the scope pattern, and the name pattern.

Example: type, scope, and name parts of a function match expression

```
"const % Puma::...::parse_% (Token *)"
```

This match expression describes the following requirements on a compared function name:

name: the function name has to match the name pattern `parse_%`

scope: the scope in which the function is defined has to match `Puma::...::`

type: the function type has to match `const %(Token *)`

If an entity matches all parts of the match expression, it becomes an element of the pointcut, which is defined and returned by the match expression.

Common descriptions of name, scope and type matching can be found in section 3.1. The following sections additionally describe the name matching of special functions.

3.3.1 Operator Function and Conversion Function Name Matching

The name matching mechanism is more complicated if the pattern is compared with the name of a conversion function or an operator function. Both are matched by the name pattern `%`. However, with a different name pattern than `%` they are only matched if the pattern begins with "`operator` ". The pattern "`operator %`" matches any operator function or conversion function name.

C++ defines a fixed set of operators which are allowed to be overloaded. In a name pattern the same operators may be used after the "`operator` " prefix to match a specific operator function name. Operator names in name patterns are not allowed to contain the wildcard character. For ambiguity resolution the operators `%` and `%=` are matched by `%%` and `%%=` in a name pattern.

Example: operator name patterns

`operator %`	matches any operator function name (as well as any conversion function name)
`operator +=`	matches only the name of a `+=` operator
`operator %%`	matches the name of an `operator %`

Conversion functions don't have a real name. For example, the conversion function `operator int*()` defined in a class `C` defines a conversion from a `C` instance into an object of type `int*`. To match conversion functions the name pattern may contain a type pattern after the prefix "`operator` ". The type matching mechanism is explained in section 3.1.3.

Example: conversion function name patterns

`operator %`	matches any conversion function name
`operator int*`	matches any name of a conversion that converts something into an `int*` object
`operator %*`	matches any conversion function name if that function converts something into a pointer

3.3.2 Constructors and Destructors

Name patterns cannot be used to match constructor or destructor names.

3.4 Variable Match Expressions

For variable (or member) matching a match expression is internally decomposed into the variable type pattern, the scope pattern, and the name pattern.

Example: type, scope, and name parts of a variable match expression

```
"const % Puma::...::parsed_%"
```

This match expression describes the following requirements on a compared variable name:

name: the variable name has to match the name pattern `parsed_%`

scope: the scope in which the variable is defined has to match `Puma::...::`

type: the variable type has to match `const %`

If an entity matches all parts of the match expression, it becomes an element of the pointcut, which is defined and returned by the match expression.

Descriptions of name, scope and type matching can be found in section 3.1.

4 Predefined Pointcut Functions

On the following pages a complete list of the pointcut functions supported by AspectC++ is presented. For every pointcut function it is indicated which type of pointcut is expected as argument(s) and of which type the result pointcut is. Thereby "N" stands for name pointcut and "C" for code pointcut. The optionally given index is an assurance about the type of join point(s) described by the result pointcut[6]. If a pointcut is used as argument of a pointcut function and the type of some join points in argument pointcut does not match one of the expected argument types of the pointcut function, these non-matching join points are silently ignored.

4.1 Types

base(*pointcut*) $\mathsf{N}_{C,F,V} \rightarrow \mathsf{N}_{C,F,V}$

returns a pointcut p_b of name join points created as follows

[6] C, C_{C}, C_{E}, C_{B}, C_{S}, C_{G}: Code (any, only *Call* (without Builtin), only *Execution*, only *Builtin*, only *Set*, only *Get*); N, N_N, N_C, N_F, $\mathsf{N}_T\mathsf{N}_V$: Names (any, only *Namespace*, only *Class*, only *Function*, only *Type*, only *Variable*)

$p_b \leftarrow \{$all base classes of classes in *pointcut* but not the classes in *pointcut*$\}$,
$p_b \leftarrow p_b \,||\, \{$all member functions and data members of classes in $p_b\}$,
$p_b \leftarrow p_b \,||\, \{$all previous definitions of member functions in *pointcut* but not the member functions in *pointcut*$\}$,
$p_b \leftarrow p_b \,||\, \{$all previous definitions of data members in *pointcut* but not the data members in *pointcut*$\}$

derived(*pointcut*) $N_{C,F,V} \rightarrow N_{C,F,V}$

returns a pointcut p_d of name join points created as follows
$p_d \leftarrow \{$all classes in *pointcut* and all classes derived from them$\}$,
$p_d \leftarrow p_d \,||\, \{$all member functions and data members of classes in $p_d\}$,
$p_d \leftarrow p_d \,||\, \{$all member functions in $pointcut$ and all redefinitions of these member functions in derived classes$\}$,
$p_d \leftarrow p_d \,||\, \{$all data members in $pointcut$ and all redefinitions of these data members in derived classes$\}$

Example: derived function matching

```
struct A {};
struct B : public A { void f(); };
struct C : public B { void f(); };
aspect Z {
  advice execution(derived("A")) : before() {
    // before execution of B::f() or C::f()
  }
};
```

Example: type matching

A software may contain the following class hierarchy.

```
class Shape { ... };
class Scalable { ... };
class Point : public Shape { ... };
...
class Rectangle : public Line, public Rotatable { ... };
```

With the following aspect a special feature is added to a designated set of classes of this class hierarchy.

```
aspect Scale {
  pointcut scalable() = "Rectangle" ||
    (base("Rectangle") && derived("Point"));

  advice "Point" : slice class : public Scalable;
  advice scalable() : slice class {
      void scale(int value) { ... }
  };
};
```

The pointcut describes the classes `Point` and `Rectangle` and all classes derived from `Point` that are direct or indirect base classes of `Rectangle`. With the first advice `Point` gets a new base class. The second advice adds a corresponding method to all classes in the pointcut.

4.2 Control Flow

cflow(*pointcut*) C→C

captures join points occurring in the dynamic execution context of join points in *pointcut*. Currently the language features being used in the argument pointcut are restricted. The argument is not allowed to contain any context variable bindings (see 4.8) or other pointcut functions which have to be evaluated at runtime like **cflow**(*pointcut*) itself.

Example: control flow dependant advice activation

The following example demonstrates the use of the **cflow** pointcut function.

```
class Bus {
  void out (unsigned char);
  unsigned char in ();
};
```

Consider the class `Bus` shown above. It might be part of an operating system kernel and is used there to access peripheral devices via a special I/O bus. The execution of the member functions `in()` and `out()` should not be interrupted, because this would break the timing of the bus communication. Therefore, we decide to implement an interrupt synchronization aspect that disables interrupts during the execution of in() and out():

```
aspect BusIntSync {
  pointcut critical() = execution("% Bus::%(...)");
  advice critical() && !cflow(execution("% os::
     int_handler()")) : around() {
    os::disable_ints();
    tjp->proceed();
    os::enable_ints();
  }
};
```

As the bus driver code might also be called from an interrupt handler, the interrupts should not be disabled in any case. Therefore, the pointcut expression exploits the **cflow()** pointcut function to add a runtime condition for the advice activation. The advice body should only be executed if the control flow did not come from the interrupt handler `os::int_handler()`, because it is not interruptable by definition and `os::enable_ints()` in the advice body would turn on the interrupts too early.

4.3 Scope

within(*pointcut*) N→C

returns all code join points that are located directly inside or at a name join point in *pointcut*

member(*pointcut*) N→N

maps the scopes given in *pointcut* to any contained named entities. Thus a class name for example is mapped to all contained member functions, variables and nested types.

Example: matching in scopes

```
aspect Logger {
  pointcut calls() =
    call("void transmit()") && within("Transmitter");

  advice calls() : around() {
    cout << "transmitting ... " << flush;
    tjp->proceed();
    cout << "finished." << endl;
  }
```

```
};
```

This aspect inserts code logging all calls to `transmit` that are within the methods of class `Transmitter`.

4.4 Functions

call(*pointcut*) $N_F \rightarrow C_C$

returns all code join points where a user provided function or member function in *pointcut* is called. The resulting join points are located in the scope of the resp. caller meaning where the function or member functions is called. The pointcut does not include join points at calls to built-in operators.

execution(*pointcut*) $N_F \rightarrow C_E$

returns all code join points where a function or member function in *pointcut* is executed. The resulting join points are located in the scope of the callee meaning where the function or member function is defined/implemented.

Example: function matching

The following aspect weaves debugging code into a program that checks whether a method is called on a null pointer and whether the argument of the call is null.

```
aspect Debug {
  pointcut fct() = "% MemPool::dealloc(void*)";
  pointcut exec() = execution(fct());
  pointcut calls() = call(fct());

  advice exec() && args(ptr) : before(void *ptr) {
    assert(ptr && "argument is NULL");
  }
  advice calls() : before() {
    assert(tjp->target() && "'this' is NULL");
  }
};
```

The first advice provides code to check the argument of the function `dealloc` before the function is executed. A check whether `dealloc` is called on a null object is provided by the second advice. This is realized by checking the target of the call.

4.5 Built-in Operators

builtin(*pointcut*) $N_F \rightarrow C_B$

returns all code join points where a built-in operator in *pointcut* is called. This pointcut function does not return join points at constructor or destructor calls. See section Object Construction and Destruction (4.6) to find out how to describe these join points.
The builtin pointcut function is a new feature that was introduced in version 2.0 and is therefore not enabled by default to avoid compatibility issues (e.g., if someone named a pointcut "builtin"). The `--builtin_operators` command-line argument enables the described functionality.

The intersection of the results of **call** and **builtin** always yields the empty pointcut:
call(*pointcut*) && builtin(*pointcut*) = $\varnothing \quad \forall$*pointcut*

Example: operator matching

The following aspect weaves code into a program that checks whether a null-pointer will be dereferenced. If this occurs, the advice will provide the code position on the error stream.

```
aspect ProblemReporter {
  advice builtin("% operator *(%)") : before() {
    if(*tjp->arg<0>() == 0) {
      cerr << tjp->filename() << " (Line " << tjp->
        line() << "): dereferencing of null-
        pointer!" << endl;
    }
  }
};
```

4.5.1 Limitations

Some built-in operators could not be fully supported. For example, weaving advice code for built-in operators in **constant expressions** would destroy the constancy of the expressions and inhibit evaluation at compile time. Therefore, operators in constant expressions are not matched. The following code listing gives some examples for operators in constant expressions.

```
class ExampleClass {
  static const int const_member = 5 * 2;
```

```
  unsigned int bitfield : 4 / 2;
};
const int const_two = 3 - 1;
static char char_array[const_two + 5];
enum ExampleEnum {
  ENUM_VALUE = const_two + 1
};
switch(const int const_temp = 1) {
  case const_temp + 1: {
    // ...
    break;
  }
}
```

A further limitation results from the fact, that the C++-standard forbids **pointers and references to bit-fields**. Thus all operators that refer to a bit-field (e.g. the assignment- or increment-/decrement-operator needs a reference as first argument) are not supported.

Moreover any operator that has an **anonymous/unnamed or local type or a type with no linkage** as argument or result is not supported (because these types shall not be used as a template argument which makes weaving impossible in most cases).

Additionally **postfix increment/decrement operators** have a second implicit argument of type `int` to distinguish between pre- and postfix operators. So e.g. "`% operator ++(%, int)`" matches the postfix increment operator and "`% operator ++(%)`" matches the prefix increment operator.

Also the **address-of operator `&`** is not supported, if the argument is a data member or member function, because these types do not exist as type of a variable.

Furthermore the C++-standard states that if the result of .* or ->* is a function, that result can be used only as the operand for the function call operator `()`. Therefore the **pointer to member operators `.*` and `->*`** that get a member function pointer as second argument are not supported, because a caching of the result is not possible.

At last there are some limitations with the **short-circuiting operators `&&`,**

|| and ?:. If the second or third argument is not evaluated, tjp->args() will return a null-pointer for the corresponding argument. Additionally the result of the args pointcut function (see 4.8) is determined at runtime, if an short-circuit argument is bound with the args pointcut function . Thus the advice code in the following example is only executed, if the first argument evaluates to `true` so that the second argument is available. In case of `||` the first argument have to be `false` to make the second argument available and in case of `?:` the first argument makes the decision about the availability of the second resp. third argument.

```
advice builtin("% operator &&(bool, bool)") && args("%", b2
   ) : before(bool b2) {
  // advice code
}
```

A **complete list with all limitations and not supported operators** can be found in the next section 4.5.2.

4.5.2 Supported And Not Supported Operators

This section contains information about the builtin pointcut function in terms of supported operators.
Table 2 shows all operators that are fully or partly supported and indicates the special characteristics of these operators, if available. For more information see section 4.5.1.
Table 3 shows not supported operators.

operator and example			special characteristics
unary	`++`	`a++`	postfix operator (second argument of type `int`)
unary	`--`	`a--`	postfix operator (second argument of type `int`)
unary	`++`	`++a`	prefix operator; not supported if `a` is a bit-field
unary	`--`	`--a`	prefix operator; not supported if `a` is a bit-field
unary	`&`	`&a`	not supported if `a` is a member
unary	`*`	`*a`	
unary	`+`	`+a`	
unary	`-`	`-a`	
unary	`~`	`~a`	
unary	`!`	`!a`	
binary	`.*`	`a.*b`	not supported if `b` is a member function pointer
binary	`->*`	`a->*b`	not supported if `b` is a member function pointer

continuation on next page...

operator and example			special characteristics
binary	`*`	`a*b`	
binary	`/`	`a/b`	
binary	`%`	`a%b`	in match expressions: escape `%` with `%%`
binary	`+`	`a+b`	
binary	`-`	`a-b`	
binary	`<<`	`a<<b`	
binary	`>>`	`a>>b`	
binary	`<`	`a<b`	
binary	`>`	`a>b`	
binary	`<=`	`a<=b`	
binary	`>=`	`a>=b`	
binary	`==`	`a==b`	
binary	`!=`	`a!=b`	
binary	`&`	`a&b`	
binary	`^`	`a^b`	
binary	`|`	`a|b`	
binary	`&&`	`a&&b`	limitations due to short-circuit evaluation (see 4.5.1)
binary	`||`	`a||b`	limitations due to short-circuit evaluation (see 4.5.1)
binary	`=`	`a=b`	copy-assignment not supported; not supported if `a` is a bit-field
binary	`*=`	`a*=b`	not supported if `a` is a bit-field
binary	`/=`	`a/=b`	not supported if `a` is a bit-field
binary	`%=`	`a%=b`	in match expressions: escape `%=` with `%%=`; not supported if `a` is a bit-field
binary	`+=`	`a+=b`	not supported if `a` is a bit-field
binary	`-=`	`a-=b`	not supported if `a` is a bit-field
binary	`<<=`	`a<<=b`	not supported if `a` is a bit-field
binary	`>>=`	`a>>=b`	not supported if `a` is a bit-field
binary	`&=`	`a&=b`	not supported if `a` is a bit-field
binary	`|=`	`a|=b`	not supported if `a` is a bit-field
binary	`^=`	`a^=b`	not supported if `a` is a bit-field
binary	`[]`	`a[b]`	
ternary	`?:`	`a?b:c`	limitations due to short-circuit evaluation (see 4.5.1)

Table 2: operators that are (partly) supported by the builtin pointcut function

operator and example		
binary	`,`	`a,b`
binary	`->`	`a->b`
binary	`.`	`a.b`
`new`		`new a`
`delete`		`delete a`
`new[]`		`new[] a`
`delete[]`		`delete[] a`
implicit conversions		
operators in constant expressions (see 4.5.1)		
operators with an anonymous/unnamed or local type (see 4.5.1)		
operators that have a type with no linkage (see 4.5.1)		

Table 3: operators that are not supported by the builtin pointcut function

4.6 Object Construction and Destruction

construction(*pointcut*) $N_C \rightarrow C_{Cons}$

returns all code join points where an instance of a class in *pointcut* is constructed. The construction join point begins after all base class and member construction join points. It can be imagined as the execution of the constructor. However, advice for construction join points work, even if there is no constructor defined explicitly. A construction join point has arguments and argument types, which can be exposed or filtered, e.g. by using the **args** pointcut function.

destruction(*pointcut*) $N_C \rightarrow C_{Des}$

returns all code join points where an instance of a class in *pointcut* is destructed. The destruction join point ends before the destruction join point of all members and base classes. It can be imagined as the execution of the destructor, although a destructor does not to be defined explicitly. A destruction join point has an empty argument list.

Example: instance counting

The following aspect counts how many instances of the class `ClassOfInterest` are created and destroyed.

```
aspect InstanceCounting {
  // the class for which instances should be counted
  pointcut observed() = "ClassOfInterest";
  // count constructions and destructions
  advice construction (observed ()) : before () {
                                        _created++; }
  advice destruction (observed ())  : after ()  {
                                        _destroyed++; }
  // counters
  int _created, _destroyed;
public:
  // Singleton aspects can have a default constructor
  InstanceCounting () { _created = _destroyed = 0; }
};
```

The implementation of this aspect is straightforward. Two counters are initialized by the aspect constructor and incremented by the construction/destruction advice. By defining `observed()` as a pure virtual pointcut the aspect can easily be transformed into a reusable abstract aspect.

4.7 Variables

get(*pointcut*) $N_V \rightarrow C_G$

returns all code join points where a global variable or data member in *pointcut* is read. The get join points are located at implicit lvalue-to-rvalue conversions according to the C++ standard. In addition, the get join points are located within all built-in compound-assignment operators, and within the built-in increment and decrement operators.

set(*pointcut*) $N_V \rightarrow C_S$

returns all code join points where a global variable or data member in *pointcut* is modified. The set join points are located within all built-in assignment operators, and within the built-in increment and decrement operators. The initialization of a global variable or data member provides no set join point.

ref(*pointcut*) $N_V \rightarrow C_R$

provides all join points where a reference (reference type or pointer) to a

global variable or data member in the *pointcut* is created. The ref join points are located within the built-in address-of operator `&`, if the operand is a global variable or data member. In addition, the ref join points are located before the initialization of a variable of reference type, including return values. Moreover, the binding of a reference parameter of a function, including default values, provides ref join points. The ref join points are also located within implicit array-to-pointer conversions according to the C++ standard.

Example: variable matching

The following aspect observes the modification of all variables (in any scope) of the type `int`. When such an integer variable is modified, the aspect reports the name of the variable and its new value, obtained by `*tjp->entity()`.

```
aspect IntegerModification {
  advice set("int ...::%") : after() {
    cout << "Setting variable "
         << tjp->signature() << " to "
         << *tjp->entity() << endl;
  }
};
```

4.7.1 Limitations

The get and set pointcut functions cover variables of fundamental type, such as integer and floating-point types, and arrays thereof. Variables of any pointer type and arrays of pointers are also supported. The get and set pointcut functions do not support **variables of class type, unions, enumerations, bitfields, and references**.

The get, set, and ref pointcut functions match only if the variable is accessed directly by its name. **Indirect variable access via pointer or reference** does not match.

The get, set, and ref pointcut functions do not match for **local variables**.

The get, set, and ref joinpoints are not located within **constant expressions**, such as the built-in operator `sizeof`.

4.7.2 Compatibility

The get, set, and ref pointcut functions are new features that were introduced in version 2.0 and are therefore not enabled by default to avoid compatibility issues (e.g., if someone named a pointcut "get"). The `--data_joinpoints` command-line argument enables the described functionality.

4.8 Context

that(*type pattern*) $N_T \rightarrow C$

returns all code join points where the current C++ `this` pointer refers to an object which is an instance of a type that is compatible to the type described by *type pattern*

target(*type pattern*) $N_T \rightarrow C$

returns all code join points where the target object of a call/set/get is an instance of a type that is compatible to the type described by *type pattern*

result(*type pattern*) $N_T \rightarrow C$

returns all code join points where the type of the return value of a call/built-in/execution/get is matched by *type pattern*

args(*type pattern*, ...) $(N_T,...) \rightarrow C$

returns all code join points where the types of the arguments of a call/built-in/execution/set are matched by the corresponding *type pattern*s.

Instead of the type pattern it is also possible here to pass the name of a variable to which the context information is bound (a **context variable**). In this case the type of the variable is used for the type matching. Context variables must be declared in the argument list of **before()**, **after()**, or **around()** and can be used like a function parameter in the advice body.

The **that()** and **target()** pointcut functions are special, because they might cause a runtime type check. The **args()** and **result()** functions are evaluated at compile time. Exception: If a short-circuit argument is bound with the args pointcut function, then the result of args depends on the runtime availability of the bound argument.

Example: context matching

4.9 Algebraic Operators

pointcut **&&** *pointcut* (N,N)→N, (C,C)→C
: returns the intersection of the join points in the *pointcut*s

pointcut || *pointcut* (N,N)→N, (C,C)→C
: returns the union of the join points in the *pointcut*s

! *pointcut* N→N, C→C
: returns all name resp. code join points that are not included in *pointcut*

Example: combining pointcut expressions

5 Slices

This section defines the syntax and semantics of slice declarations. The next section will describe how slices can be used by advice in order to introduce code. Currently, only class slices are defined in AspectC++.

5.1 Class Slice Declarations

Class slices may be declared in any class or namespace scope. They may be defined only once, but there may be an arbitrary number of forward declarations. A qualified name may be used if a class slice that is already declared in a certain scope is redeclared or defined as shown in the following example:

```
slice class ASlice;
namespace N {
 slice class ASlice; // a different slice!
}
slice class ASlice { // definition of the ::ASlice
  int elem;
};
slice class N::ASlice { // definition of the N::ASlice
  long elem;
};
```

If a class slice only defines a base class, an abbreviated syntax may be used:

```
slice class Chained : public Chain;
```

Class slices may be anonymous. However, this only makes sense as part of an advice declaration. A class slice may also be declared with the `aspect` or `struct` keyword instead of `class`. While there is no difference between class and aspect slices, the default access rights to the elements of a struct slice in the target classes are public instead of private. It is forbidden to declare aspects, pointcuts, advice, or slices as members of a class slice.

Class slices may have members that are not defined within the body of a class slice declaration, e.g. static attributes or non-inline functions:

```
slice class SL {
  static int answer;
  void f();
};
//...
slice int SL::answer = 42;
slice void SL::f() { ... }
```

These external member declarations have to appear after the corresponding slice declaration in the source code.

6 Advice

This section describes the different types of advice offered by AspectC++. Advice are categorized in advice for join points in the dynamic control flow of the running program, e. g. function call or executions, and advice for static join points like introductions into classes.

In either case the compiler makes sure that the code of the aspect header file, which contains the advice definition (if this is the case), is compiled prior to the affected join point location.

6.1 Advice for Dynamic Join Points

before(...)
: the advice code is executed before the join points in the pointcut

after(...)
: the advice code is executed after the join points in the pointcut

around(...)
: the advice code is executed in place of the join points in the pointcut

6.2 Advice for Static Join Points

Static join points in AspectC++ are classes or aspects. Advice for classes or aspects can introduce new members or add a base class. Whether the new member or base class becomes **private**, **protected**, or **public** in the target class depends on the protection in the advice declaration in the aspect.

baseclass(*classname*)
: a new base class is introduced to the classes in the pointcut

introduction declaration
: a new attribute, member function, or type is introduced

Introduction declarations are only semantically analyzed in the context of the target. Therefore, the declaration may refer, for instance, to types or constants, which are not known in the aspect definition, but only in the target class or classes. To introduce a constructor or destructor the name of the aspect, to which the introduction belongs, has to be taken as the constructor/destructor name.

Non-inline introductions can be used for introductions of static attributes or member function introduction with separate declaration an definition. The name of the introduced member has to be a qualified name in which the nested name specifier is the name of the aspect to which the introduction belongs.

7 `JoinPoint` API

The following sections provide a complete description of the `JoinPoint` API.

7.1 API for Dynamic Join Points

The JoinPoint-API for dynamic join points can be used within the body of advice code.

7.1.1 Types and Constants

`Result`
: result type of a function

`That`
: object type (object initiating a call)

`Target`
: target object type (target object of a call)

`Entity`
: type of the primary referenced entity (function or variable)

`MemberPtr`
: type of the member pointer for entity or `void *` for nonmembers

`Array`
: type of the accessed array

`Dim<i>::Idx`
: type of the *i*th dimension of the accessed array (with $0 \leq i < DIMS$)

`Dim<i>::Size`
: size of the *i*th dimension of the accessed array (with $0 \leq i < DIMS$)

`DIMS`
: number of dimensions of an accessed array or 0 otherwise

Example: type usage

7.1.2 Functions

`static AC::Type type()`
: returns the encoded type for the join point conforming with the C++ ABI V3 specification[7]

`static int args()`
: returns the number of arguments of a function for call and execution join points

`static AC::Type argtype(int number)`
: returns the encoded type of an argument conforming with the C++ ABI V3 specification

`static const char *signature()`
: gives a textual description of the join point (function name, class name, ...)

`static unsigned int id()`
: returns a unique numeric identifier for this join point

[7] http://www.codesourcery.com/cxx-abi/abi.html\#mangling

`static const char *filename()`
: returns the name of the file in which the join point (shadow) is located

`static int line()`
: the number of the line in which the join point (shadow) is located

`static AC::Type resulttype()`
: returns the encoded type of the result type conforming with the C++ ABI V3 specification

`static AC::JPType jptype()`
: returns a unique identifier describing the type of the join point

Example: static function usage

`void *arg(int number)`
: returns a pointer to the memory position holding the argument value with index `number`

`Result *result()`
: returns a pointer to the memory location designated for the result value or 0 if the function has no result value

`That *that()`
: returns a pointer to the object initiating a call or 0 if it is a static method or a global function

`Target *target()`
: returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function

`Entity *entity()`
: returns a pointer to the accessed entity (function or variable) or 0 for member functions or builtin operators

`MemberPtr *memberptr()`
: returns a member pointer to entity or 0 for nonmembers

`Array *array()`
: returns a typed pointer to the accessed array

`Dim<i>::Idx idx<i>()`
: returns the value of the *i*th index used for the array access

`void proceed()`
: executes the original join point code in an around advice by calling `action().trigger()`

`AC::Action &action()`
: returns the runtime action object containing the execution environment to execute the original functionality encapsulated by an around advice

Example: non-static function usage

7.2 API for Static Join Points

The JoinPoint-API for static join points can be used within the definition of a slice and describes the state of target class *before* the introduction took place. It is accessed through the built-in type `JoinPoint` (e.g. `JoinPoint::signature()`) and provides the following functions, types, and constants:

`static const char *signature()`
: returns the target class name as a string

`That`
: The (incomplete) target type of the introduction

`HASHCODE`
: integer hash value of the target type

`BASECLASSES`
: number of base classes of the target class

`BaseClass<I>::Type`
: type of the I^{th} base class

`BaseClass<I>::prot, BaseClass<I>::spec`
: Protection level (AC::PROT_NONE /PRIVATE /PROTECTED /PUBLIC) and additional specifiers (AC::SPEC_NONE /VIRTUAL) of the I^{th} base class

`MEMBERS`
: number of data members of the target class

`Member<I>::Type, Member<I>::ReferredType`
: type of the I^{th} attribute of the target class

`Member<I>::prot, Member<I>::spec`
: Protection level (see BaseClass<I>::prot) and additional attribute specifiers (AC::SPEC_NONE /STATIC /MUTABLE)

`static ReferredType *Member<I>::pointer(T *obj=0)`
: returns a typed pointer to the I^{th} attribute (obj is needed for non-static attributes)

`static const char *Member<I>::name()`
: returns the name of the I^{th} attribute

`FUNCTIONS`
: number of member functions of the target class

`Function<I>::prot, Function<I>::spec`
: Protection level (see BaseClass<I>::prot) and additional attribute specifiers (AC::SPEC_NONE /STATIC /VIRTUAL)

`CONSTRUCTORS`
: number of user-defined constructors of the target class

`Constructor<I>::prot, Constructor<I>::spec`
: Protection level (see BaseClass<I>::prot) and additional attribute specifiers (AC::SPEC_NONE)

`DESTRUCTORS`
: number (zero or one) of user-defined destructors of the target class

`Destructor<I>::prot, Destructor<I>::spec`
: Protection level (see BaseClass<I>::prot) and additional attribute specifiers (AC::SPEC_NONE /VIRTUAL)

8 Advice Ordering

8.1 Aspect Precedence

AspectC++ provides a very flexible mechanism to define aspect precedence. The precedence is used to determine the execution order of advice code if more than one aspect affects the same join point. The precedence in AspectC++ is an attribute of a join point. This means that the precedence relationship between two aspects might vary in different parts of the system. The compiler checks the following conditions to determine the precedence of aspects:

order declaration: if the programmer provides an order declaration, which defines the precedence relationship between two aspects for a join point, the compiler will obey this definition or abort with a compile-time error if there

is a cycle in the precedence graph. Order declarations have the following syntax:
advice *pointcut-expr* : **order (** *high, ...low* **)**
The argument list of `order` has to contain at least two elements. Each element is a pointcut expression, which describes a set of aspects. Each aspect in a certain set has a higher precedence than all aspects, which are part of a set following later in the list (on the right hand side). For example `'("A1"||"A2","A3"||"A4")'` means that `A1` has precedence over `A3` and `A4` and that `A2` has precedence over `A3` and `A4`. This order directive does *not* define the relation between `A1` and `A2` or `A3` and `A4`. Of course, the pointcut expressions in the argument list of `order` may contain named pointcuts and even pure virtual pointcuts.

inheritance relation: if there is no order declaration given and one aspect has a base aspect the derived aspect has a higher precedence than the base aspect.

8.2 Advice Precedence

The precedence of advice is determined with a very simple scheme:

- if two advice declarations belong to different aspects and there is a precedence relation between these aspects (see section 8.1 on the previous page) the same relation will be assumed for the advice.
- if two advice declarations belong to the same aspect the one that is declared first has the higher precedence.

8.3 Effects of Advice Precedence

Only advice precedence has an effect on the generated code. The effect depends on the kind of join point, which is affected by two advice declarations.

Class Join Points

Advice on class join points can extend the attribute list or base class list. If advice has a higher precedence than another it will be handled first. For example, an introduced new base class of advice with a high precedence will appear in the base class list on the left side of a base class, which was inserted by advice with lower precedence. This means that the execution order of the constructors of introduced base classes can be influenced, for instance, by order declarations.

The order of introduced attributes also has an impact on the constructor/destructor execution order as well as the object layout.

Code Join Points

Advice on code join points can be `before`, `after`, or `around` advice. For `before` and `around` advice a higher precedence means that the corresponding advice code will be run first. For `after` advice a higher precedence means that the advice code will be run later.

If `around` advice code does not call `tjp->proceed()` or `trigger()` on the action object no advice code with lower precedence will be run. The execution of advice with higher precedence is not affected by `around` advice with lower precedence.

For example, consider an aspect that defines advice[8] in the following order: BE1, AF1, AF2, AR1, BE2, AR2, AF3. As described in section 8.2 on the facing page the declaration order also defines the precedence: BE1 has the highest and AF3 the lowest. The result is the following advice code execution sequence:

1. BE1 (highest precedence)
2. AR1 (the indented advice will only be executed if `proceed()` is called!)
   (a) BE2 (before AR2, buts depends on AR1)
   (b) AR2 (the indented code will only be executed if `proceed()` is called!)
      i. original code under the join point
      ii. AF3
3. AF2 (does not depend on AR1 and AR2, because of higher precedence)
4. AF1 (run after AF2, because it has a higher precedence)

A Grammar

The AspectC++ syntax is an extension to the C++ syntax. It adds four new keywords to the C++ language: `aspect`, `advice`, `slice`, and `pointcut`. Additionally it extends the C++ language by advice and pointcut declarations. In contrast to pointcut declarations, advice declarations may only occur in aspect declarations.

[8] BE is `before` advice, AF `after` advice, and AR `around` advice

class-key:
`aspect`

declaration:
pointcut-declaration
slice-declaration
advice-declaration

member-declaration:
pointcut-declaration
slice-declaration
advice-declaration

pointcut-declaration:
`pointcut` *declaration*

pointcut-expression:
constant-expression

advice-declaration:
`advice` *pointcut-expression* **:** *order-declaration*
`advice` *pointcut-expression* **:** *slice-reference*
`advice` *pointcut-expression* **:** *declaration*

order-declaration:
`order (` *pointcur-expression-seq* `)`

slice-reference:
`slice` **::**$_{opt}$ *nested-name-specifier*$_{opt}$ *unqualified-id* **;**

slice-declaration:
`slice` *declaration*

B Match Expression Grammar

Match expression in AspectC++ are used to define a type pattern and an optional object name pattern to select a subset of the known program entities like functions, attributes, or argument/result types. The grammar is very similar to the grammar of C++ declarations. Any rules, which are referenced here but not defined, should be looked up in the ISO C++ standard.

match-expression:
match-declaration

match-id:
`%`
nondigit
match-id `%`
match-id nondigit
match-id digit

match-declaration:
match-decl-specifier-seq$_{opt}$ *match-declarator*

match-decl-specifier-seq:
match-decl-specifier-seq$_{opt}$ *match-decl-specifier*

match-decl-specifier:
nested-match-name-specifier$_{opt}$ *match-id*
cv-qualifier
match-function-specifier
`char`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

match-function-specifier:
`virtual`
`static`

nested-match-name-specifier:
match-id `::` *nested-match-name-specifier*$_{opt}$
`...` `::` *nested-match-name-specifier*$_{opt}$

match-declarator:
direct-match-declarator
match-ptr-declarator match-declarator

abstract-match-declarator:
direct-abstract-match-declarator
match-ptr-declarator abstract-match-declarator

direct-match-declarator:
match-declarator-id
direct-match-declarator `(` *match-parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
direct-match-declarator `[` *match-array-size* `]`

direct-abstract-match-declarator:
direct-abstract-match-declarator `(` *match-parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
direct-abstract-match-declarator `[` *match-array-size* `]`

match-array-size:
`%`
decimal-literal

match-ptr-operator:
`*` *cv-qualifier-seq*$_{opt}$
`&`
nested-match-name-specifier `*` *cv-qualifier-seq*$_{opt}$

match-parameter-declaration-clause:
`...`
match-parameter-declaration-list$_{opt}$
match-parameter-declaration-list `, ...`

match-parameter-declaration-list:
match-parameter-declaration
match-parameter-declaration-list `,` *match-parameter-declaration*

match-parameter-declaration:
matct-decl-specifier-seq match-abstract-declarator$_{opt}$

match-declarator-id:
nested-match-name-specifier$_{opt}$ *match-id*
nested-match-name-specifier$_{opt}$ *match-operator-function-id*
nested-match-name-specifier$_{opt}$ *match-conversion-function-id*

match-operator-function-id:
`operator %`
`operator` *match-operator*

match-operator: one of

```
new  delete   new[]   delete[]
+    -    *    /    %%   ^    &      |    ~    !    =    <    >
+=   -=   *=   /=   %%=  ^=   &=     |=   <<   >>   >>=  <<=  ==
!=   <=   >=   &&   ||   ++   --     ,    .*   ->*  ->   ()   []
?:
```

match-conversion-function-id:

`operator` *match-conversion-type-id*

match-conversion-type-id:

match-type-specifier-seq match-conversion-declarator$_{opt}$

match-conversion-declarator:

match-ptr-operator match-conversion-declarator$_{opt}$

C Structure Of The Project Repository

Figure 3 shows the internal structure of the AspectC++ model and the AspectC++ project repository. The distinction between name and code join points and also the inheritance hierarchy is visible.

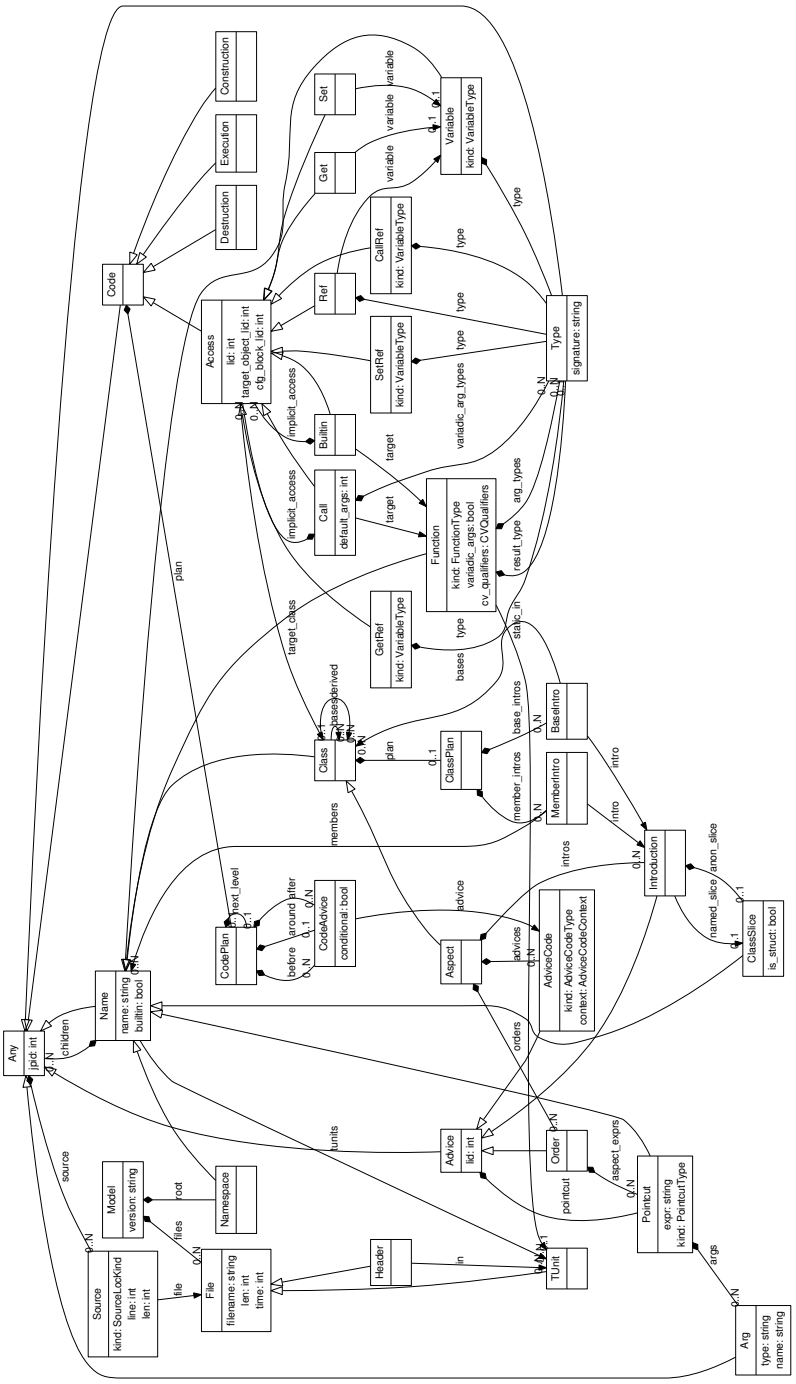

Figure 3: Structure of the AspectC++ project repository

D Project Repository File For Example on page 7

```
<?xml version="1.0"?>
<ac-model version="1.2" ids="7">
  <files>
```

```
  <TUnit filename="shape.cpp" len="42" time="1442951698" id="0"/>
</files>
<root>
  <Namespace name="::">
    <children>
      <Class name="Shape" id="1">
        <children>
          <Function kind="8" cv_qualifiers="0" name="~Shape" builtin="true">
            <children>
              <Destruction/>
            </children>
          </Function>
          <Function kind="7" cv_qualifiers="0" name="Shape" builtin="true">
            <children>
              <Construction/>
            </children>
          </Function>
          <Function kind="7" cv_qualifiers="0" name="Shape" builtin="true">
            <arg_types>
              <Type signature="const Shape &amp;amp;"/>
            </arg_types>
            <children>
              <Construction/>
            </children>
          </Function>
        </children>
        <source>
          <Source kind="1" file="0" line="1" len="1"/>
          <Source kind="2" file="0" line="1" len="1"/>
        </source>
      </Class>
      <Namespace name="Circle">
        <children>
          <Class bases="1" name="S_Circle" id="4">
            <children>
              <Function kind="7" cv_qualifiers="0" name="S_Circle" builtin="true">
                <children>
                  <Construction/>
                </children>
              </Function>
              <Function kind="7" cv_qualifiers="0" name="S_Circle" builtin="true">
                <arg_types>
                  <Type signature="const Circle::S_Circle &amp;amp;"/>
                </arg_types>
                <children>
                  <Construction/>
                </children>
              </Function>
              <Variable kind="3" name="m_radius">
                <type>
                  <Type signature="int"/>
                </type>
                <source>
                  <Source kind="1" file="0" line="8" len="1"/>
                </source>
              </Variable>
              <Function kind="3" cv_qualifiers="0" name="radius" id="3">
                <result_type>
                  <Type signature="void"/>
```

```
            </result_type>
            <arg_types>
              <Type signature="int"/>
            </arg_types>
            <children>
              <Execution/>
              <Builtin target="2" lid="0">
                <source>
                  <Source kind="0" file="0" line="11" len="1"/>
                </source>
              </Builtin>
            </children>
            <source>
              <Source kind="1" file="0" line="10" len="3"/>
            </source>
          </Function>
          <Function kind="8" cv_qualifiers="0" name="~S_Circle">
            <children>
              <Destruction/>
            </children>
            <source>
              <Source kind="1" file="0" line="13" len="1"/>
            </source>
          </Function>
        </children>
        <source>
          <Source kind="1" file="0" line="7" len="8"/>
          <Source kind="2" file="0" line="7" len="1"/>
        </source>
      </Class>
      <Function kind="1" cv_qualifiers="0" name="draw" id="6">
        <result_type>
          <Type signature="void"/>
        </result_type>
        <arg_types>
          <Type signature="int"/>
        </arg_types>
        <children>
          <Execution/>
          <Call target="3" lid="0" target_class="4">
            <source>
              <Source kind="0" file="0" line="18" len="1"/>
            </source>
          </Call>
          <Call target="5" lid="1">
            <source>
              <Source kind="0" file="0" line="19" len="1"/>
            </source>
          </Call>
        </children>
        <source>
          <Source kind="1" file="0" line="16" len="5"/>
        </source>
      </Function>
    </children>
    <source>
      <Source kind="0" file="0" line="4" len="18"/>
    </source>
  </Namespace>
```

```
        <Function kind="1" cv_qualifiers="0" name="draw" id="5">
          <result_type>
            <Type signature="void"/>
          </result_type>
          <arg_types>
            <Type signature="Shape &amp;amp;"/>
          </arg_types>
          <children>
            <Execution/>
          </children>
          <source>
            <Source kind="1" file="0" line="2" len="1"/>
          </source>
        </Function>
        <Function kind="1" cv_qualifiers="0" name="operator =" builtin="true" tunits="0"
             id="2">
          <result_type>
            <Type signature="int &amp;amp;"/>
          </result_type>
          <arg_types>
            <Type signature="int &amp;amp;"/>
            <Type signature="int"/>
          </arg_types>
        </Function>
        <Function kind="1" cv_qualifiers="0" name="main">
          <result_type>
            <Type signature="int"/>
          </result_type>
          <children>
            <Execution/>
            <Call target="6" lid="0">
              <source>
                <Source kind="0" file="0" line="24" len="1"/>
              </source>
            </Call>
          </children>
          <source>
            <Source kind="1" file="0" line="23" len="4"/>
          </source>
        </Function>
      </children>
    </Namespace>
  </root>
</ac-model>
```

List of Examples

Index