# AspectC++

ANDREAS GAL, OLAF SPINCZYK, AND WOLFGANG SCHRÖDER-PREIKSCHAT
University of Erlangen-Nürnberg, Informatik 4, Martensstrasse 1, 91058 Erlangen, Germany

ABSTRACT

Small embedded systems are forced to operate under extreme resource constraints. At the same time these systems are very complex and many concerns in the implementation of such systems are of highly crosscutting nature. The concept of aspect orientation can be applied to allow a modular implementation of these concerns. With AspectJ[a] the first complete and powerful language extension for aspect-oriented programming (AOP) has been created. However, the costs of the Java run-time environment are not feasible for most embedded applications.
This poster presents AspectC++ [1], a new language extension to C/C++ that provides powerful language elements to facilitate aspect-oriented programming even in domains with tight resource limitations.

[a] AspectJ is a trademark of Xerox Corporation.

## 1 What is AspectC++?

### 1.1 Grammar

One of the main design goals for the AspectC++ language was to keep the C++ grammar extension simple. This decision was motivated by the fact that C++ is much harder to parse than, for instance, Java due the complexity of the language and ambiguities in the grammar. The AspectC++ grammer we are using is relatively weak and requires a semantic post-processing. Match expressions for type names are specified as strings and pointcut functions follow the C++ expression syntax.

aspect-name:
    identifier

declaration:
    pointcut-declaration
    advice-declaration

class-head:
    *aspect* identifier$_{opt}$ base-clause$_{opt}$

pointcut:
    constant-expression

member-declaration:
    abc-declaration
    pointcut-declaration
    advice-declaration

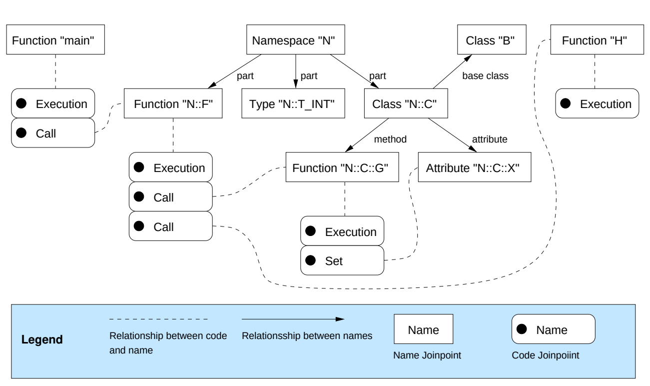
abc-declaration:
    *alters* pointcut
    *observes* pointcut

pointcut-declaration:
    *pointcut* declaration

advice-declaration:
    *advice* pointcut : declaration

### 1.2 Joinpoint Model

While AspectJ explicitly distinguishes between **pointcuts and advice** on the one hand and **Generic Type Names (GTNs) and introductions** on the other, AspectC++ handles both in a uniform way. The result is a **typed joinpoint model**. The major advantage of this model is that sets of classes, namespace, or functions can be named. With virtual name pointcuts they can even be used to parameterize reusable aspects.




### 1.3 Joinpoint Types

AspectC++ uses typed joinpoints. The major categories are **code joinpoints** and **name joinpoints**.

| Symbol | Type | Category | Symbol | Type | Category |
|---|---|---|---|---|---|
| C | any | code | N | any | name |
| $C_C$ | Call | code | $N_N$ | Namespace | name |
| $C_E$ | Execution | code | $N_C$ | Class | name |
| $C_S$ | Set | code | $N_F$ | Function | name |
| $C_G$ | Get | code | $N_T$ | Type | name |

### 1.3 Pointcut Functions

The AspectC++ pointcut functions are similar to those found in AspectJ, but have to take the **type** of the joinpoints into account.

| Name | Type | Result |
|---|---|---|
| call | $N \rightarrow C_C$ | function calls |
| execution | $N \rightarrow C_E$ | function bodies |
| set | $N \rightarrow C_S$ | attribute/global object access (write) |
| get | $N \rightarrow C_G$ | attribute/global object access (read) |
| within | $N \rightarrow C$ | all code join points within a function/class, … |
| that | $N \rightarrow C$ | affected object (`this`) is instance of the specified class |
| target | $N \rightarrow C$ | target object (for functions calls for example) is instance of the specified class |
| args | $(N,...) \rightarrow C$ | join points where arguments are of the specified type(s) |
| result | $N \rightarrow C$ | result is of the specified type |
| cflow | $N \rightarrow C$ | all dynamic code join points occuring after entering and before leaving the specified function(s) |
| base | $N \rightarrow N_{C,F}$ | all base classes (respectively redefined methods) |
| derived | $N \rightarrow N_{C,F}$ | all derived classes (respectively redefined methods from derived classes) |
| && | $(N,N) \rightarrow N$, $(C,C) \rightarrow C$ | intersection |
| \|\| | $(N,N) \rightarrow N$, $(C,C) \rightarrow C$ | set union |
| ! | $N \rightarrow N$, $C \rightarrow C$ | complement (of the same type) |

## 2 Example

### 2.1 An Off-The-Shelf Component

The AspectC++ syntax can be explained best using a simple example. As basis we use a standard quicksort algorithm as it can be found in many course books. In this example, we want parallelize the quicksort algorithm to speed up the execution on multi-processor machines. AspectC++ allows to leave the quicksort implementation unchanged. The component code continues to contain solely the functional part of the program, i.e. the quicksort algorithm. An additional aspect encapsulates the code to realize the crosscutting concern of parallelism.

```
void quicksort(int *a, int lo, int hi ) {
    int h, l, p, t;

    if (lo < hi) {
        l = lo; h = hi; p = a[hi];
        do {
            while ((l < h) && (a[l] <= p)) l = l+1;
            while ((h > l) && (a[h] >= p)) h = h-1;
            if (l < h) {
                t = a[l]; a[l] = a[h]; a[h] = t;
            }
        } while (l < h);
        t = a[l]; a[l] = a[hi]; a[hi] = t;
        quicksort( a, lo, l-1 );
        quicksort( a, l+1, hi );
    }
}
```

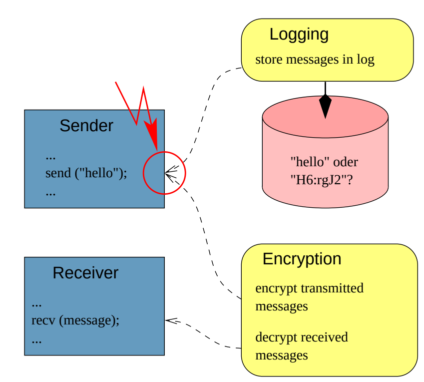
### 2.2 A Multi-Threading Aspect

The **Threading** aspect encapsulates completely the code necessary to parallelize the quicksort code using the **pthread** library.

```
#include <pthread.h>
class SortJob {
    ...
    void run () { pthread_create (...); }
    void wait () { pthread_join (...); }
};
aspect Threading {
    int depth, no; bool threaded;
    SortJob *job[1 << FORK_DEPTH];
    Threading () : depth (0), threaded (false), no (0) {}
    advice execution("void quicksort(...)") && args (array, l, h) :
    around (int *array, int l, int h) {
        if (threaded) { thisJoinPoint->proceed (); return; }
        if (depth == FORK_DEPTH)
            job[no++] = new SortJob (array, l ,h);
        else { depth++; thisJoinPoint->proceed (); depth--; }
        if (depth == 0) {
            threaded = true;
            for (int r = 0; r < no; r++) job[r]->run ();
            for (int t = 0; t < no; t++) job[t]->wait ();
            for (int d = 0; d < no; d++) delete job[d];
            threaded = false; no = 0;
        }
    }
};
```

## 3 Features

AspectC++ introduces a number of novell features to support large and highly configurable systems with many, potentially conflicting aspects.




### 3.1 Build Management

The build process can be streamlined by specifying explicitly the source code modules affected by an aspect.

```
#pragma weave "Sender.cc" "Receiver.cc"
aspect Encrypt {
    ...
};
```

### 3.2 Aspect Behavior Contracts

Aspect behavior contracts describe how an aspect interacts with component code. This information can be used to reduce the testing and verification effort in configurable or evolving systems.

```
aspect Encrypt {
    alters "Sender" || "Receiver";
    ...
};
aspect Logging {
    observes "Sender" || "Receiver";
    ...
};
```

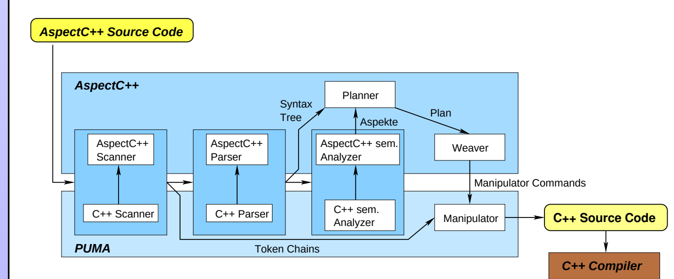
### 3.3 Aspect Conflict Resultion

Based on behavior contracts the compiler can detect conflict situations between aspects. The compiler enforces that the programmer specifies a strict ordering for all these conflict cases. In contrast to the **aspect dominance** concept the ordering is specified separate from the conflicting aspects. This increases reusability of aspect code and allows to specify different orderings for subsystems.

```
aspect EncryptAndLog {
    advice within("Confidential%") :
        order("Encryption", "Logging");
    advice !within("Confidential%") :
        order("Logging", "Encryption");
};
aspect AlwaysEncrypt {
    advice within("%") :
        order("Encryption", "Logging");
};
```

## 4 Implementation

AspectC++ is implemented as a preprocessor that takes AspectC++ source code as input and generates pure C++ code, which can be compiled with standard C++ compilers. This increases the portability and allows the programmer to continue using his favorite compiler and tool chain.




The AspectC++ compiler implementation is based on the PUMA source code manipulation framework developed at the University of Magdeburg, Germany. Binary executables of the AspectC++ compiler for Windows, Linux, and Solaris are available from the AspectC++ homepage (`http://www.aspectc.org/`). The authors are currently working on a cleanup required for a source code release of the AspectC++ compiler and the PUMA framework.

## 5 Conclusions and Future Work

AspectC++ is an aspect-oriented extension to the C++ language. Originally AspectC++ was closely modeled after AspectJ. In the meantime we have replaced a number of key concepts like the joinpoint model, aspect behavior contracts and aspect conflict resultion with solutions we consider more powerful than those found in other aspect languages. We feel that the proposed mechanisms are generic enough to be considered for inclusion in other aspect languages. For the implementation of the AspectC++ compiler special care has been taken to avoid unnecessary resource requirements. Thus, AspectC++ can be considered a **powerful and efficient** aspect-oriented programming language. This allows to combine the concepts of aspect-orientation and object-orientation even in application domains with tight resource limitations like the area of deeply embedded systems. As far as future work is concerned, we plan to add a generalization of the current proceed mechanism. The envisioned **action mechanism** envisioned would allow aspect code to store, retrieve and delegate the execution of the original code in a joint at will. Initial support for this mechanism in AspectC++ showed very promising results.

## References

[1] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, Sydney, Australia, Feb. 2002.