AN ASPECT WEAVER FOR QUALIFIABLE APPLICATIONS

I. Birrer*, P. Chevalley*, A. Pasetti* / **, O. Rohlik***

**ETH-Zentrum, Institut für Automatik, Physikstr. 3, 8092 Zürich, Switzerland*
***P&P Software GmbH,C/O Institut für Automatik, Physikstr. 3, 8092 Zürich, Switzerland*
****ESA-Estec, PO Box 299, Noordwijk, The Netherlands*
ibirrer@aut.ee.ethz.ch, philippe.chevalley@esa.int,
pasetti@pnp-software.com, ibirrer@aut.ee.ethz.ch

Aspect Oriented Programming (AOP) offers means to automatically transform some existing base code to endow it with new properties. An *aspect language* allows the desired transformation to be specified and an *aspect weaver* reads the specification of the transformation expressed in the aspect language and applies it to the base code. AOP techniques allow systematic changes – insertion of synchronization code, of observability code, of instrumentation code, etc – to be applied automatically to a large code base.

The first part of this paper presents the AOP paradigm. The second part presents the XWeaver aspect weaver that is being developed at ETH-Zurich with the cooperation of ESA-Estec. The XWeaver differs from other aspect weavers in being *minimally intrusive.* XWeaver is minimally intrusive in the sense that it gives a high degree of control over the structure of the modified code and it generates code which differs as little as possible from the base code. For these reasons, XWeaver is especially suitable for applications that must undergo a qualification process such as space applications. The target language of XWeaver is C++.

1 Aspect Oriented Programming

The same software application can be looked at from different *perspectives* and each perspective defines a *model* to represent the application. The term *aspect* designates one particular perspective and its associated model. As an example, consider a real-time application. There are at least two obvious perspectives from which such an application can be considered: the *functional perspective* and the *real-time perspective*. The former perspective privileges the description of the algorithms and logic that are implemented by the application. A suitable model for it could be a UML class diagram that shows how the modules making up the application are organized and describing the functional behaviour that each of them implements. The real-time perspective of the application instead privileges its timing-related properties (execution times, timing distribution of its inputs, output deadlines, etc). A model describing this perspective might cover issues such as: the tasks making up the application, their scheduling policies, the synchronization mechanisms that are used to protect shared resources, etc. Each of these two perspectives defines an aspect and a model of the application.

An example of a less obvious type of aspect might be the algorithm optimization policy used in the application. If the application includes computationally intensive segments, it may make sense to define a general "optimization model" that might cover issues like: implementation of matrix operations, handling of sparse data structures, implementation of floating-point operations, etc.

The error handling approach provides still another example of an aspect. If recording of run-time errors is important for an application, a model could be defined that describes which types of errors should be checked for and the actions that should be taken when errors are detected.

It should be stressed that there is no fixed set of aspects that is important for all applications. The aspects outlined above are just examples of potential aspects but, clearly, different applications have different concerns and therefore different sets of applicable aspects. The important point to note that, in order to capture the entire behaviour of an application, it will normally be necessary to consider several aspects. Traditional modelling and programming approaches have privileged the functional aspect but, except for trivial cases, this needs to be complemented with other aspects.

The AOP paradigm was introduced to help handle multiple aspects of an application. More specifically, AOP allows the principle of *separation of concerns* to be applied to all aspects of an application. This principle states that a model of an application should be organized as a set of lower level units where each unit encapsulates one particular feature of the application. The advantage of this approach is that the description of a feature is localized and is therefore more easily controllable. The problem addressed by AOP arises from the fact that application of the principle of separation of concerns to different aspects of the same application typically gives rise to organizations of the associated models that are difficult to map to each other. This is illustrated in figure 1. The figure shows two models (aspects) of the same application. Each model addresses a particular aspect of the application and each model is organized according to the principle of separation of concerns. This means that each model represents the application as a set of lower-level units (the small darker boxes). Since the two models are intended to represent the same application, there must exist some kind of mapping between them. Ideally, one would like this mapping to hold both at the level of the models themselves and at the level of the modular units into which the models are decomposed (i.e. one would like features that are encapsulated in a single modular unit in one model to be mapped to features that are encapsulated in a single modular unit in the other model). Unfortunately, this is usually not possible. The more normal situation is the one shown in the figure where a modular unit of model A is mapped to several modular units of model B. This is schematically shown in the right-hand side of the figure where the two models are "superimposed" and where a feature of the green aspect is shown to affect several modular units of the yellow aspect. Using the terminology of AOP, this fact is often expressed by saying that the green aspect *cross-cuts* the yellow aspect.

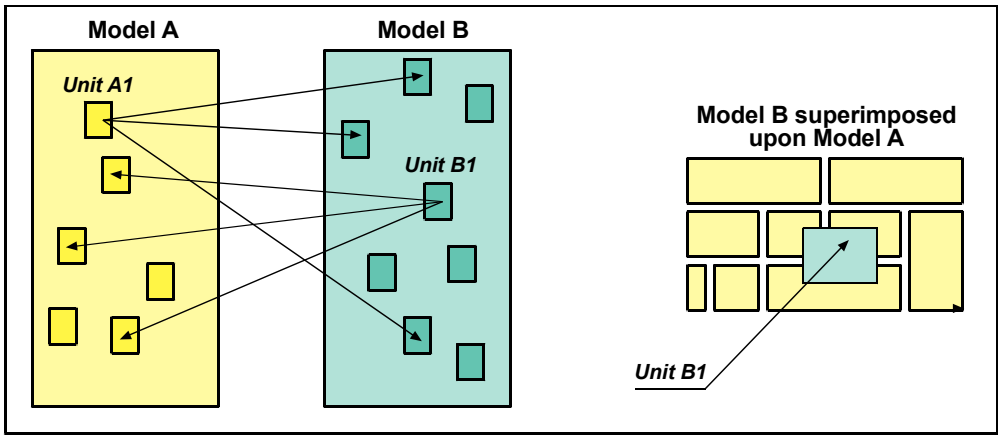

Fig. 1 – Aspects as Perspectives on Models of an Application

Traditional programming techniques – from procedural to object-oriented programming – have privileged the *functional aspect* of applications. The modelling techniques upon which they are based are targeted at modelling functional behaviour and the principle of separation of concerns is applied by organizing an application as a set of cooperating functional units (where, depending on the particular programming paradigm, a functional unit may be a procedure, a module, an object, a class, etc).

Modelling techniques have also been developed for some other typical aspects but, traditionally, it has been impossible to enforce the principle of separation of concerns with respect to more than one aspect at the same time. To illustrate, and with reference to the examples given above, consider the case of an application where both functional and error handling aspects are important and assume that the application is implemented in a class-based language such as Java or C++. In that case, the principle of separation of concerns can be applied to its functional aspect by suitably designing the classes and their interactions. It will then often be possible to localize the code that implements a particular functional requirement in a specific class (or even in a specific method of a class). This has several advantages:

- Checking the correctness of the implementation of the requirement is easy because only one class/method needs to be checked

- Changing the implementation of the requirement is easy because the change only affects one single class/method
- Responding to a requirement change is likely to be easy because the change can often be accommodated with changes in a single class/method

Once implementation in a conventional class-based language is selected, however, it will normally be impossible to localize the code that implements the error-handling aspect of the application. Assume for instance that all application methods return an error code that indicates whether the method completed successfully or whether it encountered some error. Then, simple examples of error handling policies at component level are:

- Never check the return values of methods (i.e. ignore all errors)
- Always check the return value of all methods and, if an error is reported, create an entry in a log file
- Always check the return value of all methods and, if an error is reported, perform a software reset

The code that implements the above policies is spread over the entire code base of the target application (or, using the standard AOP terminology, the error handling aspect *cross-cuts* the functional aspect). This means that changing the way the aspect is implemented (i.e. changing the error handling policy) requires global changes to the application code base. This is far more expensive and error-prone than would be the case if the implementation of the aspect were localized in a dedicated "module".

The AOP paradigm provides efficient ways to express aspects and to implement specifications of aspects into application code in a manner that preserves the principle of separation of concerns. The process of modifying an existing code base to modify the implementation of a certain aspect is called *aspect weaving*. At its most basic, aspect weaving can be seen as a source code transformation process and the aspect oriented language can be seen as a sort of meta-language that specifies the code transformation (see fig. 2). The aspect weaver is a compiler-like program that reads the aspect program and uses it to modify the base code and automatically generate a new base code modified to implement the desired aspect.

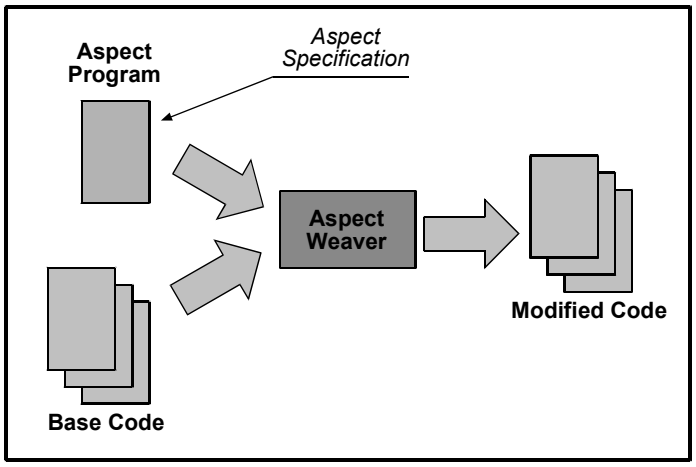

Fig. 2 – Aspect Weaving as Automatic Code Transformation

To illustrate, consider again the previous example of implementing different error handling policies in a certain piece of code. The code on the left-hand side of figure 3 represents the base code which does not implement any error handling. The error handling policy is encoded in an aspect program (not shown in the figure). After the aspect weaving process, a new modified program is automatically generated that is identical to the base program but which includes new code that implements a particular error handling policy. The new code inserted by the aspect weaver is shown shaded in the right-hand box in the figure. Error handling is now encapsulated in a dedicated module (the aspect program). Changes to the error handling policy are localized to this module and can be quickly and safely projected upon the entire base code by the aspect weaver.

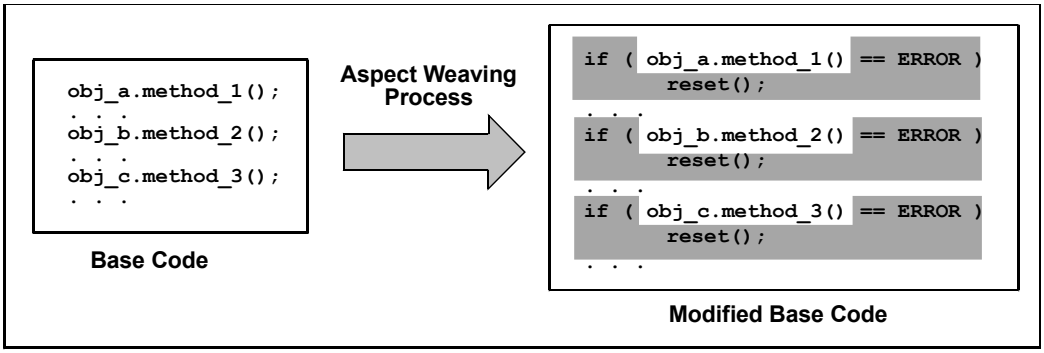

Fig. 3 – Aspect Weaving to inject Error Handling Code

The AOP paradigm is comparatively new and language support is still limited but growing fast. Aspect languages and aspect weavers exist for C, C++ and Java [Asp04, Asj04]. The next section describes a new aspect language and its associated aspect weaver that are especially targeted at qualifiable applications.

2 XWeaver Motivation

The XWeaver project [Xwe04] was motivated by a desire to apply AOP techniques to software that must undergo a qualification process, such as the ECSS-E40 [Ecs04] process for space applications. 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 code. The two basic approaches are:

- The qualification process is applied to the base code and to the aspect weaver. It is assumed that this ensures that the modified code (base code + woven code) is of sufficient quality and therefore not in need of a dedicated qualification process.
- The qualification process is applied to the modified code only and there is no need to qualify the weaver.

The first approach is impractical in the short term because of the difficulty of qualifying an aspect weaver. The second approach, on the other hand, places some indirect constraints on the aspect weaver which must be capable of producing modified code that is amenable to qualification. At the very least it is desirable that the modified code not be harder to qualify than equivalent code that had been written by hand. In practice, this means that the modified code must satisfy the following requirements:

- it must comply with the same coding rules laid down for manually written code
- it must adhere to the same language subset specified for manually written code
- it must be commented to the same level as manually written code

Existing aspect weavers that were evaluated at the *Institut für Automatik* do not satisfy the above requirements. Arguably, their most important shortcoming is that they are unable to handle comments. This is an important drawback because in many cases the code documentation is directly embedded in the source code in the form of JavaDoc or JavaDoc-like comments. The code documentation is automatically generated by processing these comments. If aspect weavers do not update the code comments, then the code documentation becomes invalid and this clearly makes the qualification process of the modified code more expensive. Other shortcomings concern the visual structure of the modified code that is often harder to read than the original base code (the original code layout is normally lost during the weaving process) and the presence of extraneous code that is "pulled in" by the weaver. The XWeaver tool was developed to address these concerns. More specifically, it implements a weaving process that does not change in any way the base code and that is capable of generating comments to document the newly woven code. Broadly speaking, the intention of XWeaver is to produce a modified code that "looks like" manually written code and that is therefore as easy to qualify as code that had been modified by hand. This is also expressed by saying that XWeaver must be *minimally intrusive* in the sense that it must not disrupt the structure of the existing code.

3 XWeaver Approach

The shortcomings of traditional aspect weaving approaches highlighted above stem from the fact that conventional aspect weavers 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 derivation tree. The code modifications defined by the aspect program are performed upon this abstract form of the base code. A code-generating back-end then constructs the modified code. The base code is entirely re-generated. This model allows aspect weavers to carry out sophisticated modifications of the base code but it also destroys some valuable information about the base code, most notably its comments and its layout.

XWeaver takes a different approach in that it operates upon a model of the code that preserves *all* the information in the base code, including formatting, layout and comments. Following recent work by several authors [Bad00, Mam00, Col02, Col03], an XML-based model of the code is used. In particular, among the several offerings currently on the market, the XWeaver project selected srcML [Mal02, Src04]. The main attraction of srcML from the point of view of the XWeaver project is that it preserves all the information in the base code and it offers a "round trip" facility that allow the source code to be re-generated from its XML model in its exact original form.

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

Since XWeaver operates upon an XML-based model of the base code, one could argue that there is no need for a dedicated aspect language since an aspect transformation can be directly expressed as an XSL transformation. This position is correct but impractical. Writing an aspect transformation directly in XSL would be a difficult, error-prone, and rather tedious task requiring a detailed knowledge of XSL and srcML. AspectX provides a higher-level way of describing an aspect transformation. Indeed, one can recognize two primary components in XWeaver. The first one is essentially a compiler that translates the AspectX program into an equivalent XSL program. This program is formulated as a set of rules that define transformations to be performed upon the elements in the srcML model of the base code. The second component of XWeaver provides an interpretation engine that can implement the transformation defined by these rules. The advantage of using AspectX rather than XSL is therefore the same advantage that one has from using a high-level programming language instead of assembler.

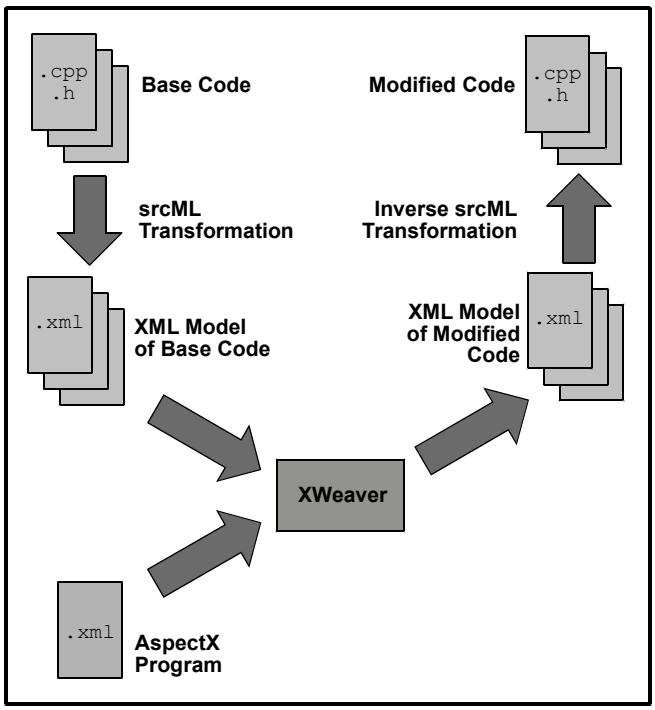

Fig. 4 – XWeaver Approach

AspectX and XWeaver are still at an experimental stage and their scope is correspondingly limited. In particular, they are aimed only at a subset of C++ (roughly limited to the EC++ constructs). However, the language and weaver are already capable of dealing with concrete and recurring aspect weaving problems such as: insertion of pre- and post-conditions, insertion of synchronization code, instrumentation of existing code, or the transformation of passive objects into active objects with their own thread of execution. More importantly, both the language and its weaver are designed to be extensible and adaptable which will allow their scope to grow gradually and will empower its users to customize them to meet their special needs.

4 References

[Bad00] Badros, Greg J. *JavaML: A Markup Language for Java Source Code*. Proceedings of the 9th International World Wide Web Conference (WWW9), Amsterdam, The Netherlands, May 13-15 2000.

[Col03] Collard, M.L., Kagdi, H., Maletic, J.I. *An XML-Based Lightweight C++ Fact Extractor*. Proceedings of the IEEE International Workshop on Program Comprehension (IWPC 2003), Portland, OR, May 10-11, 2003, pp. 124-143.

[Col02] Collard, M.L., Maletic, J.I., Marcus, A. *Supporting Document and Data Views of Source Code*. Proceedings of the 2nd ACM Symposium on Document Engineering (DocEng'02), McLean, VA, November 8-9, pp. 34-41.

[Mal02] Maletic, J.I., Collard, M., Marcus, A. *Source Code Files as Structured Documents*. Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 26-29, 2002, pp. 289-292.

[Mam00] Mamas, Evan, Kontogiannis, Kostas, *Towards Portable Source Code Representations Using XML*, Proceedings of Working Conference on Reverse Engineering (WCRE) Brisbane Australia, November 2000, pp.172-182.

[Zou01] Zou, Ying, Kontogiannis, Kostas, *Towards a portable XML-based source code representation*. Proceedings of XML Technologies and Software Engineering (XSE), 2001.

[Sch02] Schonger, Stefan, Pulvermueller, Elke, Sarstedt, Stefan. *Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees*. Proceedings of the 2nd German GI Workshop on Aspect-Oriented Software Development, February 2002, pp. 59-64.

[Ecs04] ECSS-E-40Part1B, *Space engineering – Software – Part1: Principles and requirements*, ESA Publications, 28 November 2003

[Asp04] AspectC++ Web Site, http://www.aspectc.com/

[Asj04] AspectJ Web Site, http://www.eclipse.org/aspectj/

[Src04] srcML Web Site, AspectC++: http://www.aspectc.com/

[Xwe04] XWeaver Web Site, control.ee.ethz.ch/~ceg/XWeaver