

1 Introduction

The aspect oriented programming (AOP) paradigm has become an important topic of conversation in software engineering. It helps programmers in the separation of concerns, focusing on modularization and encapsulation of cross-cutting concerns which exist in many parts of the program.

There are already several products available which focus on aspect oriented programming. Such a product is commonly called an aspect weaver. Among these products is XWeaver, a tool for aspect oriented programming for C/C++ and Java applications. It has some unique features which make it especially interesting for applications that must undergo a qualification process at the level of source code. This is often the case in the embedded control domain or for mission critical software. The unique features I will introduce are non-intrusiveness and customizability. The first feature, non-intrusive, claims that it does not disrupt the structure of existing code and inserts code that complies with the original structure. The second feature, customizability refers to the possibility of tailoring the rules that are used to weave new code into existing code.

I will introduce and explain the technical approach of XWeaver, compare it to other products and shows how to use it in a real world example.

2 Problem Description

Embedded applications are often critical and their code must consequently be subjected to some kind of qualification program that certifies that it has reached some minimal level of quality. Since an aspect weaver is a tool to weave new code into existing code, the question arises as to whether the qualification process should be performed upon the weaver tool or upon the woven (modified) code. The two basic approaches are:

1. The qualification process is performed upon the base code and upon the aspect weaver. It is assumed that the modified code is of sufficient quality.
2. The qualification process is performed on the modified code only. There is no need to qualify the weaver.

The first approach is regarded as impractical because of the difficulty of qualifying an aspect weaver. The second approach places some constraint on the aspect weaver which must be capable of producing modified code that is amenable to qualification. This means that the modified code must satisfy some requirements, namely it must comply with the same coding rules, it must adhere to the same language subset and it must be commented to the same level as manually written code.

Existing aspect weavers such as AspectJ [1] or AspetC++ [2] do not satisfy the above requirements. The most important shortcoming is that they are unable to handle comments. This is an important drawback because in many cases the documentation is directly embedded in the source code and can be automatically processed. If an aspect weaver does not update the comments in the source code, the documentation becomes invalid and this clearly makes the qualification process more expensive.

3 Aspect Oriented Programming

Before giving an overview and describing the architecture of XWeaver, I will introduce a few basic concepts of the AOP paradigm which are common to most products in this field.

Every aspect program uses an aspect language to specifies changes which have to be done upon some base code. The key concepts for such a language are aspects, pointcuts and advices. An aspect is a container which holds advices and pointcuts. It is very much like a C++ class. An aspect describes the changes which are to be done in the base code. Finally, a pointcut describes where to apply such a change. Therefore, every advice must have a link to a pointcut in order to apply the changes at the right location in the base code.

An aspect weaver therefore provides some tools to work with the aspect language. More precisely, it interprets the aspects and modifies the code appropriately. The modification of the code is called weaving, hence the name aspect weavers. The implementation of the aspect language differs significantly between aspect weavers. The concept, how-

ever, remains the same for most of them.

4 XWeaver Approach

The shortcomings of other aspect weavers highlighted in the previous section stem from the fact that they operate upon an abstract representation of the base code. Commonly, the first stage in their processing is the parsing of the base code and the construction of its abstract syntax tree. The code modifications defined by the aspect program are performed upon this abstract form of the base code. A code generator then constructs the modified code by entirely re-generate the base code. This approach allows an aspect weaver to perform sophisticated operations but also destroys valuable information, most notably its comments.

XWeaver takes a different approach in that it operates upon a model of the code that preserves all information in the base code, including formatting, layout and comments. A solution which is capable of providing such a model for C/C++ and Java source code is srcML [3]. The model provided by srcML is XML based. Using an XML-based representation as the starting point for the weaving process has the further advantage of partially decoupling the aspect weaver and the aspect language from the language of the base code.

The use of an XML-based model of the base code suggests the use of XSL [4] as a language to implement the weaver. This, in turn, is only feasible in a simple manner if the aspect program is also written in XML. For this reason, an XML-based aspect language called AspectX was defined. It expresses the aspects that must be woven into the base code by XWeaver.

Figure 1 illustrates the weaving process used by XWeaver.

5 Architecture

The XWeaver package consists of three components - the *compiler*, the *locator* and the *weaver* - and a set of weaving rules. As mentioned earlier, these components interpret aspects written in the AspectX language and make changes to the base code as appropriate. Each component covers a certain functionality in this weaving process. In the next

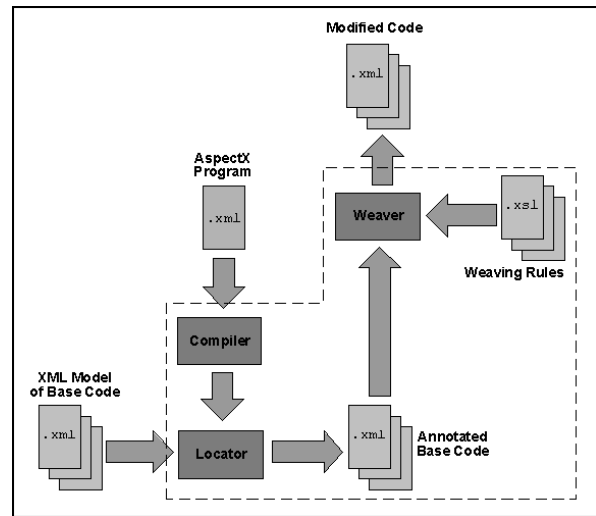


Figure 1: Shows the weaving process of XWeaver

paragraphs, I will describe every component and its functionality.

5.1 Compiler and Locator

The purpose of the compiler is to translate an aspect written in AspectX into an XSL program called the locator. The locator in the next step is concerned with the identification of the locations in the base code where changes are required. It annotates, based on the pointcut informations given in the AspectX program, the base code with code change information at the location where the change may be required. Thus, the annotated base code produced by the locator program contains all the information present in the base code and adds the code change information to it.

5.2 Weaver

As the name suggests, this component is responsible to weave the changes specified by the AspectX program into the base code. The weaver is implemented as an XSL program. In pursuit of the two main goals (extensibility and customizability), the weaving is organized as the application of weaving rules to the annotated base code. The weaver program inspects the annotated base code and, whenever an annotation is found, to determine which

aspect defines the transformation to be performed at that point in the code and then to call the appropriate weaving rule that will insert the new code. XWeaver provides a large set of predefined rules. Extensibility is achieved by allowing users to add new rules that implement new kinds of transformations and customizability is achieved by allowing users to modify the default rules thus allowing them to tailor the way aspects are woven into the base code.

Figure 2 shows all components just described and illustrates the whole weaving process.

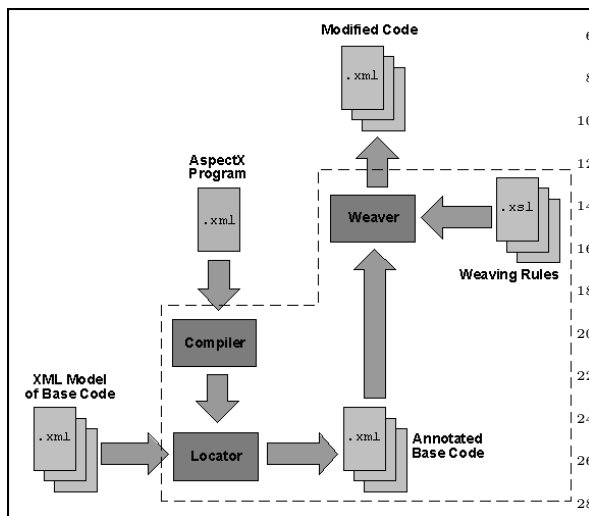


Figure 2: The XWeaver weaving process

6 XWeaver Usage

In this section, we will step through a simple example to show how XWeaver can be used. For our example, we will use C++ as base code. The code is shown in Listing 1. It introduces a very basic data pool which stores a temperature captured by a sensor. This value can be read and written by using the provided methods. Often, such data pools must be made thread safe in the sense that the read and write operations must be locked. The main.cpp file shown in listing 2 shows how our example code is used. As you see in the main function, three threads are created. They concurrently increase and decrease the value in the data pool. After

the loop has finished, every thread reads the actual value, which should be 0. Since the read and write operations are not locked, this can lead to false values. You can prove this by running the code. The code, however, would have to be rewritten in a real world example. Even though, the need to provide thread safe access to a data pool is needed in many applications.

Listing 1: Example base code

```

1  /* Begin Header File */
2  #ifndef DATAPPOOL_H_
3  #define DATAPPOOL_H_
4  class Datapool {
5
6  private:
7      int _temp;
8      int _level;
9
10 public:
11     Datapool();
12     virtual ~Datapool();
13
14     int getLevel();
15     int getTemp();
16     void setLevel( int level_ );
17     void setTemp( int temp_ );
18     void increaseTemp( int interval_ );
19     void decreaseTemp( int interval_ );
20 };
21 #endif /*DATAPPOOL_H_*/
22 /* End Header File*/
23
24 /* Begin Cpp File */
25 #include "Datapool.h"
26
27 Datapool::Datapool() : _temp(0), _level(0) {
28 }
29 Datapool::~Datapool() {
30 }
31 int Datapool::getLevel() {
32     return _temp;
33 }
34 int Datapool::getTemp() {
35     return _temp;
36 }
37 void Datapool::setLevel( int level_ ) {
38     _level = level_;
39 }
40 void Datapool::setTemp( int temp_ ) {
41     _temp = temp_;
42 }
43 void Datapool::increaseTemp( int interval_ ) {
44     int newTemp = getTemp() + interval_;
45     _temp = newTemp;
46 }
47 void Datapool::decreaseTemp( int interval_ ) {
48     int newTemp = getTemp() - interval_;
49     _temp = newTemp;
50 }
51 /* End Cpp File */
  
```

Listing 2: Example main class

```

#include <stdio.h>
#include <pthread.h>
#include "Datapool.h"

#define NLOOP 100000
  
```

```

6  Datapool datapool;
   void *doit(void *);
8
10 int main() {
   pthread_t tidA, tidB, tidC;
   pthread_create(&tidA, NULL, &doit, NULL);
   pthread_create(&tidB, NULL, &doit, NULL);
   pthread_create(&tidC, NULL, &doit, NULL);
14
   // Wait for threads to terminate
   pthread_join(tidA, NULL);
   pthread_join(tidB, NULL);
   pthread_join(tidC, NULL);
   printf("Temperature: %d\n", datapool.getTemp);
20   return 0;
   }
22
   void * doit( void *vptr) {
24     for (int i = 0; i < NLOOP; ++i) {
       datapool.increaseTemp( 1 );
26     }
28
       for (int i = 0; i < NLOOP; ++i) {
30         datapool.decreaseTemp( 1 );
32     }
   }
}

```

We will use XWeaver to weave all the necessary code into this base code to make it thread-safe. The first thing we have to do is to write an aspect for this. To facilitate the user in doing this, XWeaver provides a plugin for Eclipse [5] development environment. The plugin is called AXDT and can be installed using the install mechanism provided by Eclipse. It helps with creating and editing aspects and introduces an eclipse builder which does the weaving process in the background. Furthermore it assists the developer with several views that help to understand the relation between the base code and aspects. A screenshot of the advice editor provided by AXDT is shown in Figure 3. This screenshot also shows the advices which were used for the DATAPOOL example.

The resulting code we want to have as a result of the weaving process is shown in listing 3. In the example, we use POSIX threads to implement the locking mechanism. However, one big advantage of aspect oriented programming is the fact that the implementation could easily be changed by writing another aspect. As you can see, we weave code into the constructor and in all methods which changes the value in the DATAPOOL. Running this code will not lead to false results anymore. Explaining the aspect language used by the XWeaver in detail would go beyond the length of this article. Anyway, if you use the provided eclipse plugin, a detailed knowledge of the language is not necessary because the aspect is automatically created by the AXDT.

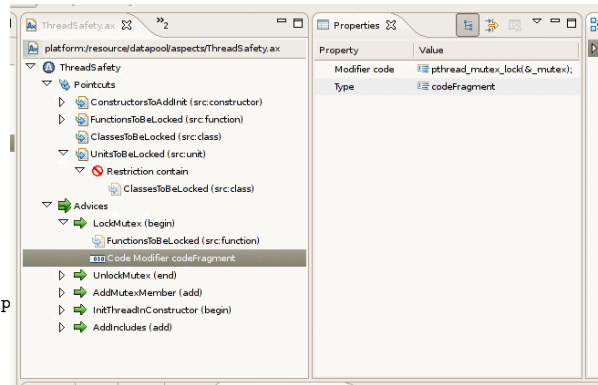


Figure 3: A screenshot of the advice editor

For this reason, I have omitted to show the aspect code.

After the whole aspect (i.e. the advice and the pointcut) has been completed in the AXDT, it can be saved. The weaving process is then automatically started and the woven code is stored in a separate directory. You can inspect and verify the woven code. If you change something in the aspect, the weaving process will only build whatever is necessary by using its built-in incremental build function. Hence, the speed of the build process will be drastically improved.

Listing 3: Resulting code after the weaving process

```

#include "Datapool.h"

Datapool::Datapool() : _temp(0), _level(0) {
    pthread_mutex_init(&_mutex, NULL);
}
Datapool::~Datapool() {
}
int Datapool::getTemp() {
    return _temp;
}
void Datapool::setTemp( int temp_ ) {
    pthread_mutex_lock(&_mutex);
    _temp = temp_;
    pthread_mutex_unlock(&_mutex);
}
void Datapool::increaseTemp( int interval_ ) {
    pthread_mutex_lock(&_mutex);
    int newTemp = getTemp() + interval_;
    _temp = newTemp;
    pthread_mutex_unlock(&_mutex);
}
void Datapool::decreaseTemp( int interval_ ) {
    pthread_mutex_lock(&_mutex);
    int newTemp = getTemp() - interval_;
    _temp = newTemp;
    pthread_mutex_unlock(&_mutex);
}

```

7 Conclusion

As we have seen, XWeaver is a product with some unique features. Most notable is the fact that it preserves all information in the base code, including formatting, layout and comments. Although creating an aspect in XWeaver is facilitated by the provided eclipse plugin, it still needs some knowledge about the AspectX. The installation of XWeaver should not introduce any problems, especially if the eclipse plugin is used. A good documentation of the whole product, including samples, is provided on the product website [6].

To conclude, XWeaver does not provide advanced features such as loading aspects at runtime. If such features are needed, other weavers like AspectJ or AspectC++ are preferable. It does, however, provide some unique features which can be useful in the development of critical applications. Additionally, it supports both, the C/C++ and Java language. Using the introduced architecture, it could even be extended to support other languages.

References

- [1] Eclipse AspectJ Project Team. AspectJ, an aspect-oriented extension to Java. URL <http://eclipse.org/aspectj/>.
- [2] Olaf Spinczyk et al. AspectC++, an aspect-oriented extension to C++. URL <http://www.aspectc.org/>.
- [3] Jonathan Maletic et al. srcML, A translator from C/C++ and Java to srcML. URL <http://www.sdml.info/projects/srcml/>.
- [4] James Clark. XSLT, Extensible Stylesheet Language Transformations. URL <http://www.w3.org/TR/xslt>.
- [5] Eclipse Project Team. The Eclipse development Platform. URL <http://eclipse.org/>.
- [6] P&P Software. XWeaver, An Aspect Weaver. URL <http://pnp-software.com/XWeaver>.