

# AspectC++

## **An Aspect-Oriented Extension to C++**

Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat

TOOLS Pacific

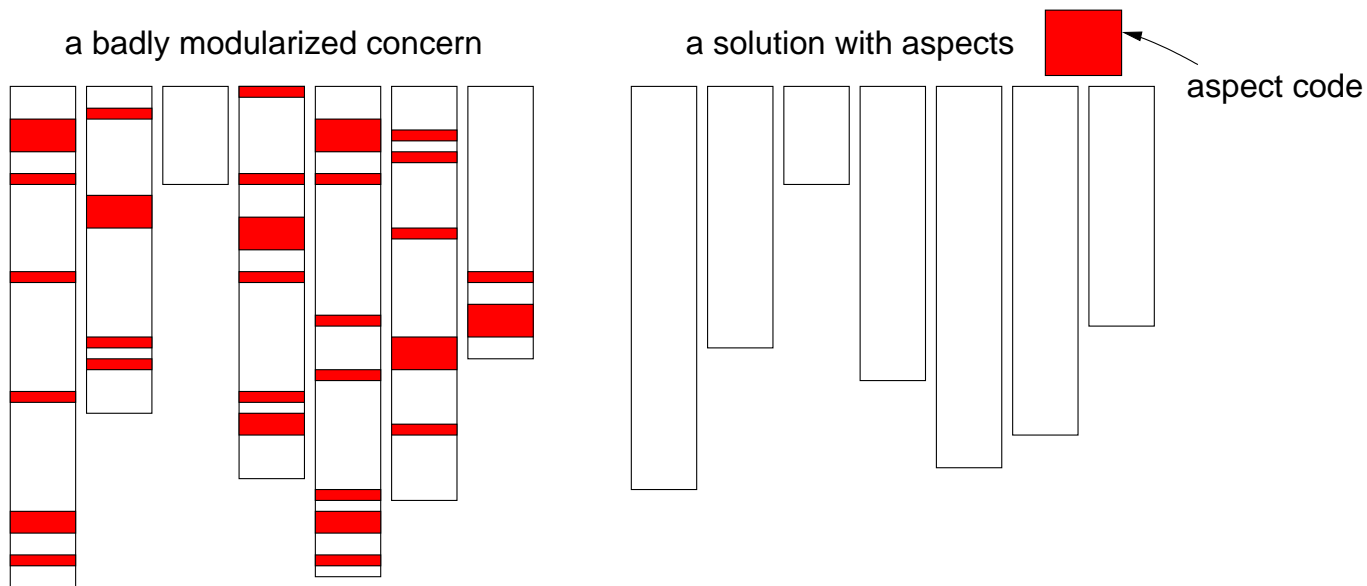
February 19th, 2002

## Outline

- Motivation
- Language Introduction
- Applications
- Implementation
- Future Work
- Conclusions

## Motivation

- Aspect-oriented programming allows a modular implementation of “crosscutting concerns”



- Aspects can contribute ...
  - ★ additional activities to the control flow
  - ★ attributes, functions, base classes, etc. to the static program structure

## Language Introduction (1)

- Aspects are a language element to implement a crosscutting concern in a modular way

```
aspect MyAspect  
{  
};
```

## Language Introduction (2)

- Aspects can do everything that classes can ...

```
aspect MyAspect : public ABaseClass
{
    int anAttribute;
    void aMemberFunction();
};
```

## Language Introduction (3)

- Aspects can define *advice*...

```
aspect MyAspect
{
    advice execution("void AClass::aFunction()"):
    before ()
        { cout << "AClass::aFunction running" << endl; }
};
```

**before**, **after** and **around** advice can add behavior to certain points in the control flow, which are defined by a pointcut.

## Language Introduction (4)

- Aspects have reflective access to information about the *join point*...

```
aspect MyAspect
{
    advice execution("% AClass::%(...)"): before()
    { cout << thisJoinPoint->toString() << endl; }
};
```

**thisJoinPoint** provides a join point signature, the types and values of arguments and result, the called and calling object, ...

## Language Introduction (5)

- Aspects can (re-)use *pointcut* definitions...

```
aspect MyAspect
{
    pointcut virtual methods() =
        execution("% AClass::%(...)");
    advice methods(): before() { do_before(); }
    advice methods(): after() { do_after(); }
};
```

A **pointcut** defines a set of join points, it can be 'virtual' and 'pure virtual', defined and redefined like a function...

## Language Introduction (6)

- Aspects can use *introductions* to add new members and baseclasses...

```
aspect MyAspect
{
    pointcut targetClasses() = classes("AClass");
    advice targetClasses(): int aNewMember;
    advice targetClasses(): void aFunction();
    advice targetClasses(): baseclass(aBase);
};
```

Introductions (a special kind of **advice**) manipulate the static program structure

## Language Introduction (7)

- Pointcuts are the key language element to deal with crosscutting concerns...

Match Expressions	
Types	"int", "Chain", "Memory%"
Attributes	"Chain* Chain::next", "% State::%"
Functions	"void reset()", "% printf(...)", "void %(int,%)"

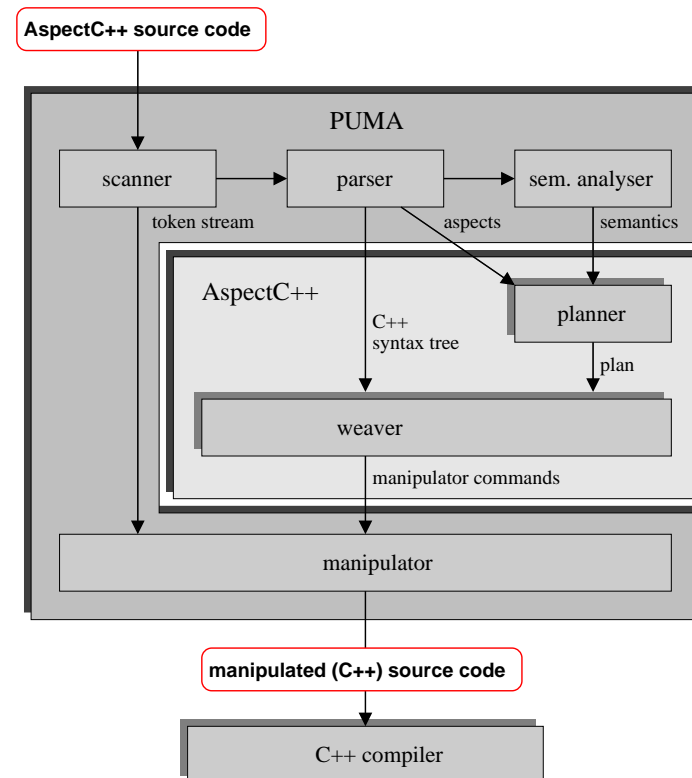
Pointcut Functions	
Functions	<i>call(pointcut), execution(pointcut), callsto(pointcut)</i>
Attributes	<i>get(pointcut), set(pointcut)</i>
Types	<i>classes(pointcut), derived(pointcut), base(pointcut)</i>
Scope	<i>within(pointcut)</i>
Control Flow	<i>cflow(pointcut), reachable(pointcut)</i>
Context	<i>that(pointcut or name), target(pointcut or name), args(...)</i>
Algebraic	<i>  , &amp;&amp;, !</i>

## Applications

- The language semantics is close to AspectJ
  - ★ see **[www.aspectj.org](http://www.aspectj.org)**
  - ★ Simple “trace” aspects
  - ★ New (implementations of) design patterns
  - ★ Multi-threading and synchronization
  - ★ ... many more crosscutting concerns
- Low-end mobile and embedded systems: crosscutting concerns *and* resource restrictions
  - ★ Program instrumentation with monitoring code
  - ★ Run-time surveillance
  - ★ Fault tolerance
  - ★ Distribution

## Implementation (1)

- **ac++** is a preprocessor
  - ★ use your favorite C++ compiler
  - ★ code generation “on demand”
  - ★  $\Rightarrow$  no overhead
- Problems: C++ is not Java
  - ★ attribute access via pointers is hard to detect
  - ★ Classes/Aspects must be defined before they are used
- Current C++ parser limitations
  - ★ no namespaces
  - ★ only limited support for templates



## Implementation (2)

- Really no overhead?

```
aspect Trace
{
    pointcut methods() = execution("% AClass::%(...)");

    advice methods(): void before()
    {
        cout << "exec " << thisJoinPoint->toString();
    }
};
```

- How big is a **JoinPoint** object?

## Implementation (3)

```
class Trace
{ struct __Tjps__advice_0 { const char *_to_string; };
  void __advice_0 (__Tjps__advice_0 *thisJoinPoint)
    { cout << "exec " << thisJoinPoint->_to_string; }
};

class AClass
{ void __old_foo(int) { /* ... */ }
  void foo (int arg0)
  {
    static Trace::__Tjps__advice_0 __tjp_Trace__advice_0 =
      { "void AClass::foo(int)" } ;
    Trace::aspectOf ()->__advice_0 (&__tjp_Trace__advice_0);
    __old_foo (arg0);
  }
};
```

## Future Work

- One of many ideas: *Actions*

```
advice somePointcut() : void around()  
{ thisJoinPoint->proceed(); /* do the intercepted action */ }
```

- Actions can be: calling/executing a function, setting/getting an attribute
- Aspect code should be able to...
  - ★ store and execute actions (at any time in any order)
  - ★ construct actions from a magic “action ID”
- Applications: re-usable aspects to...
  - ★ manipulate interaction protocols (e.g. elevator algorithm)
  - ★ cache results from idempotent functions
  - ★ customize binding of components (e.g. remote object invocation)

## Conclusions

- AspectC++ is an aspect-oriented extension to C++
- The semantics is very similar to AspectJ<sup>1</sup>
- No run-time support required
  - ★ ideal AOP language for low-end mobile and embedded devices
  - ★  $\Rightarrow$  most of the users on our mailing-list have non-academic addresses
- Research with AspectC++ has just begun. . .
  - ★ distribution, fault-tolerance, real-time aspect

---

<sup>1</sup>AspectJ is a trademark of Xerox corporation.

**The End**

- You are invited:

<http://www.aspectc.org/>