AspectC++ – A Language Overview

©2005 Olaf Spinczyk <os@aspectc.org>
Friedrich-Alexander University Erlangen-Nuremberg
Computer Science 4

May 20, 2005

This is an overview about the AspectC++ language, an aspect-oriented extension for C/C++, prepared for the European AOSD Network of Excellence. Detailed information on AspectC++ including manuals and tutorial slides with lots of examples is available from *the AspectC++ Project Homepage*.

The overview is based on the documentation for AspectC++ 0.9.3. The language is still “work in progress”.

Joinpoint Model and Pointcut Language

AspectC++ supports *static* joinpoints as well as *dynamic* joinpoints. While static joinpoints are named entities in the static program structure, dynamic joinpoints are events that happen during the program execution.

Static Joinpoints

The following kinds of C++ program entities are considered as static joinpoints:

- classes, structs, and unions
- namespaces
- all kinds of functions (member, non-member, operator, conversion, etc.)

Static joinpoints are described by *match expressions*. For example, `"% ...::foo(...)"` is a match expression that describes all functions called `foo` (in any scope, with any argument list, and any result type). More information on match expressions is given below. Note that not all of these static joinpoint types are currently supported as a target of advice (see section ’Advice for Static Joinpoints’).

Dynamic Joinpoints

The following kinds of events that might happen during the execution of a program are considered as dynamic joinpoints:

- function call

- function execution
- object construction
- object destruction

The description of dynamic joinpoints is based upon the description mechanism for static joinpoints in conjunction with joinpoint type specific *pointcut functions*. For example:

- ***call***`("% ...::foo(...)")`
- ***execution***`("float MathFuncs::%(float)")`
- ***construction***`("SomeClassName")`
- ***destruction***`("A"||"B")`

While `"% ...::foo(...)"` represents a set of static joinpoints, i.e. all functions called `foo`, the expression `call("% ...::foo(...)")` describes all calls to these functions. A similar mapping from static to dynamic joinpoints is done by the `execution()`, `construction()`, and `destruction()` pointcut functions.

Pointcut Functions

Further pointcut functions are used to filter or select joinpoints with specific properties. Some of them can be evaluated at compile time while others yield conditions that have to be checked at run time:

- ***cflow***(*pointcut*) – captures all joinpoints in the dynamic execution context of the joinpoints in *pointcut*.
- ***base***(*pointcut*) and ***derived***(*pointcut*) – yield classes based on queries in the class hierarchy
- ***within***(*pointcut*) – filters joinpoints depending on their lexical scope
- ***that***(*type pattern*), ***target***(*type pattern*), ***result***(*type pattern*), and ***args***(*type pattern*) – filters joinpoints depending on the current object type, the target object type in a call, and the result and arguments types of a dynamic joinpoint.
- **&&**, **||**, **!** – intersection, union, and exclusion of joinpoints in pointcuts

Instead of the *type pattern* it is also possible to pass the name of a *context variable* to which context information from the joinpoint shall be bound. In this case the type of the context variable is used for the type matching.

Match Mechanism Capabilities

The match mechanism provides `%` and `...` as wildcard symbols. Thereby the following features are supported:

- pattern based name matching, e.g. `"%X%"` matches all names that contain an uppercase `X`
- flexible scope matching, e.g. `"Foo::...::Bar"` matches Bar in the scope `Foo` or any of its nested scopes
- flexible matching of function argument type lists, e.g. `"% foo(...,int)"` matches `foo` with an `int` as its last argument type
- matching template argument lists, e.g. `"C<T,...>"` matches an instance of the class template `C` with a first template argument type `T`
- type patterns, e.g. `"const % *"` matches all pointer types that reference objects of a constant type

Named Pointcuts

Pointcut expressions can be given a name. The definition of a named pointcut can be placed in any aspect, class, or namespace. The mechanism can be used for dynamic as well as static joinpoints. For example:

```
class OStream {
  // ...
  pointcut manipulator_types() = "hex"||"oct"||"bin"||"endl";
};
```

In the context of an aspect, named pointcuts can also be defined as virtual or pure virtual, which allows refinement/definition in a derived aspect (see example at the end of this overview).

Advice Model and Language

Advice for Static Joinpoints

The only kind of advice for static joinpoints that is currently supported by AspectC++ is the *introduction*. By using this kind of advice the aspect code is able to add new elements to classes, structures, or unions. Everything that is syntactically permitted in the body of a C++ class can be introduced by advice:

- attribute introduction, e.g. `advice "AClass" : int _introduced_attribute;`
- type introduction, e.g. `advice "AClass" : typedef int INT;`
- member function introduction, e.g. `advice "AClass" : void f();`
- nested type introduction, e.g. `advice "AClass" : class Inner { ... };`
- constructor introduction, e.g. `advice "AClass" : AspectName(int, double);`

- destructor introduction, e.g. `advice "AClass" : ~AspectName() { ... }`
- base class introduction, e.g. `advice "AClass" : baseclass(ANewBaseclass);`

The syntax `advice <`***`target-pointcut`***`> : <`***`introduction`***`>` supports the introduction of an element into an arbitrary set of target classes with a single advice.

Advice for Dynamic Joinpoints

Advice for dynamic joinpoints is used to affect the flow of control, when the joinpoint is reached. The following kinds of advice are supported:

- before advice
- after advice
- around advice

These advice types can orthogonally be combined with all dynamic joinpoint types. Advice for dynamic joinpoints is defined with the following syntax:

```
advice <target-pointcut> : (before|after|around) (<arguments>) {
  <advice-body>
}
```

While the before and after advice bodies are executed before or after the event described by <***`target-pointcut`***>, an around advice body is executed instead of the event.

Advice Language and Joinpoint API

The advice body is implemented in standard C++. Additionally, the *joinpoint API* can be used to access (read and write) *context information* (e.g. function argument and result values) as well as *static type information* about the current joinpoint. To access the joinpoint API the object `JoinPoint *tjp` can be used, which is implicitly available in advice code. Advice that uses the static type information provided by the joinpoint API is called *generic advice*. This concept is the key for generic, type-safe, *and* efficient advice in AspectC++. The static type information from the joinpoint API can even be used to instantiate template metaprograms, which is a technique for joinpoint specific code generation at compile time.

The joinpoint API is also used to proceed the execution from within around advice (`tjp->proceed()`). Alternatively, `tjp->action()` may be called to retrieve and store the *proceed context* as an `AC::Action` object. Later, `action.trigger()` may be called to proceed the intercepted flow of control.

Catching and changing exceptions can be done by standard C++ features in around advice (try, catch, throw).

Example

The following advice is *generic advice*, because its implementation can deal with multiple overloaded `C::foo(...)` implementations that have different result types:

```
advice execution("% C::foo(...)") : around() {
  cout << "executing " << JoinPoint::signature()
       << " on " << *tjp->that() << endl;
  tjp->proceed ();
  cout << "the result was " << *tjp->result() << endl;
}
```

Aspect Module Model

The following example code shows an aspect `Logging` defined in AspectC++:

```
aspect Logging {
  ostream *_out; // ordinary attributes
public:
  void bind_stream(ostream *o) { _out = o; } // member function
  pointcut virtual logged_classes() = 0; // pure virtual pointcut
  // some advice
  advice execution("% ...::%(...)") && within(logged_classes()) :
    before () {
    *_out << "executing " << JoinPoint::signature () << endl;
  }
};
```

Aspects are the language element that is used to group all the definitions that are needed to implement a crosscutting concern. An aspect definition is allowed to contain member functions, attributes, nested classes, named pointcuts, etc. as ordinary classes. Additionally, aspects normally contain advice definitions.

Aspects that contain pure virtual member functions or pure virtual pointcuts are called *abstract aspects*. These aspects only affect the system if a (concrete) aspect is derived, which defines an implementation for the pure virtual functions and the pure virtual pointcuts. Abstract aspects are the AspectC++ mechanism to implement reusable aspect code.

Aspect inheritance is slighly restricted. Aspects can inherit from ordinary classes and abstract aspects but not from concrete aspects. Derived aspects can redefine virtual pointcuts and virtual functions defined by base aspects.

Aspect Instantiation Model

By default, aspects are singletons, i.e. there is one global instance automatically created for each non-abstract aspect in the program. However, by defining the `aspectof()` static member function of an aspect the user can implement arbitrary instantiation schemes, such as *per-target*, *per-thread*, or *per-joinpoint*. For each dynamic joinpoint that is affected by the aspect the `aspectof()` function has to return the right aspect instance on which the advice bodies are invoked. The instances themselfs are usually created with the introduction mechanism. The `aspectof()` function has access to the joinpoint API in order to find the right aspect instance for the current joinpoint. Here is an example:

```
aspect InstancePerTargetAspect {
  pointcut target_class() = "TargetName";
  // an attribute of the aspect (stored per target)
  int _calls;
  // aspect instance created by introduction:
  advice target_class() : InstancePerTargetAspect _instance;
  // definition of the instantiation scheme in aspectof():
  static InstancePerTargetAspect *aspectof() {
    return tjp->target()->_instance;
  }
  // the advice
  advice call("% ...::%(...)") && target(target_class()) : before () {
    _calls++;
  }
  // attribute initialization in the aspect constructor
  InstancePerTargetAspect () : _calls (0) {}
};
```

Aspect Composition Model

In AspectC++ any number of aspects can be used in the same application. Aspect composition is currently restricted to inheritance from abstract aspects. Concrete aspects cannot be used for the implementation of new aspects.

Aspect interactions can be handled by the developer on a per joinpoint basis with *order advice*, e.g. `advice execution("void f%(...)") : order("Me", !"Me")` gives the aspect `Me` the highest precedence at all joinpoint described by the pointcut expression. Order advice can be given by any aspect for any aspect, thus can be separated from the affected aspects. Aspect names in order advice declarations are match expressions and may contain wildcards, e.g. `order("kernel::%", !"kernel::%")` gives every aspect declared in the namespace `kernel` precedence over all other aspects. Besides this partial order in the precedence of aspects, the precedence of advice within one aspect is determined according to its position in the aspect definition. As long as it does not conflict with order advice AspectC++ aims to give aspects the same precedence at all joinpoints.

Aspect Weaving Model

The only implementation of AspectC++ is based on static source to source transformation. Nevertheless, the language could also be used for dynamic weavers if some really hard technical challenges like runtime introductions were solved.

Example

A reusable observer pattern implementation in AspectC++:

```
aspect ObserverPattern {
  // Data structures to manage subjects and observers
  ...
  public:
  // Interfaces for each role
  struct ISubject {};
  struct IObserver {
    virtual void update(ISubject *) = 0;
  };

  // To be defined by the concrete derived aspect
  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;
  // subjectChange() matches all non-const methods by default
  pointcut virtual subjectChange() =
    execution("% ...::%(...)" && !"% ...::%(...) const")
                              && within(subjects());

  // Add new baseclass to each subject/observer class
  // and insert code to inform observers
  advice observers() : baseclass(IObserver);
  advice subjects() : baseclass(ISubject);
  advice subjectChange() : after() {
    ISubject* subject = tjp->that();
    updateObservers(subject);
  }

  // Operations to add, remove and notify observers
  void updateObservers(ISubject* sub) { ... }
  void addObserver(ISubject* sub, IObserver* ob) { ... }
  void remObserver(ISubject* sub, IObserver* ob) { ... }
};
```

Concrete aspect, which applies the pattern:

```
#include "ObserverPattern.ah"
#include "ClockTimer.h"

aspect ClockObserver : public ObserverPattern {
  // define the pointcuts
  pointcut subjects()   = "ClockTimer";
  pointcut observers()  = "DigitalClock"||"AnalogClock";
public:
  advice observers() :
    void update( ObserverPattern::ISubject* sub ) {
    Draw();
  }
};
```