Aspect-Oriented Extensions to HEP Frameworks

P. Calafiura [*], C. E. Tull [†], LBNL, Berkeley, CA 94720, USA

Abstract

In this paper we will discuss how Aspect-Oriented Programming (AOP) can be used to implement and extend the functionality of HEP architectures in areas such as performance monitoring, constraint checking, debugging and memory management. AOP is the latest evolution in the line of technology for functional decomposition which includes Structured Programming (SP) and Object-Oriented Programming (OOP). In AOP, an Aspect can contribute to the implementation of a number of procedures and objects and is used to capture a concern such as logging, memory allocation or thread synchronization that crosscuts multiple modules and/or types. Since most HEP frameworks are currently implemented in c++, for our study we have used AspectC++, an extension to c++ that allows the use of AOP techniques without adversely affecting software performance. We integrated AspectC++ in the development environment of the Atlas experiment, and we will discuss some of the configuration management issues that may arise in a mixed c++/AspectC++ environment. For each of the concerns we have examined we will discuss how traditional programming techniques compare to the AOP solution we developed. We will conclude discussing the short and medium term feasibility of introducing AOP, and AspectC++ in particular, in the complex software systems of the LHC experiments.

ASPECT ORIENTED PROGRAMMING

Aspect Oriented Programming[1] is a methodology introduced by Xerox PARC in the late 90s to decompose problems into functional components (e.g. Object-Oriented classes) and aspects that crosscut functional components, such as message logging, thread synchronization, execution tracing. AOP goal is to localize the implementation of these crosscutting concerns. This is of particular interest to developers of large software systems, such as HENP frameworks, allowing framework developers to evolve the design of the framework explicit and implied interfaces without disrupting the work of component developers.

AspectC++

There are about ten actively supported tools available for aspect oriented software development. For this study we choose to use AspectC++[2], an aspect oriented extension to c++, modeled on the popular aspectJ language[3]. The AspectC++ compiler, ac++, weaves aspect code into the original c++ sources, and emits standard c++ with a structure suitable for component-based software development. ac++ is an open-source, actively developed and supported project which is near to its first production release[1].

Overview of ac++ syntax

There are several excellent introductions to AOP concepts[4] and to AspectC++ in particular[5]. Here we can only provide a quick overview of ac++ syntax, and introduce its fundamental concepts:

`aspect:` An `aspect` captures a crosscutting concern. Otherwise behaves very much like a c++ class, defining a scope with access control to its data members and methods that can be overloaded or virtual.

```
aspect RedirectOstream:
  virtual public IProperty {
private:
   ISvcLocator* m_locator;
public:
   const MsgStream& log() const;
};
```

`advice` **and** `joinpoint`: An `advice` adds behaviour to an aspect defining what to do `before`, `after`, and `around` certain `joinpoints` in the control flow. A `joinpoint` is the locus of the code where a certain action occurs (a method is called, a function is executed, a variable is set,...).

```
aspect RedirectOstream {
 advice
 call("ostream &
       ostream::operator <<(const char*&)") &&
                                     args(what):
 around(const char*& what)
 {
   log() << what;
 }
}; //streamlined example
```

[*] pcalafiura@lbl.gov
[†] cetull@lbl.gov

[1] for this study we used release 0.9pre1

`pointcut` and `JoinPoint` API: A `pointcut` is a set of logically related `joinpoints` that defines when a certain `advice` will run. A pointcut is often `virtual` allowing to abstract the definition of when a certain aspect (for example execution Tracing) will be applied.

```
aspect Trace {
   pointcut virtual methods() = 0;
public:
   advice methods () : void before () {
         printf ("before \"%s\"\n",
                 JoinPoint::signature ());
   }
   advice methods () : void after () {
     printvalue(thisJoinPoint->result(),
                JoinPoint::resulttype());
   }
};
aspect TraceInAlgs : public Trace {
  pointcut methods() =
    execution("% %::%(...)" ) &&
    within(derived("IAlgorithm"));
};
```

The `Trace` aspect above also illustrate part of the rich joinpoint API, which includes static reflection information such as `JoinPoint::signature` or `JoinPoint::resulttype` as well as handles to active objects such as caller and called objects (`this` and `that`), function parameters (`arg`) and return values (`result`).

Introductions: An aspect can introduce in c++ classes new methods, data members and even base classes. For example one can force a set of classes defined in the pointcut `mapkeys` to inherit from the class `ThreadNo`:

```
pointcut mapkeys()="String || UnsignedLong";
aspect threadSafe {
private:
  advice mapkeys() : baseclass(ThreadNo);
};
```

EXAMPLES OF HEP CROSSCUTTING CONCERNS

We tried to apply AOP concepts and techniques to our problem domain: large scale HEP software systems. We choose Gaudi[6] as a representative HEP component-based architecture because it has been successfully adopted by several HEP experiments including the Atlas athena framework [7] that we contributed to develop.

Logging

In athena/Gaudi all messaging is supposed to happen via a `MsgStream` class. `MsgStream` allows to classify messages according to their origin and severity level. The underlying `IMessageService` implementation will use this information e.g. to control job verbosity or, in on-line applications, to send messages to the relevant logging processes.

We experimented with a `RedirectOstream` aspect which intercepts all output to the standard library streams `cout` and `cerr`[2] . The aspect can be used in the development stage to flag these illegal outputs, turning one implicit contract requirement for Gaudi components into an explicitly enforced one. It can also be used in production to simply redirect `cout/cerr` outputs to `MsgStream`, using the joinpoint reflection API to add the required origin information to the message.

Interactive Job Configuration

Gaudi allows to invoke a callback function whenever a property attribute of an object is set/modified. This allows for example to modify interactively, from the python prompt, a cut used by a reconstruction algorithm. To support this, Gaudi requires these property attributes to inherit from a `Property` base class. Unfortunately the majority of Gaudi objects does not yet use this `Property`-based attributes, preferring to use plain data types as `long` or `double` as their properties.

We tried to use the `set` pointcut function of ac++ to introduce a callback to properties of arbitrary type using an `after` advice, which would have been activated after the value of the attribute is set or reset. Unfortunately the `set` pointcut function is not yet supported by ac++.

Object History

The athena/Gaudi History mechanism keeps track of which Algorithm component added a data object to the Event Transient Data Store[7]. We experimented with an aspect that extends this mechanism, keeping a complete log of every component that created the object or modified it. The pointcut for this aspect is defined as the union of all non-const methods of a data object class[3] (including its constructors).

Reference Management

Gaudi allows to use plain c++ pointers to access most of the object managed by the framework, such as data object, services, tools. While using pointers keeps the interface efficient and simple to understand, it leaves the ownership of the returned objects ambiguous. The implied contract with the component developers is that they should never take ownership of a data object and that they should always release any tool or service they requested from Gaudi. Using ac++ it is possible to check these contract requirements

[2]Unfortunately the ac++ release we used (0.9) did not allow to specify templated joinpoints. Since in recent c++ standard library implementations (e.g. the one that comes with g++ release 3.x) `cout/cerr` are template instantiations we were forced to compile our aspect using an older non-templated implementation that came with gcc 2.95.

[3]this of course assumes that the implementation of the data object and the codes modifying it are const-correct.

explicitly, returning an error when a data object is accidentally deleted by a client. It is also possible to release all Gaudi tools and services requested by a component when the component is finalized.

Thread-aware Naming Service

All objects in Gaudi are identified by their type[4] and by a user-defined instance name. This scheme had to be extended to allow for multiple identical instances of certain objects (e.g. the Algorithm components) running in multiple threads: a Gaudi helper class is used to attach the a numeric thread identifier to the instance name and to handle it internally. While this works fine, the helper code clutters the original (non MT-aware) name-server code. More importantly it proved to be hard to identify and edit all name-server codes that needed to use the helpers.

In our ac++ solution, we used ac++ insertions to extend the instance identifier inserting the threadID helper as a base class

```
advice mapkeys() : baseclass(ThreadNo);
```

Then, using a carefully crafted pointcut, we instruct the program to use the new MT-aware object identifier where the old one was. While it is still hard to define the pointcut, as we need to locate all joinpoints where the identifier type is used, no Shotgun Surgery[5] is required any more: the changes needed to use the `ThreadNo` helper class are localized in the aspect rather than scattered through all the name server code.

DISCUSSION

Do we Need AOP?

During the course of this study we convinced ourselves that ”Thinking in Aspects” provides a vocabulary, if not yet a paradigm, to identify and design crosscutting concerns. Describing, say, message logging or object lifetime management as crosscutting concerns makes the “contract” of your component architecture more explicit.

At the implementation level, localizing the code related to a crosscutting concern into an aspect has obvious advantages especially in systems with constantly evolving requirements[6]. Localizing the aspect implementation also helps developing new components and especially helps to integrate external codes into a component.

One criticism that has been made about using aspects in large component-based systems is that it makes even more difficult to understand the control flow of an application. While this is true, especially with the tools we have at hand today, similar criticisms were made at various stages in the past about object-oriented encapsulation and message-passing, abstract interfaces and pluggable components. We think that a combination of better development environments and developer's experience will eventually alleviate this problem.

[4]more precisely by a Gaudi-specific numeric type ID

[5]one of Fowler's ”code smells”[8]: a single change in the system interface forces to modify many classes through the project.

[6]which is the least that can be said of HENP systems...

Can we Use AOP?

One of the advantages of using c++ for HENP applications is the language rich feature set[7]. Many aspects can be implemented directly in c++ using techniques such as policy-based design using templated wrappers and namespaces[5]. There are classes of problems, for example object references, where using policy templates is simpler, more expressive and just as flexible as using aspects. Having said that it is obvious that using an aspect oriented language like AspectC++ in general allows to write aspects that are more compact, easier to understand, and much more powerful.

Based on our limited experience, we think that AspectC++ is a very promising language and toolkit. It is open-sourced, actively developed, well designed and specified.

On the other hand we think ac++ is not quite ready to be put in production in large c++ systems like the ones being developed for LHC. The main issue is the insufficient coverage of the c++ standard, in particular the fact that templates can not yet be parsed by ac++. This, for example, prevents developers from specifying joinpoints involving class or function templates. The situation is quickly improving from one release to the next and we trust that by the time ac++ reaches release 1.0, template support will be sufficient.

Another issue, which is also being addressed by the ac++ community, is the lack of documentation: in many cases the only way for us to find an answer to some basic questions was to use the pretty active ac++ users mailing list.

In our experience the ac++ compiler is currently too slow to be used for frequent builds of systems with hundreds (or thousands) of classes. It does not help that there is no real dependency management tool yet, and the only way to make sure a system is in a coherent state after modifying an aspect definition is to trigger a complete rebuild. Once again this is being addressed by the ac++ developers that are looking into integrating their tool with some popular compiler such as g++. When good dependency management will become available we would like to further investigate the issue of physical coupling introduced by aspects. Superficially the coupling issues seem to be similar to the ones created by using class templates but there may be subtler effects that only real-life experience would show.

ACKNOWLEDGMENTS

We would like to thank the ac++ team and community for their support and advice. We would also like to thank

[7]notoriously rich some would say!

Wim Lavrijsen who presented this paper at the conference on our behalf.

REFERENCES

[1] `http://www.aosd.net`

[2] `http://www.aspectc.org`

[3] `http://www.aspectj.org`

[4] Special Issue on AOP, Communications of the ACM, Vol 44, Issue 10, Oct 2001.

[5] AspectC++ Tutorial, AOSD 2004 Lancaster UK 2004. `http://www.aspectc.org/Publications.6.0.html`

[6] M. Cattaneo *et al.* , "Status of the GAUDI event-processing framework", CHEP 2001: Proceedings. Edited by H. S. Chen. Beijing, China, Science Press, 2001. 757p. `http://proj-gaudi.web.cern.ch/proj-gaudi`

[7] C. G. Leggett *et al.* , "Status of the Athena Framework", this Proceedings.

[8] M. Fowler *et al.* , "Refactoring", Addison-Wesley 1999.