Implementing Adaptability in Embedded Software through Aspect Oriented Programming

I. Birrer
ETH – Zurich
Physikstrasse 3
Zurich CH-8092
SWITZERLAND
ibirrer@control.ee.ethz.ch

V. Cechticky
ETH – Zurich
Physikstrasse 3
Zurich CH-8092
SWITZERLAND
cechti@control.ee.ethz.ch

A. Pasetti
P&P Software GmbH
Physikstrasse 3
Zurich CH-8092
SWITZERLAND
pasetti@pnp-software.com

O. Rohlik*
ETH – Zurich
Physikstrasse 3
Zurich CH-8092
SWITZERLAND
rohlik@control.ee.ethz.ch

Abstract **– Reusability is the key to reduction in software costs for embedded systems. Software is only reusable if it can be adapted to different operational environments. Conventional software technologies promote functional adaptability. In the embedded world, however, non-functional aspects are especially important. This paper introduces Aspect Oriented Programming (AOP) as a technique to achieve adaptability to non-functional requirements and it presents the XWeaver tool. This is an aspect weaver that is specifically aimed at embedded applications, especially those with high criticality requirements. The paper describes the structure of XWeaver and discusses experience from its usage.**

I. INTRODUCTION

Software-related costs account for a growing share of total development costs of embedded systems in general and mechatronic systems in particular. 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 software for mechatronic applications is usually developed in low-level languages like C using a procedural or a modular design paradigm. The adaptability mechanisms offered by this type of approach are extremely 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 is arguably the main reason for the difficulty in introducing a reuse culture in the embedded world.

The transition to object-oriented language like C++ partially addresses this deficiency [1]. Object orientation offers more sophisticated adaptation mechanisms such as inheritance or object composition. These mechanisms, however, only cover *functional* adaptability, namely adaptability with respect to the algorithms implemented by the software. Adaptability to changes in the non-functional aspects of an application are 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 object-oriented mechanisms. This is a serious shortcoming because non-functional aspects play an especially important role in embedded applications where product differentiation is often rooted in non-functional differences in the software.

Against this background, this paper introduces aspect oriented programming as a way to handle adaptability to non-functional aspects and it introduces XWeaver – an aspect weaver that the authors have developed and that is specifically targeted at embedded applications.

Section II describes the aspect oriented programming paradigm in general. Section III discusses aspect oriented programming in the context of embedded applications and presents the XWeaver tool [2, 3]. Section IV discusses some concrete applications of the tool. The XWeaver tool is available as free software under a GPL licence.

II. ASPECT ORIENTED PROGRAMMING

Aspect oriented programming, or AOP for short, is a new software paradigm [4] which 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 modularization makes it possible to easily change the way these aspects are implemented to adapt them to changing operational circumstances. The non-functional aspects that are so important for embedded applications are often those that are non-localized and are therefore precisely those that an AOP approach can help handle.

A. What is an Aspect?

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

* corresponding author

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. This need is especially acute in embedded applications 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.

B. Aspects and Separation of Concerns

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. 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 of the application. The advantage of this approach is that the description of a feature is localized and is therefore more easily controllable. Localization and controllability of implementation taken together support adaptability because they allow the implementation of the feature to be easily modified to respond to changes in the operational context.

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 aspect (model) A is mapped to several modular units of aspect (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 aspect B is shown to affect several modular units of the aspect A. Using the terminology of AOP, this fact is often expressed by saying that the aspect B *cross-cuts* the aspect A.

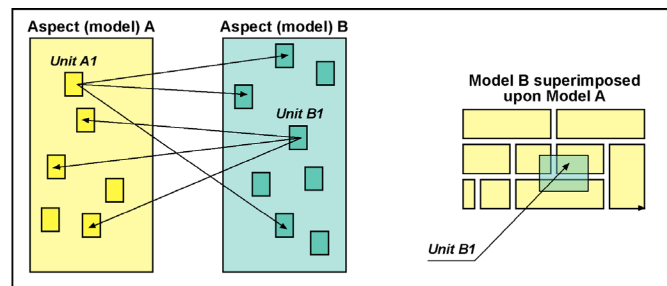

Fig. 1. Aspects as perspectives on models of an application.

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).

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 of the application.

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 adapting the implementation of the aspect to a new operational environment (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" (i.e. if the principle of separation of concerns were applied to the error-handling aspect as well as to the functional aspect).

C. 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 code base to modify the implementation of a certain aspect is called *aspect weaving*. Several different AOP languages exist that implement aspect weaving in different ways. 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 starting and the end code are written in the same language. This view of AOP is illustrated figure 2.

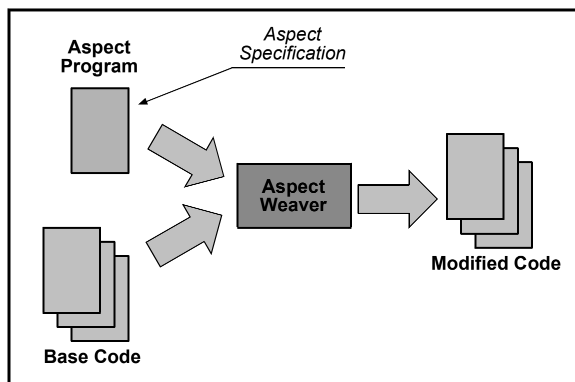

Fig. 2. Aspects weaving as automatic code transformation

The boxes at the bottom left corner represent the starting base code. This is 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 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. Note that both the base program and the modified base program are written in the same language.

To illustrate, consider again the previous example of implementing 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 of the error handling policy could be done by modifying only the localized code in the aspect program.

III. THE XWEAVER ASPECT WEAVER

Although the AOP paradigm is comparatively new, aspect weavers already exist for the most commonly used languages (C++ [5, 6], and Java [7]). As will be argued below, however, these weavers are targeted at desktop applications and would be unsuitable for most embedded systems. In order to allow application of an AOP approach to embedded application, the authors have developed XWeaver as an aspect weaver specifically targeted at embedded – and in particular critical – application.

A. 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 rather than of the source code. A modified source code can usually be generated but this is normally unreadable because it has lost both the layout and structure 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, diminished visibility will make debugging far harder because it makes it harder to map executable instruction to source code.

Especially serious problems arise in the case of critical applications – a category to which belong a vast number of embedded applications (in the avionics, medical, transportation and other fields). The code of critical applications must be subjected to some kind of qualification programme 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 code. The two basic approaches are:

- The qualification process is performed upon the base code, the aspect weaver and the code to be woven. 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 performed on the modified code only and 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 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

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 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 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. 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 and must insert new code that complies with this structure.

Additionally, XWeaver was developed to satisfy two further requirements that are of importance in the embedded domain, namely *customizability* and *extensibility*. Customizability 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 embedded applications which are often characterized by idiosyncratic requirements. In order to accomodate them, the aspect language and the weaving process must be correspondingly flexbile 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.

B. 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 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 [8, 9, 10, 11, 12], an XML-based model of the code is used. In particular, among the several offerings currently on the market, the XWeaver project selected srcML [9, 10]. The main attraction of srcML from the XWeaver point of view 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. 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 (classes, methods, if-then-else clauses, etc) 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 specifically designed to be extensible.

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 is probably not 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.

The use of an XML-based model of the base code suggests the use of XSL[1] 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 to be woven by XWeaver. The corresponding language is called *AspectX*.

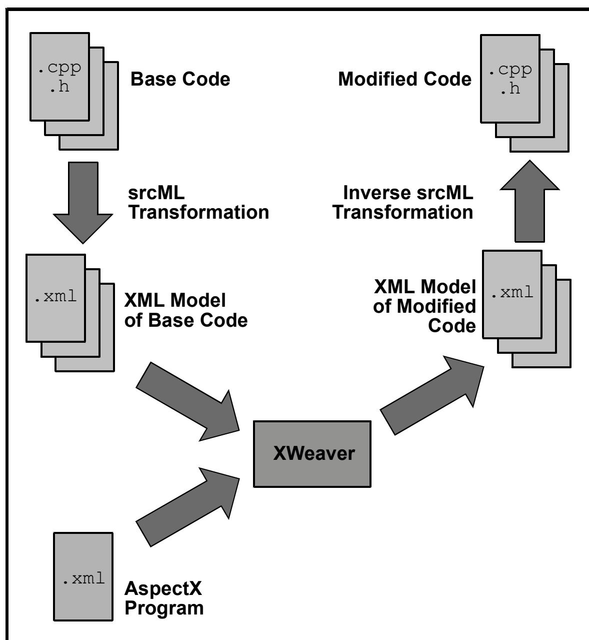

Fig. 3. Mode of operation of XWeaver.

Figure 3 illustrates the weaving process used by XWeaver. 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, error-prone, and rather tedious task requiring a detailed knowledge of XSL that only few users are likely to have. AspectX is intended to provide a higher-level way of describing an aspect transformation and one that is also accessible to 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 XSLT program. This AspectX compiler is implemented as a by using 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 directly XSL is therefore the same advantage that one has from using a high-level programming language instead of assembler.

[1] Extensible Stylesheet Language

IV. USAGE EXPERIENCE

The development of XWeaver is currently on-going. The most up-to-date information about the project status can be found on the project web site [3]. This web site also gives access to the weaver software which is freely available and is distributed under the terms of the GPL licence. At the time of writing, XWeaver is available as a research prototype and 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 EC++ subset of the language (this excludes constructs like templates, multiple inheritance, run-time type identification, exception handling, etc). Despite these limitations, the weaver is already capable of dealing with concrete problems such as:

- Insertion of pre- and post-conditions code in selected methods of the base code;
- Insertion of execution tracing code at selected locations in the base code;
- Insertion of synchronization code to ensure access in mutual exclusion to selected methods in the base code;
- Insertion of code to transform a passive object into an active object with its own thread of execution;
- Insertion of comments that creates or modifies existing class or method comments.

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 a random selection of classes from the OBS Framework [13]. The OBS Framework is a repository of reusable components for embedded control systems. This code base is intended to be representative of typical embedded application code. The second code base is a simple but complete application (the “car application”). The application can be compiled, linked and run both before and after an aspect transformation 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 but, within these limits, it gives full control to the programmer over the transformation process. The price paid for this high level of control is the comparatively low level of the AspectX language that requires the programmer to specify in detail the desired transformations.

XWeaver is currently at the prototype level (but it is already available for use and can be downloaded from [3]) and work is in progress to improve it. This work concentrates on two areas. The first one concerns the ease of use of the tool. AspectX is an XML-based language and its use requires some familiarity with XML [14] and XPath [15]. Additionally, since the weaver operates upon the XML model of the code generated by srcML, users must become acquainted with the structure of this model. The sample programs delivered with the tool greatly simplify the task of writing AspectX programs but it remains true that writing such programs can be error-prone for beginners. In order to address this shortcoming, the 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, and again in order to make the tool more user-friendly, a GUI-based user interface will be developed to replace the current DOS-based command line interface.

The second area of improvement concerns the range of aspect transformations implemented by XWeaver. In particular, XWeaver is currently biased towards C++ rather than C. Given the prevalence of C in embedded systems, this imbalance needs to be corrected with more emphasis on C-specific transformations. It should however be noted that the expansion of the transformation capabilities of XWeaver is a "physiological change" since, as explained in the previous section, XWeaver is designed as a repository of independent and self-contained transformation rules. Extensions are implemented simply by adding new rules to this repository.

V. CONCLUSIONS

This paper starts from one premise and makes two claims. The premise is that reduction of software-related costs in embedded systems can only be achieved by increasing software reusability and that this can in turn only 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 software because they are the only means to control the implementation of non-functional aspects which often play a crucial role in embedded applications. 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 introduced in this paper as an aspect weaver specifically aimed at embedded applications will be seen as an important means to control software development costs of embedded applications.

VI. REFERENCES

[1] A. Pasetti, *Software Frameworks and Embedded Control Systems*, LNCS Vol. 2231, Springer-Verlag, 2002.

[2] I. Birrer, P. Chevalley, A. Pasetti, O. Rohlik, "An Aspect Weaver for Qualifiable Applications", *Proceedings of the 14-th Digital Avionics Systems in Aerospace (DASIA)*, Nice, France, 2004.

[3] XWeaver project web site, http://www.pnp-software.com/Xweaver

[4] K. Czarnecki, U. Eisenecker: *Generative Programming – Methods, Tools, and Applications*, Addison-Wesley, 2000.

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

[6] S. Chiba: "OpenC++ Programmer's Guide for Version 2", *Technical Report SPL-96-024*, Xerox PARC, 1996.

[7] J.D. Gradecki, N. Lesiecki: "Mastering AspectJ: Aspect-Oriented Programming in Java", *Wiley Publishing, Inc.*, Indianapolis, USA, 2003.

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

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

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

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

[12] E. Mamas, K. Kontogiannis, "Towards Portable Source Code Representations Using XML", *Proceedings of Working Conference on Reverse Engineering (WCRE)* Brisbane Australia, pp.172-182.

[13] OBS Framework Project Web Site, http://www.pnp-software.com/ObsFramework

[14] XML Tutorial, www.w3schools.com/xml/default.asp

[15] XPath Tutorial, www.w3schools.com/xpath/default.asp