ADAPTING CONTROL SOFTWARE SYSTEMS THROUGH ASPECT-ORIENTED PROGRAMMING

Iwan Birrer [*] Philippe Chevalley [**] Ondřej Rohlík [*]

[*] *Institut für Automatik, ETH Zürich, Switzerland*
[**] *European Space Agency, Noordwijk, The Netherlands*

Abstract: The current practice in the development of control systems shows an increasing demand on software reuse. This paper addresses this issue and describes a prototype tool, called XWeaver, which is based on the aspect-oriented programming technology to achieve the adaptability of reusable software components in an automated way. XWeaver is an aspect weaver for C/C++ that is specifically designed for adapting software with high criticality requirements, as it is for a majority of control applications. *Copyright ©2005 IFAC*

Keywords: Software tools, embedded systems, safety-critical, program costs

1. INTRODUCTION

If software-related costs account for a growing share of total development costs of control systems, the simplest and most effective way to contain these costs is to increase the level of *software reuse* i.e. to reuse the same software component in different operational contexts. In practice, different contexts will always impose different requirements and hence a software component will only be reusable if it can be adapted to these different contexts. In this sense, adaptability is the key to reusability: a software component is reusable only to the extent that it can be adapted to different operational environments.

Embedded control software is often developed in programming languages using a procedural or a modular design paradigm. The adaptability mechanisms offered by this type of approach are very limited. Essentially, adaptability is restricted to the parameterization of routines and functions and to the use of compiler flags to control the selection of software configuration. This low level of adaptability was arguably the main reason for the difficulty in introducing a reuse culture in the world of control systems.

The transition to the object-oriented (OO) technology addresses this deficiency (Pasetti, 2002) with features such as object composition and inheritance. However, these features only cover functional adaptability, namely adaptability with respect to the algorithms implemented by the software (i.e. control algorithm). Adaptability to changes in non-functional aspects (i.e. aspects that covers software features not directly related to its primer purpose) is difficult or impossible to model and implement. Thus, for instance, changes in the error handling policy, in the concurrency and sychronization model, or in the balance between memory and CPU efficiency can hardly be covered by OO features. This is a serious shortcoming because non-functional aspects often play an important role in control applications where product differentiation is often rooted in non-functional differences in the software. Aspect-Oriented Programming (AOP) is a recent concept that allows a designer to model a software according to different functional and non-functional views.

The paper is organised as follows. Section 2 introduces the aspect-oriented programming as a way to handle adaptability to non-functional aspects. Section 3 discusses AOP in the context of control applications and presents in details XWeaver. Section 4 discusses some concrete applications of the tool and relates first experiences in its usage.

2. ASPECT-ORIENTED PROGRAMMING

Aspect-oriented programming is a relatively new software paradigm (Czarnecki and Eisenecker, 2000) that allows uniform treatment of aspects of a software application, which, when a conventional design approach is used, are distributed over the entire code base. The AOP approach allows these aspects to be modularized and then makes it possible to easily change the way these aspects are implemented to adapt them to changing operational circumstances.

2.1 An Aspect Definition

A software application can be seen from different perspectives and an aspect designates one particular perspective and its associated model. As an example, two obvious perspectives can at least be considered for a real-time application: the functional perspective and the real-time perspective. The former perspective privileges the description of the control algorithms and logic that are implemented by the application. A suitable model for it could be a UML (Uniform Modeling Language) class diagram that shows how the modules making up the application are organized and describes the functional behaviour that each of them implements. The real-time perspective instead privileges its timing-related properties (e.g. execution times, output deadlines). Both perspectives define an aspect and a model of the application. In the case of control systems where fault-tolerance is crucial, the error handling policy provides another example of aspect. It should be stressed that the aspects of importance depends on each application. Those outlined above are just examples of potential aspects but, clearly, different applications have different concerns and therefore different sets of applicable aspects. In order to capture the entire behaviour of an application, it will normally be necessary to consider several aspects. Traditional engineering approaches have privileged the functional aspect but, except for trivial cases, this needs to be complemented with other aspects. This point is especially acute in control systems where tight memory and CPU budgets and close interaction with the physical environment impose non-functional constraints that must be handled through non-functional aspects in the software.

2.2 Aspects and Separation of Concerns

The principle of separation of concerns states that a model of an application should be organized as a set of lower level units where each unit encapsulates one particular feature (or view) of the application. The advantage of this approach is that the description of a software feature is localized and is therefore more easily controllable. Localization and controllability of implementation taken together support adaptability because they allow the implemented feature to be easily modified in response to changes in the operational context.

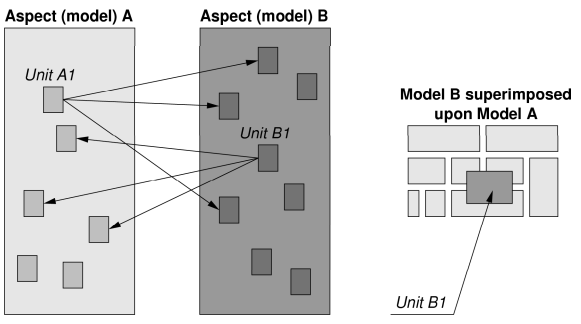

Fig. 1. Aspects as perspectives on models of an application

The problem addressed by AOP arises from the fact that the application of the principle of separation of concerns to different aspects of the same software typically gives rise to organizations of the associated models that are difficult to map to each other. This is illustrated in Figure 1 where two models of the same application are represented. Each model addresses a particular aspect of the application and 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 common situation is the one shown in the figure where a modular unit of Aspect A is mapped to several modular units of Aspect B and vice versa. This is schematically shown in the right-hand side of the figure where the two models are "superimposed" and where a feature of Aspect B is shown to affect several modular units of Aspect A. Using the terminology of AOP, this fact is often expressed by saying that Aspect B *cross-cuts* Aspect A.

Procedural and object-oriented programming techniques 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.

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 an object-oriented language. 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.

Assume for instance that all application functions return an error code that indicates whether the function completed successfully or not. Then, simple examples of error handling policies at component level are:

- **Recovery action**: Always check the return values of functions and, if an error is reported, perform a reset of the application;
- **Logging action**: Always check the return values of functions and, if an error is reported, create an entry in a log file;
- **No action**: Never check the return values of methods (i.e. ignore all errors).

The code that implements the above policies is spread over the entire structure of the target application (i.e. the error handling aspect cross-cuts the functional aspect). This means that adapting the implementation of the aspect to a new operational constraint (i.e. changing the error handling policy) requires global changes to the source code. This is far more expensive and error-prone than would be the case if the implementation of the aspect was localized in a dedicated "module" (i.e. if the principle of separation of concerns was applied to both the functional and error handling aspects).

2.3 Aspect Languages and Aspect Weavers

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 source code to reflect the implementation of a certain aspect is called *aspect weaving*. Several AOP languages exist but they are often based on different aspect weaving strategies. 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. AOP then becomes a form of automatic code generation where both the input and the output code are written in the same language. Figure 2 illustrates the weaving process.

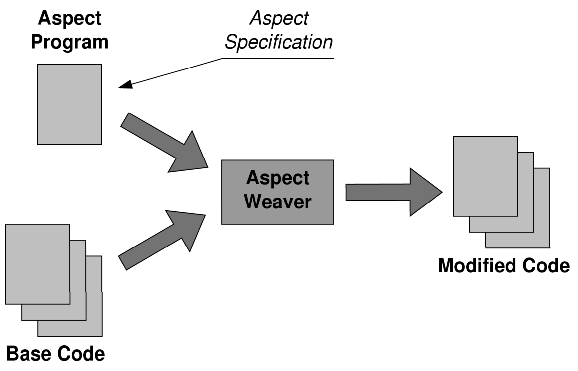

Fig. 2. Aspect weaving as automatic code transformation

The boxes at the bottom left corner represent the starting base code organized as a set of modules. The box at the top left corner represents an aspect program that defines a particular code transformation. The aspect weaver is a compiler-like program that uses the aspect program to modify the base code and automatically generate a new source code implementing the desired aspect. Note that the base code and the modified code are both written in the same language.

To illustrate, consider again the previous example dealing with the implementation of different error handling policies in a certain piece of code. All existing aspect languages would allow the error handling policies considered in the examples to be encapsulated in a single aspect program. The base code could be developed independently of any particular error handling policy and the aspect weaving process could be used to project a particular error handling policy upon it. Adaptation could be achieved by modifying only the meta-code localized in the aspect program.

3. THE XWEAVER ASPECT WEAVER

Although several aspect weavers already exist for the most commonly used languages (C++ (Spinczyk *et al.*, 2002) and Java (Gradecki and Lesiecki, 2003)), as will be argued below, these weavers are targeted at desktop applications and would be unsuitable for control systems. In order to offer an AOP tool to the adaptation of embedded control applications, the authors have developed XWeaver as an aspect weaver specifically targeted at critical embedded applications.

3.1 Motivation

The inadequacy of conventional aspect weavers for embedded applications is due to the fact that these weavers operate upon the abstract syntax tree representation of the base code. The weaving process, in other words, is performed upon the output of the parser rather than on the source code. This means that the aspect modifications are introduced at the level of the object code rather than of the source code. A modified source code can usually be generated but it is normally unreadable because it has lost both the layout and the comments that were present in the original code. In the embedded world, it is normally unacceptable not to have visibility over the source code. Among other things, a poor visibility makes debugging and code inspection far complicated.

Especially serious problems arise in the case of critical applications, which must be subject to some kind of qualification programme to certify that they have 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 tool or the woven code.

If the *qualification process* is performed upon the base code, the aspect weaver and the code to be woven, then this ensures that the modified code is of sufficient quality and therefore does not need any dedicated qualification process. The other possibility is to perform a qualification process only on the modified code and in this case there is no need to qualify the weaver. The first approach is regarded as impractical in the short term because of the difficulty of qualifying an aspect weaver. This difficulty is due both to the intrinsic complexity of aspect weavers and to the lack of experience in qualifying this type of applications. 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 written by hand. In practice, this means that the modified code must:

(1) Comply with the same coding rules laid down for manually written code,
(2) Adhere to the same language subset specified for manually written code,
(3) Be commented to the same level as manually written code.

Conventional aspect weavers 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-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 and structure 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 is intended to implement 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.

Additionally, XWeaver was developed to satisfy two further requirements that are of importance in the software reuse domain, namely customisability and extensibility. *Customisability* refers to the possibility of tailoring the rules that are used to weave new code into existing code. *Extensibility* refers to the possibility of adding new rules to handle new types of aspects. Both are important for control applications, which are often characterized by idiosyncratic requirements. In order to accomodate them, the aspect language and the weaving process must be correspondingly flexible and adaptable. Extensibility and adaptability are also important for another reason. Developing a comprehensive aspect language and aspect weaver for a base language of the complexity of C++ is a daunting task. It is believed that a more practical approach is to begin by developing an aspect language and weaver that only cover a core of functionalities of the base language but to ensure that these are extensible so as to allow the language and the weaver to grow gradually. Sometimes, on the other hand, it may be necessary to avoid using some language constructs, e.g., for safety reasons.

3.2 XWeaver *Approach*

The shortcomings of traditional aspect weaving approaches highlighted in the previous section stem from the fact that conventional aspect weavers operate upon an abstract representation of the base code. They parse the base code, construct its abstract syntax tree, and apply the modifications defined by the aspect program 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 secondary but valuable information about the base code, most notably its comments.

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 (Badros, 2000; Mamas and Kontogiannis, 2000), an XML-based model of the code is used. In particular, among the several offerings currently available, the XWeaver project selected srcML (Collard *et al.*, 2002). The main attraction of srcML is that it preserves all the information in the base code and it offers a "round trip" facility that allows the source code to be re-generated from its XML model in its exact original form. A drawback of srcML is that its model of the base code is more coarse-grained than would be the case if full parsing were carried out. Dedicated XML elements are only used for high-level structures (e.g. classes, methods, if-then-else clauses) and this poses a fundamental limit to the kind of transformations that can be performed by XWeaver. However, srcML may be upgraded in the future to produce a finer-grained representation of the base code. In order to be ready to take advantage of these upgrades, XWeaver was designed to be extensible.

The use of an XML representation of the code suggests the use of XSL (Extensible Stylesheet Language) as an implementation language of the weaver, but this imposes the aspect program to be written in XML. For this reason, an XML-based language, called *AspectX*, was defined to express the aspects to be woven. The choice of an XML-based representation of the base code 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. XWeaver is targeted at C/C++ applications but it only operates upon the srcML representation of C/C++. It is conceivable that srcML may be extended to represent other object-oriented languages (notably Java). The upgrade of XWeaver and AspectX to handle this case would probably be significantly simpler than if the weaver and its language were directly operating upon the base code. Complete decoupling from the base language may not be a realistic option but the presence of an XML layer between the base code and the weaving process helps insulate the latter from the former.

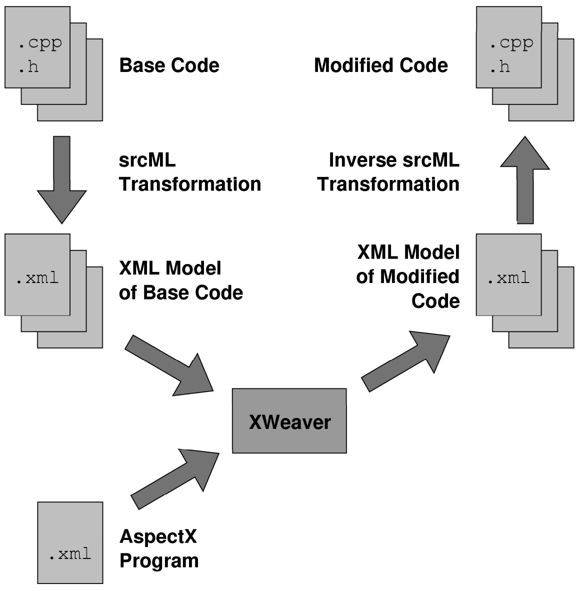

Fig. 3. Mode of operation of XWeaver

As illustrated in Figure 3, the weaving process has two inputs: the base code and the AspectX program that specifies the target transformation. The weaving of the transformation is performed by the XWeaver program that acts upon an XML-based model of the base code that is constructed by the srcML application. The XWeaver produces an XML-based model of the modified code. The modified code is finally derived by applying to this model the inverse srcML transformation.

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 and rather tedious task requiring a detailed knowledge of XSL. AspectX is intended to provide a higher-level way of describing an aspect transformation to be easily accessible by non-specialist users. 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 compiler is implemented as a set of rules that defines transformations to be performed in the srcML model of the base code. The second component of XWeaver provides an engine that can implement the transformation defined by these rules. The advantage of using AspectX rather than directly XSL is the same advantage that one has from using a high-level language instead of assembler. Moreover, syntax and semantics of AspectX is based on state-of-the-art aspect language AspectJ (Gradecki and Lesiecki, 2003), which further ease mastering AspectX.

4. USAGE EXPERIENCE

XWeaver is a research prototype tool available (*XWeaver Project Web Site*, 2004) under the terms of the GPL license. At current time, it has a somewhat limited scope in that it is restricted to the object-oriented part of C++ and, among the C++ constructs, it only covers those used by the Embedded C++ (or simply EC++) subset of the language. EC++ is a language of choice for many critical control applications, especially in the space and avionic industry. The weaver is already capable of dealing with concrete problems such as the insertion of:

- Pre- and post-conditions code in selected methods of the base code,
- Synchronization code to ensure access in mutual exclusion to selected methods,
- Code to transform a passive object into an active object with its thread of execution,

The above and other types of aspect transformations were demonstrated on a library of sample aspect programs that is delivered together with the XWeaver tool to provide guidelines to users on how to implement aspect programs in AspectX. The aspect programs in this library operate upon two distinct code bases. The first one is the OBS Framework (*OBS Framework Web Site*, 2004), which is a repository of reusable components for embedded control systems. This code base is intended to be representative of typical embedded control application code. The second code base is a simple but complete application (the "car application"), which can be compiled, linked and run both before and after an aspect weaving so as to check the effect of the transformation.

XWeaver has proved to be a very satisfactory tool. The range of transformations it can handle at present is limited when compared to other aspect weavers (AspectJ, AspectC++) but, within these limits, it gives full control to the programmer over the transformation process.

5. CONCLUSIONS AND FUTURE WORK

This paper starts from one premise and makes two claims. The premise is that reduction of software-related costs in embedded control systems can only be achieved by increasing software reusability and that this only can be done by making software artefacts more adaptable. Based on this premise, the first claim made by this paper is that aspect oriented techniques are essential to increase the adaptability of embedded control software because they are the only means to control the implementation of various non-functional aspects. The second claim is that existing aspect weavers are not well suited to embedded applications because they operate on an abstract representation of the base code and therefore do not offer sufficient control over the transformation process. If one accepts these two claims, then the XWeaver tool will be seen as an important means to control software development costs.

Even if first evaluation results allowed us to draw promising conclusions, the prototype tool has still to be improved. Two areas are considered. The first one concerns the possible difficulty to handle the AspectX language. Although sample programs delivered with the tool greatly simplify the task of writing AspectX programs, it remains true that writing such programs can be error-prone for beginners. In order to address this shortcoming, XWeaver will be upgraded to perform more error checking and to have improved error-reporting facilities to help users rapidly debug their aspect programs. Additionally, a graphical user interface is being developed to replace the current command line interface. The second area of improvement concerns the range of aspect transformations implemented. In particular, XWeaver is currently biased towards C++ rather than C. Given the prevalence of C in control systems, this imbalance needs to be corrected with more emphasis on C-specific transformations.

REFERENCES

Badros, J. (2000). JavaML: A Markup Language for Java Source Code. In: *Proc. 9th Int. World Wide Web Conference.*

Collard, M., J. Maletic and A. Marcus (2002). Supporting Document and Data Views of Source Code. In: *Proc. 2nd Symposium on Document Engineering.*

Czarnecki, K. and U. Eisenecker (2000). *Generative Programming – Methods, Tools, and Applications.* Addison-Wesley.

Gradecki, J.D. and N. Lesiecki (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java.* Wiley Publishing.

Mamas, E. and K. Kontogiannis (2000). Towards Portable Source Code Representations Using XML. In: *Proc. Working Conference on Reverse Engineering.*

OBS Framework Web Site (2004). `http://pnp-software.com/ObsFramework/`.

Pasetti, A. (2002). *Software Frameworks and Embedded Control Systems.* LNCS. Springer.

Spinczyk, O., A. Gal and W. Schröder-Preikschat (2002). AspectC++: An Aspect-Oriented Extension to C++. In: *Proc. 40th Int. Conference on Technology of Object-Oriented Languages and Systems.*

XWeaver Project Web Site (2004). `http://www.pnp-software.com/XWeaver/`.