Journal of Computations & Modelling, vol.1, no.1, 2011, 1-16
ISSN: 1792-7625 (print), 1792-8850 (online)
International Scientific Press, 2011

RTE-AMO: A Round-Trip Engineering – Based Approach for Software Maintenance Optimization on a MDA Engine: application to OptimaDev

Thomas Djotio Ndie[1], Claude Tangha[2] and Raphaël Ledoux Fongang[3]

Abstract

In this paper, we propose a Round-Trip Engineering-based approach of optimizing the corrective, adaptive, evolutionary or perfective maintenance of applications on the OptimaDev, a MDA engine. It is a method which allows the automatic synchronization of updates between the model of an application and its source code when changes are brought vice-versa to the one or the other. The modifications made either on the model or in the source code and intervening after the source code generation from the MDA engines are not reflected mutually. In the context of the model-driven architectures (MDA), the ideal is to constantly have a model of the application which reflects its source code. Our approach helps to significantly reduce the quota of the total budget of the software development affected to maintenance.

---

[1] Department of Computer Science, University of Yaounde I, Yaounde, Cameroon, e-mail: tdjotio@gmail.com
[2] Department of Computer Science, National Advanced School of Engineering, University of Yaounde I, Yaounde, Cameroon, e-mail: ctangha@gmail.com
[3] Kossery Technion, Douala, Cameroon, e-mail: raphfongang@gmail.com

Article Info: *Revised* : July 20, 2011. *Published online* : August 31, 2011.

Keywords: software engineering, optimization, maintenance, MDA, round-trip engineering, OptimaDev.

1 Introduction

Any modification which happens further to the creation of a software comes back as part of the normal process of its maintenance. The software development driven by architectures well known as MDA for Model Driven Architecture [1], [2], [6], [7], [8] imposed itself in the software development process, especially in the computer engineering services-based companies which mostly already use an MDA engine like OptimaDev [1], [13] enabling them to generate source code from the model [3], [4], [18]. Maintenance is indeed a fundamental stage of the life cycle of a software. Anomalies can be detected, new functionalities added, tests remade in order to be sure that corrections do not bring other problems on the system. It then raises the question to know how to make to keep their coherence in the case of great or big applications? In other words how to automatically synchronize the updates of the source code after modification of the model and vice versa?

According to [21], an investigation carried out in the USA near 55 companies reveals that 53% of the total budget of a software is appointed to its maintenance. This cost is distributed as follows: 34% for the progressive or evolutionary maintenance (modification of the initial specifications), 10% for adaptive maintenance (new environment, new users), 17% for the corrective maintenance (correction of bugs), 16% for perfective maintenance (to improve the performances without changing the specifications), 6% for assistance to the users, 6% for the quality control, 7% for the organization / monitoring, 4% for miscellaneouses.

At the end of the process of creation of software by code generation from a MDA engine like OptimaDev [1], [13], there is very often a distortion between the model and the application. This not-coherence between the code and the model involves a risk of error at the time of the future maintenance operations that causes an increase in the cost of maintenance of the application. To keep or to preserve coherence between the model of an application and its source code when one or the other is modified and particularly in the context of MDA, we propose in this paper an approach which consists in applying the Round-Trip Engineering (RTE) [13], [14], [17], [20], [23] to the results produced by an engine MDA, the case of OptimaDev is used. Our article is structured in 3 sections. The first section is devoted to software maintenance where we introduce the synchronization methods between the code and the model and the concepts relating to the RTE. In the second section, we detail our methodological approach, followed by its implementation which we present in the section 3 before finishing by the conclusion.

2 Software maintenances techniques

The update of the model of an application is not easy when the software project includes several classes and the update of the corresponding source code becomes complex. To automate the update of the model, some software builders or editors use Computer Aided Software Engineering (CASE) which, for some, already integrate the stage of maintenance by allowing the update of the model and/or of the source code but manually and of a separate way.

2.1 Synchronization methods between code and model

MDA Engines [13], [18] allow to make the generation of software components by using associated concepts. Following this generation, there is a source code which corresponds exactly to the model. Two methods lend themselves well to preserve or to keep the coherence of models and of source code: the MDE (Model Driven

*Identify applicable sponsor(s) here. (dispensable)

Engineering) and the RTE (Round-Trip Engineering).

2.1.1. Model Driven Engineering (MDE)

MDE is an approach which takes all its importance as part of software and material architectures run by models using standards such as MDA. Such architectures integrate with a software development process based on models by ensuring at every level of modeling, that the reused and got models have requested or needed qualities. This step run by models thus puts the model at the center of the concerns of the analysts/designers. The choice of the formalism of description of models holds a key importance. Currently two tendencies free: the first one more general as UML [5], [6], the second more fulfilling requirements of a field or domain with Domain-Specific Languages (DSL) and the Domain Specific Model Languages (DMSL). In the first case, the development process is longer, because one starts from a more abstract model. The second is more targeted, because designers can lean on technologies peculiar to a domain. MDA leans less for MDE, because its implementation binds the model to the target platform. Therefore, we chose Round-trip since it is independent of the platform of execution [8], [9], [10].

2.1.2. Round-Trip Engineering (RTE)

RTE is a software development method which allows the generation of the model from the source code and the generation of the source code from the model [13], [14], [17], [19], [20]. In other words, the round trip allows a bi-transformation between the model and the source code.

Let m be the model of a product p. Let r be the reverse engineering function such as

$$r(p) = m.$$

Let f be the function of the product generation such as

$$f(m) = p.$$

If $f(r(p)) = p$ and $r(f(m)) = m$, then $< f, r >$ is the RTE of the system. Round-Trip is seen as a combination of Forward and of reverse engineering [17], [23]. The *forward engineering* is aimed at producing code from the model. In our case, one speaks about Round-Trip only when there is already a source code from a model provided by a MDA engine; we use OptimaDev is this context [1]. The reverse engineering**,** often considered to be a method allowing to have the source code from a binary file makes indeed the opposite of the forward engineering, by allowing to produce the model from the source code. In our case it is considered to be a method allowing having a UML model from the source code.

2.2 OptimaDev: our engine MDA

OptimaDev is the MDA engine that we developed [13]; it is fully described in [1]. It allows reducing the development costs and the duration of the realization of an application: reduction of the number of developers intervening on the application and reduction of the duration of creation of the software. Starting from a XMI[4] (XML Metadata Interchanges) model, of a set of XSLT (extensible Stylesheet Language Transformations) files and Spring[5] files, it allows to generate 50% of the applications. The result got from the generation will be changed or modified and adapted to the problem arising.

3 Methodological Approach and design

3.1 Methodological approach

UML 2.0 proposes many notations allowing to observe different components of a

[4] It is a standard of UML metadata exchange in XML which defines an open and independent way to any CASE editor to describe UML models.
[5] Spring is an open source framework for 3-third applications which facilitates the development and tests the http://www.springsource.org/.

program, like its static structure, the interactions and correlations between different objects or else a use case. We use in our case the class diagrams. For the application of RTE, we exploit XMI files equivalent to the class diagrams of [19]. From the XMI file and the source code (class files), we arise a tree structure where each node has: a set or group of information of definition like the name of a class or the signature of a method; a set of atomic properties like the visibility; several sets of thread elements where each unit is called “group” like class attributes or a class method. A node representing a method has as threads its parameters and a graph of flow control. The graph of flow control represents all the instructions of the method. The synchronizer works only on these two trees, by taking into account only the nodes contents.

The trees built from the XMI file and from the source code are constituted of nodes and graphs of flow control. Therefore, to create a tree, it is enough to create its nodes and its graphs of flow control. Nodes have static data. In other words, to create a node from the XMI files (source code) returns to copy corresponding data in the nodes. To create a node of a class C1 in the tree, one will create a node in which one will copy the data of the class C1 from the source (XMI file or source code). This node will be connected to the root. We adopted six stages for the Round-Trip Engineering defined as follows:

- Stage 1: Creation of the derivation tree of the code in RAM;
- Stage 2: Creation of the the simplified tree of the code in RAM;
- Stage 3: Creation of the simplified tree of the model;
- Stage 4: Synchronization of data of the model and of code;
- Stage 5: update of the source code or forward engineering;
- Stage 6: update of the model or reverse engineering.

3.2 Design of the Round-Trip

For the modeling of the Round-Trip, we are going to use UML 2.0 [6], represent the use case diagram of a source code or a model and also show the different classes necessary in its realization. To accomplish the Round-Trip of different performed modifications, that it is on the code or the model, it will be necessary to take into account all the possibilities related to the update of the source code or the model by the developer. We will limit ourselves to object oriented programs and for this reason, we are going to use classes as background or like basic elements. The Figure 1 illustrates the use case diagram resulting from the analysis. It summarizes the modifications performed on a class.

In this use case diagram depicted figure 1, use cases of the type "management" reference create, update and delete operations of the concerned item. These operations are performed by the developer. This diagram includes following use cases:

- Management of classes which includes: Management of the declarations of classes, Management of the inheritance, Management of types of Parametres and Management of associations;
- Management of the declarations of attributes;
- Manage the methods: this use case takes into account those of "Manage of the methods declarations", of "Manage the methods contents ", of "Manage the method parameters or settings ".

From the analysis which produced us the use case of the figure 1, we obtained the class diagram illustrated figure 2. One divided the different classes of the system in three groups which are: classes representing the data structure of the system, classes building data in RAM and classes to perform updates in the different parts of the system (source code and model). We describe the classes hereafter:

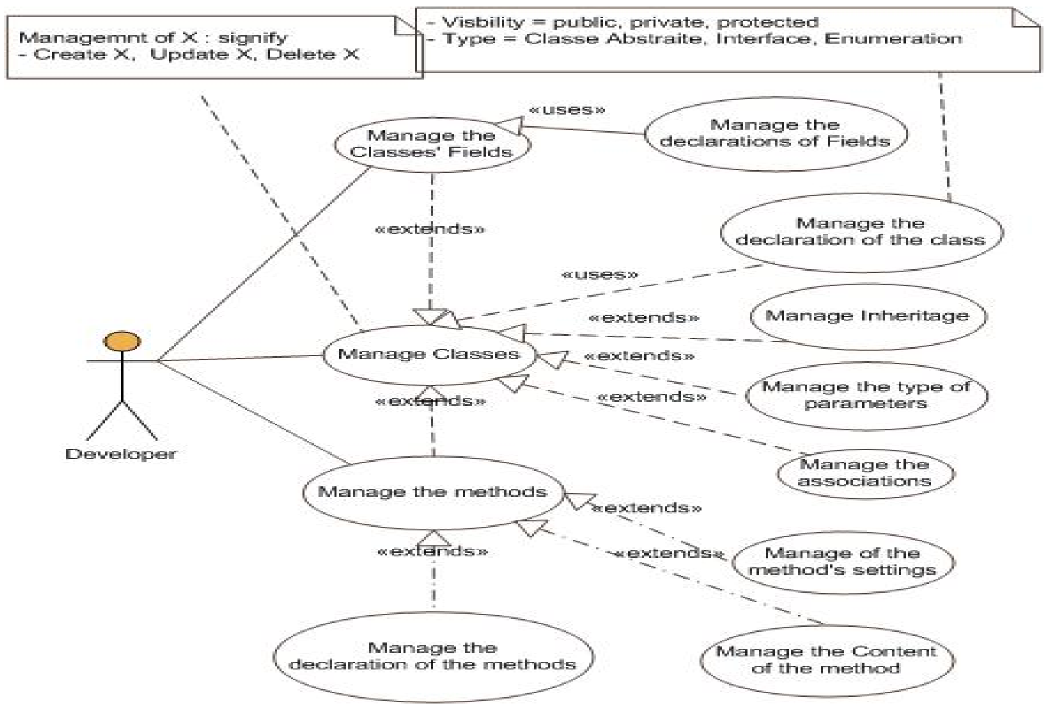

Managemnt of X : signify
- Create X, Update X, Delete X
- Visbility = public, private, protected
- Type = Classe Abstraite, Interface, Enumeration
«uses»
Manage the Classes' Fields
Manage the declarations of Fields
«extends»
Manage the declaration of the class
«uses»
«extends»
Manage Inheritage
Manage Classes
«extends»
Manage the type of parameters
«extends»
«extends»
Manage the associations
Developer
Manage the methods
«extends»
«extends»
Manage of the method's settings
«extends»
Manage the declaration of the methods
Manage the Content of the method

Figure 1. Use cases diagram of modifications performed out on a class

3.2.1. Classes representing the data structure of the system:

- *GeneralVertex*: it represents the abstract class from which inherits all other elements representing the structure of the system;
- *RacineXMIVertex*: it represents the root of the nodes of the tree build from the XMI file. There should only be one instance of this class in RAM. Therefore, it uses the singleton pattern.
- *RacineCodeVertex*: it represents the root of the nodes of the tree constructed from the source code. This class uses the singleton pattern.
- *ClassVertex*: it corresponds to the node giving the representation of a class.
- *AttributeVertex*: it corresponds to the node giving the representation of an attribute of a class.
- *MethodVertex*: it corresponds to the node giving the representation of a

method of a class.

- *PackageVertex*: it corresponds to the node giving the representation of the package in the XMI file.
- *ParameterMethod*: it corresponds to the node giving the representation of the parameters of a method.
- *AssociationVertex*: it corresponds to the node giving the representation of an association between classes.
- *GFCRacineVertex*: it corresponds to the root of the graph of the flow control of a method.
- *GeneralGFCVertex*: it represents the abstract class from which inherit all nodes of the graph of the flow control.
- *GFCVertexImpl*: it corresponds to a node of the graph of control flow.

3.2.2. Classes building data in RAM

- *ConstructorRacineCodeVertex*: built a tree from source code. The root of such tree is an instance of the class: *RacineCodeVertex*.
- *ConstructorRacineXMIVertex*: built a tree from a XMI file. The root of such tree is an instance of the class: *RacineXMIVertex*.

3.2.3. Classes to perform updates in the different parts of the system (source code and model):

- IVisitorVertex: Interface allowing to explore a tree in RAM. The visitor pattern is applied with the aim of separating the actions from the data structure.
- *VisitorVertexGraphToCode*: it allows updating source code from the tree. It inherits the *IVisitorVertex* interface.
- *VistorVertexGraphToXMI*: it allows updating the XMI file from the tree. It inherits the *IVisitorVertex* interface.

This modeling stage preceedes that of the implementation from use cases presented above. Insofar as this paper falls under the perspectives of OptimaDev [1], [13], we exploit the same tools or programming language used for its realization.

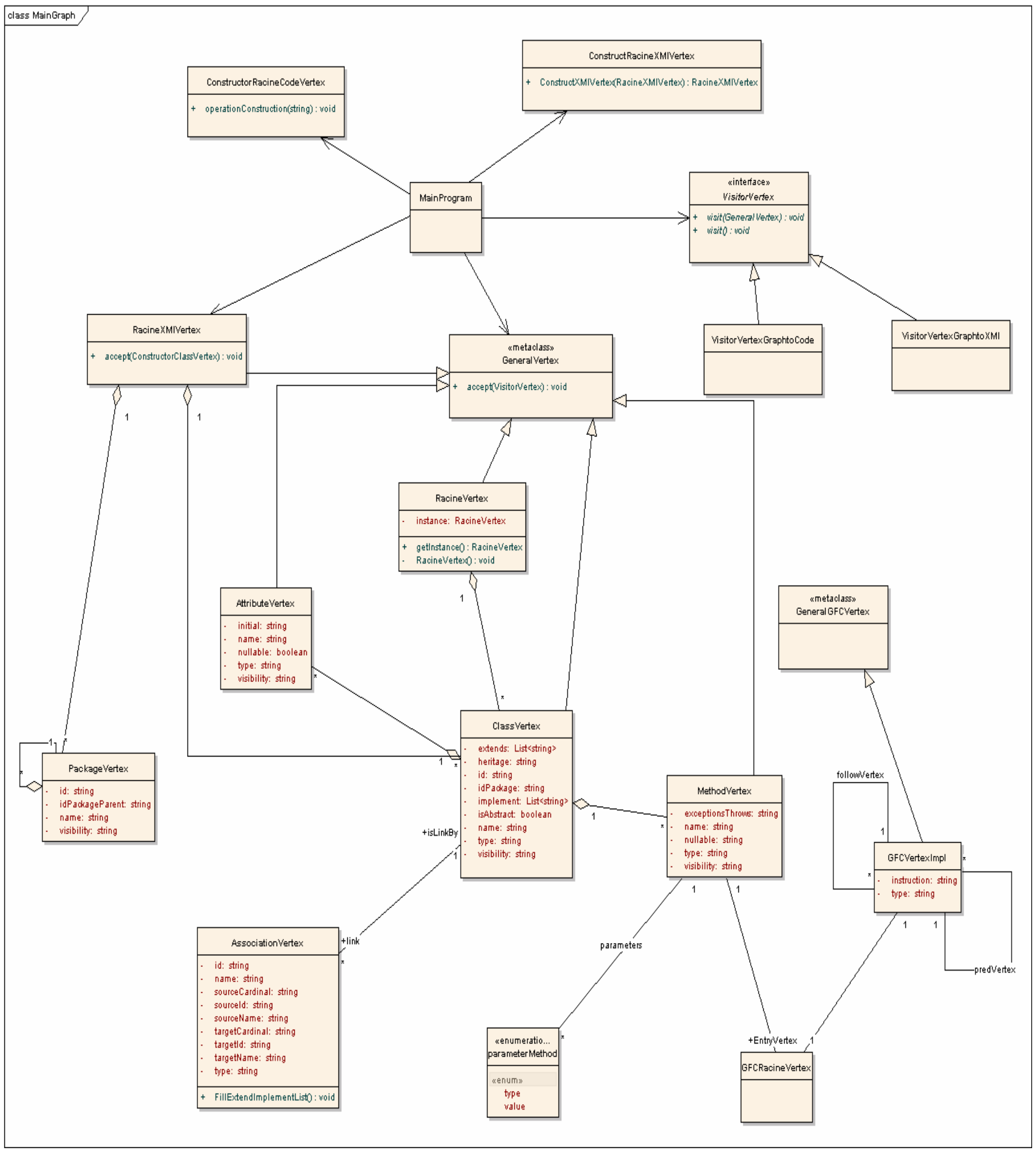


Figure 2. Class diagram of the system

4 Realization of an archetype and obtained results

4.1 Implementation environment

The principal tools which we used for the implementation are: Visual Studio 2005; Enterprise Architect 6.5; Log4net; Antlr version 3 [12]. To successfully complete this realization, we began by analyzing both input data of the process which are: the source code and the model. For the source code, we used Java and C# programming languages and for the model, the XMI file generated by Enterprise Architect. The result of the intersection of these two input data allowed us to constitute the tree used to perform synchronization.

In term, we implemented the Round-Trip for the Java language by taking into account the class diagrams. We integrated it into OptimaDev [1], [13]. Steps 1, 2, 3, 4, 6 of our method described in the section 3.1 work as planned. Step 5 works for some type of precise class because Antlr lacks loading all the source code in RAM. For example, it does not load comments because they are ignored by the class parsor. At step 5, the class is entirely rewritten from the source code into RAM. The comments are lost after rewriting.

New rules of structuring of source code were brought for the implementation of the Round-Trip. For the Java, the rules are the following: a comment has been introduced to represent the hierarchy of the class' packages in the model. This comment must be placed in first in the class file concerned as we illustrate by following example:

```
//@packageModel("Logical View\Analysis Model\Business
Component\Personne\sisv\")
```

Annotations were added above elements to be updated in the model. They correspond to the identifiers of these elements in the model as illustrated by following example:

```
  @objectid("EAID_31ADFD69_D4AA_4247_A4D3_51B1611574B0")
  public interface IPersonneSISV
```

4.2 Implementation of the Round-Trip Engineering [23]

Our starting point of application of the RTE is in the OptimaDev development environment. It is possible to decide to accomplish it either on a subset or subpart of the project constituted in a directory or folder (see Figure 3), or on a source code file from the document explorer simply from context menus on the concerned object. In the case of a source code file, the right click can be made on its opened contents or on the icon of aforementioned file in the document explorer.

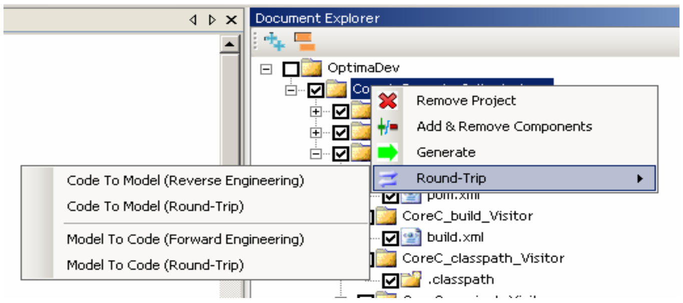
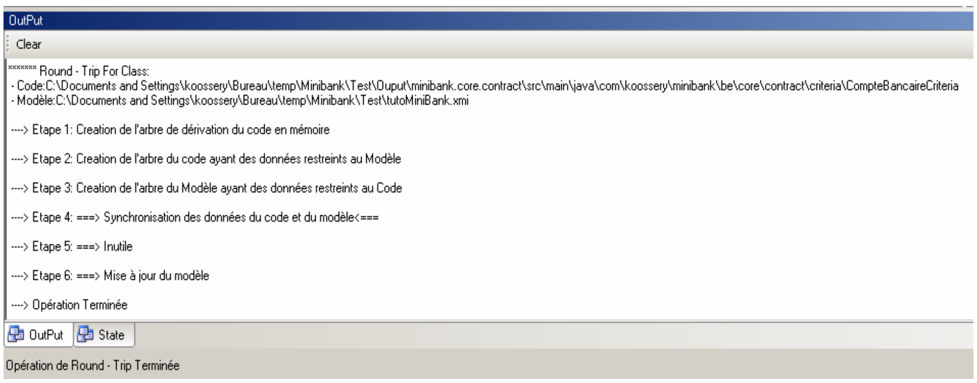


Figure 3: Context menu of the Round-Trip on a directory

The following Figure 4 illustrates the capacity of our solution to follow the trace of the execution of the Round-Trip from OptimaDev [1], [13].



```
******* Round - Trip For Class:
 - Code:C:\Documents and Settings\koossery\Bureau\temp\Minibank\Test\Ouput\minibank.core.contract\src\main\java\com\koossery\minibank\be\core\contract\criteria\CompteBancaireCriteria
 - Modèle:C:\Documents and Settings\koossery\Bureau\temp\Minibank\Test\tutoMiniBank.xmi

 ----> Etape 1: Creation de l'arbre de dérivation du code en mémoire

 ----> Etape 2: Creation de l'arbre du code ayant des données restreints au Modèle

 ----> Etape 3: Creation de l'arbre du Modèle ayant des données restreints au Code

 ----> Etape 4: ===> Synchronisation des données du code et du modèle<===

 ----> Etape 5: ===> Inutile

 ----> Etape 6: ===> Mise à jour du modèle

 ----> Opération Terminée
```

Opération de Round - Trip Terminée

Figure 4: Output of a Round-Trip Operation from OptimaDev

5 Conclusion

We meet in this paper a need of application or software developers computer engineering services-based companies by offering a process based on Round-Trip engineering (RTE), to automate the synchronization of the source code and the model in the project development lifecycle. RTE is a combination of both forward and reverse engineering. For its design and its implementation, we took into account these two aspects.

In the phase of implementation, the main difficulty is currently the implementation of the stage 5 of our solution which consists in the update of the source code due to the lacks and limits of Antlr. So, to reconstruct or reconstitute the source code, we rewrote the file of the modified class. The second difficulty was to create a grammar for the C# language. It was necessary to start from the specifications given by ECMA-334, [16].

We plan to include our results into a MDA engine like OptimaDev. This integration promises in the perspectives of validation of optimization at large scales a benefit or profit of the costs of a progressive or evolutionary maintenance of more than 80%. An ongoing experimentation on some cases randomly chosen in a software industry Koossery Tech Cameroon is in process of confirming it.

ACKNOWLEDGEMENTS

Special acknowledgements go Koossery Tech Cameroon, our industrial partner to have providing us a real test environment.

References

[1] T. Djotio Ndie, C. Tangha, Fritz Ekwoge Ekwoge, MDA (Model-Driven Architecture) as a Software Industrialization Pattern: An Approach for a Pragmatic Software Factories, *J. Software Engineering & Applications (JSEA)*, **3**(6), (2010), 561-571.

[2] D.S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*), Wiley, 2003.

[3] J. McGovern, *A Practical Guide to Enterprise Architecture <http://www.amazon.com/Practical-Guide-Enterprise-Architecture/dp/0131412752/ref=cm_lmf_tit_5_rsrsrs0>*, Prentice Hall PTR, 2003.

[4] F.D. Weekes, *LAX to JFK and Back: Enough Reading for the Round Trip*, Authors Choice Press, 2005.

[5] A. Cassidy, *A Practical Guide to Information Systems Strategic Plannin*, 1st Ed, CRC Press, 1998.

[6] Norman S. Nise, *Control Systems Engineering*, 5th Ed., Wiley , 2007.

[7] K. Lano, *Advanced Systems Design with Java, UML and MDA*; Butterworth-Heinemann, 1st Ed, 2005.

[8] D. Gasevic, D. Djuric and V. Devedzic, *Model Driven Engineering and Ontology Development*, 2nd Ed, Springer, 2009.

[9] Mathias Fritzsche1 et al, *Towards Utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis* , proceedings of LNCS, ECMDA-FA, 1st Ed Springer, (2008).

[10] Fabio Perez Marzullo et al, *A Practical MDA Approach for Autonomic Profiling and Performance Assessment*, proceedings of LNCS, ECMDA-FA, 1st Ed Springer**,** (2008).

[11] Andreas Ulrich and Alexandre Petrenko, *Reverse Engineering Models From Traces to ValidateDistribueted Systems- An insdustrial Case Study*, Proceedings of LNCS-4530, Springer, (2007).

[12] Terrence Parr, *The Definitive ANTLR Reference Building Domain-Specific Language* , Pragmatic Bookshelf, 2007.

[13] E.E. Fritz, *Pragmatic Software Factories: Industrialization of the Development of Software*, Masters of Thesis of the National Advanced School of Engineering, University of Yaounde 1, 2007.

[14] R. Delamare, *Rétro-ingénierie des modèles dynamiques d'UML*; Equipe Triskell, IRISA, IFSIC, Université de Rennes 1, (ftp://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2006/rapbiblio_delamare_romain.pdf , consulted the 3rd of April2009), 2006.

[15] A. Henriksson, H. Larsson, *A Definition of Round-trip Engineering*, (http://www.ida.liu.se/~andhe/re.pdf, consulted the 22nd of April 2009), 2009.

[16] Steven E. Weisler and M. Slavko, *Theory of Language* (*Bradford Books*), The MIT Press; illustrated edition, 1999.

[17] ECMA-334, *C# Language Specification,* 4th Ed, ECMA International, (http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf, consulted the December 05th 2009), 2006.

[18] Ira D. Baxter and M. Mehlich, *Reverse Engineering is Reverse Forward Engineering,* 4th Working Conference on Reverse Engineering (WCRE '97) Computer Society, (1997).

[19] Dofactory, *Visitor Pattern in C# and VB.NET* (http://www.dofactory.com/Patterns/PatternVisitor.aspx, consulted the April 4th 2009), 2009.

[20] C. Tommie, C. Wolfe and C. Ludet, *UML Model XMI Files - the Roundtrip* (https://gforge.nci.nih.gov/frs/download.php/931/Roundtrip09142006v2.ppt , consulted the December 25th 2009), 2006.

[21] SEI Pekin, *An Architecture Based Round-trip Approach towards Component Composition*, Software Engineering Institute, Peking University, (www.ipl.t.u-tokyo.ac.jp/~xiong/UTPKUWorkshop/res/song.ppt, consulted the March 24th 2009), 2009.

[22] L. Audibert, *UML 2.0.*, IUT de Villetaneuse - Département Informatique, (http://www-lipn.univ-paris13.fr/audibert/pages/enseignement/cours.htm, consulted the April 13th 2009), 2009.

[23] Raphaël-Ledoux Fongang, *Optimisation de la maintenance des applications: Round-Trip Engineering dans un moteur MDA,* Masters of Thesis of the National Advanced School of Engineering, University of Yaounde 1, 2009.