Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System

Leonardo Murta Chessman Corrêa João Gustavo Prudêncio Cláudia Werner
Federal University of Rio de Janeiro - COPPE - System Eng. and Computer Science
P.O. Box 68511 - Rio de Janeiro, RJ 21945-970 Brazil

{murta, chessman, gustavo, werner}@cos.ufrj.br

ABSTRACT

Models are becoming first class artifacts in Software Engineering. Due to that, an infrastructure is needed to support model evolution in the same way we have for source-code. One of the key elements of such infrastructure is a version control system properly designed for models. In previous work, we presented Odyssey-VCS, a version control system tailored to fine-grained UML model elements. In this paper, we discuss the main improvements that we are incorporating on the second release of this system, which are: support for UML 2, reflective processing, explicit branching and auto-branching, generic merge algorithm, support for pessimistic concurrency policy, and support for hooks.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – *computer-aided software engineering (CASE), object-oriented design methods.*
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *version control.*
D.2.9 [Software Engineering]: Management – *software configuration management.*

General Terms

Management, Design.

Keywords

Unified Modeling Language (UML), Version Control.

1. INTRODUCTION

In the last years, model-driven development has emerged as an important technique for software development. Model-driven approaches focus on the definition of high level models and apply subsequent transformations to obtain implementation artifacts [1]. One of the most known organizations behind model-driven development is Object Management Group (OMG) [28], which advocate the use of Unified Modeling Language (UML) [29, 30] among other standards as part of a software design approach named Model-driven Architecture (MDA) [20]. Due to this scenario, UML is becoming more than a software documentation notation. UML diagrams can now be transformed into source-code or even automatically executed [19].

CVSM'08, May 17, 2008, Leipzig, Germany.

However, this new scenario brings together new challenges: in the past an enormous infrastructure has been built to develop and maintain source-code, which does not properly work for models. One of the key elements of this infrastructure is the version control system, which is responsible for keeping a software system consisting of many versions and configurations well organized [34].

The existing version control systems, conceived for dealing with source-code, were intentionally designed to be generic [11], avoiding language-specific support. For example, most current version control systems are based on file system structures, while modeling languages are based on higher level structures. The mapping of these complex structures used by modeling languages to file structures is dangerous due to concept mismatch. Nevertheless, the Software Configuration Management (SCM) community has already detected that many unaddressed research issues rely on breaking the assumption of generic and language independent SCM [11].

In previous work, we presented Odyssey-VCS [21, 24], a version control system tailored to fine-grained UML model elements. During the design of Odyssey-VCS, we aimed to overcome some of the challenges described in the SCM literature [10], such as: (1) data model that deals with complex objects; (2) homogeneous versioning for different types of objects; (3) distributed and heterogeneous workspaces; and (4) concurrent engineering with high level models.

In this paper, we present some ongoing improvements over Odyssey-VCS. These improvements encompass: (1) support for UML 2, (2) reflective processing, (3) explicit branching and auto-branching, (4) generic merge algorithm, (5) support for pessimistic concurrency policy, and (6) support for hooks.

The rest of this paper is organized as follows. Section 2 briefly describes the first release of Odyssey-VCS. Section 3 details the improvements being implemented on Odyssey-VCS 2. Section 4 present some related work, and we conclude the paper in Section 5 with an outlook at our future work.

2. ODYSSEY-VCS

The main goal of the first release of Odyssey-VCS was to provide a version control system that aids architects in the concurrent modeling of software systems using heterogeneous UML-based CASE tools. To achieve this goal, Odyssey-VCS adheres to well adopted specifications, such as Meta Object Facility (MOF) [25], UML, XML Metadata Interchange (XMI) format [27], and Java Metadata Interface (JMI) [7].

Odyssey-VCS was conceived to work in a similar way of popular file-based version control systems, such as Subversion [5]. Models

are checked-out from a central server, changed in parallel by different users in private workspaces, and finally checked-in back to the repository. The main concerns of Odyssey-VCS reside on server-side and include concurrency control and model versioning. Due to that, the visual presentation of model differences and client-side model editing are lead to external tools, such as CASE tools. It is worth to notice that Odyssey-VCS is not tailored to a specific UML diagram. It works over UML models and is able to version any kind of information contained in these models.

The communication among UML-based CASE tools and Odyssey-VCS takes place via Web Services [3]. The CASE tools externalize UML models as XMI files and stream these files to Odyssey-VCS through Web Services calls. At server-side, Odyssey-VCS loads these XMI files into a MOF Repository named MDR [18] and manipulates the UML models via JMI API. Each XMI file, which represents a specific version of a UML model, becomes an extent in the MDR repository.

Odyssey-VCS has its own versioning model. This model, which is implemented as a MOF meta-model, is responsible for storing versioning information and linking this version information to the data model. In our case, the data model is an instance of the UML meta-model. Figure 2 shows this scenario, highlighting that each UML model element version will be linked by the Odyssey-VCS model inside MDR. This allows us to perform further querying and retrieval.

Finally, Odyssey-VCS maintains a per-project behavior descriptor that informs how each UML model element type should be handled. This behavior descriptor determines whether evolution information is needed or not for a specific UML model element. This evolution information comprises a unique version identification and auxiliary contextual information, such as who changed the element, when it was changed, and why it has been changed. Moreover, this behavior descriptor also indicates which elements are considered atomic for conflict detection purpose. Odyssey-VCS raises a conflict flag when two or more developers concurrently change an element that is considered atomic.

3. ODYSSEY-VCS 2

The second release of Odyssey-VCS encompasses multiple improvements over the first release. These improvements, which are discussed in the next sections, demanded some changes in the Odyssey-VCS versioning model. Figure 1 shows a scrap of the Odyssey-VCS versioning model, composed by five mains classes: configuration item, version, transaction, user and model element.

Each configuration item is composed by versions. Each version has relationships to the next and previous versions, which can be null for the first and last versions of a configuration item, respectively. A specific attribute differentiates versions that were deleted by the user. In addition, it also has relationships to branched and merged versions, which allow non-sequential development.

Versions are queried or created by transactions. We currently support

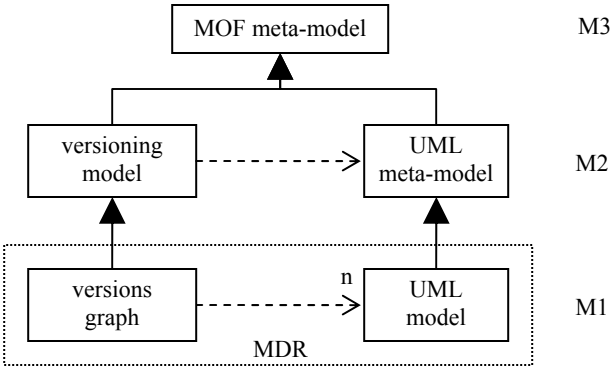

Figure 2. Relationship between Odyssey-VCS and UML

both read-only and read-write transactions, such as history, check-out, and check-in. It is important to notice that the explicit link among read-only transactions and versions allows us to increase awareness support of Odyssey-VCS: it is possible to know which model elements where checked-out or had its history analyzed by a specific user.

Finally, versions have relationships to model elements that cross the Odyssey-VCS versioning model border. This relationship connects the Odyssey-VCS versioning model with the UML meta-model, as shown in Figure 2. The *EModelElement* class belongs to the M3 level and is extended by meta-models at M2 level. Due to that, UML model elements, which inherit from this class, are subject for versioning according to Odyssey-VCS approach.

3.1 Support for UML 2

The first release of Odyssey-VCS can be seen as a proof-of-concept prototype, which allowed us to experiment and better understand the real challenges of model versioning. This release had some performance issues, mainly due to overheads imposed by MDR [21]. On top of that, the MDR project is compliant to MOF 1.4, preventing us to support UML 2. Due to that, we replaced MDR by a new infrastructure for meta-modeling in the second release of Odyssey-VCS.

This new infrastructure, named Eclipse Modeling Framework (EMF) [8], has a specific module that provides an implementation for the UML 2 meta-model [9]. Additionally, it allows us to interact with the meta-model via an API similar to JMI. It is also pos-

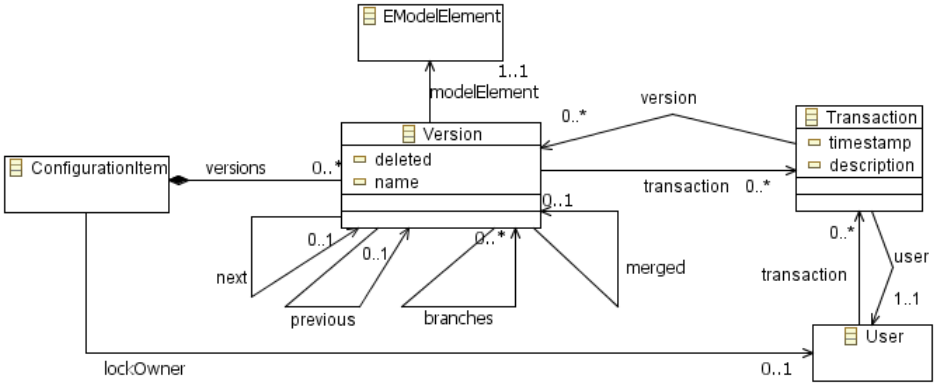

Figure 1. Scrap of the Odyssey-VCS versioning model

sible to create our own meta-model and to export models in terms of XMI files.

EMF and MDR share many similarities. Both have the concept of extent for loading and saving XMI files and provide API to manipulate the models. Moreover, both allow transient and non-transient repositories. In the case of MDR, non-transient repositories use a B-Tree implementation. On the other hand, EMF uses XMI files or a relational database mapping for non-transient repositories.

Most version control systems stores some client-side information to help workspace control. For instance, CVS [12] and Subversion [5] use directories *CVS* and *.svn*, respectively. In our case, we use UML Profiles for performing this task. Our profile has a stereotype named *OdysseyVCSElement* that stores the base element ID and the element version. The base element ID is the ID of the checked-out model element. With this kind of information, Odyssey-VCS is able to correctly identify the checked-out element during check-in and merge. In addition, keeping the base element ID in the UML model elements allows us to correctly track renames, moves, and copies of UML model elements.

3.2 Reflective Processing

The first version of Odyssey-VCS used handlers to deal with specific types of model elements. A handler has methods for providing new instances and for processing instances of a specific type. For example, the *ClassHandler* is a handler for the *Class* type. It knows how to create a new *Class* instance and how to process classes, which consists on processing primitive properties of the *Class* type and recursively processing *attributes* and *operations*.

Due to this design decision, we had to implement a handler class for each model element type in the UML meta-model. This led us to an incomplete support of the UML meta-model and difficulties to upgrade the meta-model. For instance, we needed to check all model elements types during meta-model upgrade, and create new handle classes to process new types.

However, these handle classes hold a high degree of similarity. All of them create instances of a specific type, process the primitive attributes of the type (such as *name* in the case of *ClassHandler*) and recursively run for the relationships of the type (such as *attributes* and *operations*, among others, in the case of *ClassHandler*).

Fortunately, the M3 layer of EMF, named Ecore, provides a reflective API. This API allows the type identification of a specific model element (modelElement.eClass()), new instances creation for specific types (UMLFactory.eINSTANCE.create(type)), attributes identification for a specific type (type.getEAllAttributes()), and relationships identification for a specific type (type.getEAllReferences()). Table 1 and Table 2 summarize some methods of the Ecore reflective API for types and instances, respectively.

The second release of Odyssey-VCS uses this API for its branching and merge algorithms, allowing us to get rid of the previous handler structure. This improvement has two major benefits: (1) it makes easier to upgrade the UML meta-model, because our versioning model and algorithms have fewer explicit dependencies to a specific release of the UML meta-model, and (2) it opens some possibilities for generic meta-model versioning, if our versioning model and algorithms become generic enough to work over any Ecore-based meta-models.

Table 1. Summary of the reflective API for types

Method	Description
getEAllSuperTypes()	Provides all local and inherited super types of a specific type
getEAllAttributes()	Provides all local and inherited attributes of a specific type
getEAllOperations()	Provides all local and inherited operations of a specific type
getEAllReferences()	Provides all local and inherited references of a specific type
getEAllContainments()	Provides all local and inherited containment reference of a specific type

Table 2. Summary of the reflective API for instances.

Method	Description
eClass()	Provides the type of a specific instance
eGet(EStructuralFeature)	Provides the value of an attribute or reference of a specific instance
eSet(EStructuralFeature, Object)	Defines the value of an attribute or reference of a specific instance

3.3 Branching

The first release of Odyssey-VCS has no explicit support for branching. However, it was able to track copies of model elements due to the base element ID discussed in Section 3.1. When a checked-out model element is copied to another part of the model, the identification of the original model element, which is stored in a stereotype attached to the element, is also copied together with the copied model element. This allows Odyssey-VCS to correctly keep the history of the copied elements during check-in.

Currently, we are working on a solid support for branching for the second release of Odyssey-VCS. We identified two type of branch: implicit and explicit branching. The implicit branching support consists on the previously described situation. There is no explicit command for creating a branch and the branch does not occur by forking the whole data model, but only by coping fragments of the data model to other locations inside the data model. Subversion is an example of an existing version control system that provides this kind of support for branching. The explicit branching support consists on branching the whole data model due to an explicit call to a branch command. CVS is an example of an existing version control system that provides this kind of support for branching.

Both branching supports are useful for specific situations, and they can be combined to provide a more powerful solution to the users. For example, after finishing analysis, an architect may copy the analysis model into another package, and start the design. In this scenario, implicit branching was used because both analysis and design models belong to the same system and both branches will live forever. However, if an architect wants to test a new module on the system, an explicit branch should be used. It is worth to notice that both “system without the new module” and “system with the new module” are variants of the same system.

One of them will be kept, and the other will be discarded in the future.

The explicit support for branching in Odyssey-VCS also allows us to implement the concept of auto-branching. Auto-branching consists on automatically creating a branch for every check-out/change/check-in cycle. Most current version control systems, with few exceptions [2], looses information during parallel development (two or more people changing the same artifact at the same time). This occurs because users are forced to merge before they check-in. Due to that, there is no way to identify the original change intention of the user, but only the change already merged with changes performed by other users. With auto-branching, we can store both the intention and the merged changes.

This auto-branching feature can be easily introduced in Odyssey-VCS by changing the current check-in algorithm. The new check-in algorithm will consist on performing a branch, applying the original change to the branch, and attempting to merge these changes back to the main line of development. If the merge fails, the user will be forced to manually merge his changes with changes made by other users. The new check-in with merged changes will go directly to the main line of development, finishing the auto-branching lifecycle, as shown in Figure 3 (auto-branching with dashed lines).

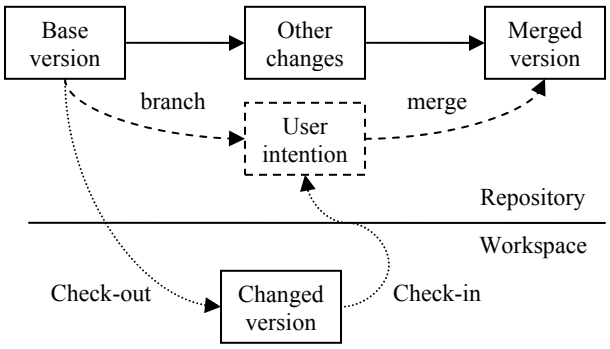

Figure 3. Auto-branching support

3.4 Merge

The improvements described in Sections 3.2 and 3.3 allowed some additional improvements to the merge algorithm. The new merge algorithm replaced the old type-specific merge algorithm by applying a generic strategy to all model element types via the reflective API. It also takes advantage of auto-branching for difference detection.

The new merge algorithm follows a generic 3-way merge approach [6], receiving the base (B), source (S), and target (T) versions of a model element as arguments and returning the merged version (M). The base version is the greater common version between both paths (B → S and B → T). The source and target version are the versions being merged. It is worth to notice that the ID support discussed in Section 3.1 is used to perform all necessary matches among model elements.

The algorithm can be decomposed into three main parts: (1) existence analysis, (2) attributes processing, (3) relationships processing. The existence analysis verifies the arguments and decides weather the algorithm should continue, finish or raise a conflict. Table 3 shows the possible results for this analysis. It is important to notice that the *M = null* statement represents the element deletion in the merged version without further processing.

Table 3. Existence analysis

∃ B	∃ S	∃ T	Result
True	True	True	Merge algorithm continues (see attributes processing in Table 4)
True	True	False	If S == B then M = null Else raises conflict: "source changed and target deleted"
True	False	True	If T == B then M = null Else raises conflict: "target changed and source deleted"
True	False	False	M = null
False	True	True	Impossible (different elements cannot have the same ID)
False	True	False	M = S
False	False	True	M = T
False	False	False	M = null

After performing the existence analysis, the merge algorithm may continue and run the attributes processing. In this phase, the algorithm aims to merge the attributes of each model element. Table 4 shows the possible results for this analysis for each attribute.

Table 4. Attributes processing

S == B	T == B	S == T	Result
True	True	True	M = T (or M = S or M = B)
True	True	False	Impossible (transitive relation)
True	False	True	Impossible (transitive relation)
True	False	False	M = T
False	True	True	Impossible (transitive relation)
False	True	False	M = S
False	False	True	M = T (or M = S)
False	False	False	Raises conflict: "same attribute changed in both source and target"

Finally, the relationships processing considers both containment and non-containment relationships. In the case of containment relationships, changes in the other end of the relationship represent changes in the model element being processed. For example, a class has containment relationships to its operations. If an operation changes, the class is indirectly changed. On the other hand, only additions and removals of non-containment relationships change the model element being processed. For example, suppose that class *A* has an association with class *B* (non-containment relationship). Changes in class *B* are not propagated to class *A*. However, the creation of a new association between classes *A* and *C* indirectly changes class *A*.

Besides this change propagation issue, the merge algorithm is recursively called for both containment and non containment relationships. The results of these recursive calls are assigned to the relationship field of the model element being processed.

3.5 Concurrency Policies

The first release of Odyssey-VCS only allowed the use of optimistic concurrency policy. However, in some situations, such as

refactorings, it is desired to prohibit other architects to change a specific part of the model for a period of time. In this case, a pessimistic concurrency policy is handy.

Fortunately, such pessimistic concurrency policy can be implemented in Odyssey-VCS without changing its versioning model. A lock command, if applied on a specific model element, results on the creation of a new *transaction* instance (see Figure 1) that associates the locked configuration item with the user who performed the lock. Moreover, the lock *transaction* holds the reason why a lock was needed.

During the lock period, no check-in command is accepted, except the check-in command performed by the lock owner. In this case, the lock is automatically removed. Another way to remove the lock is the explicit use of the unlock command.

3.6 Hooks

Another weak aspect in the first release of Odyssey-VCS is regarding extension mechanisms. This release does not provide mechanisms to trigger external tools in response to specific events. This kind of support is usually needed for performing external tasks prior or as a consequence of an event.

A common strategy for implementing this extension mechanism in version control systems is using the Observer pattern [13] associated to specific events. These events are usually related to the moments before and after the execution of the version control commands. For instance, Subversion provides support for pre and post events for commit, property change, lock, and unlock.

We intent to introduce a generic hook support for the second release of Odyssey-VCS. This support will allow us to implement any pre and post commands hooks in the future. In addition, we are currently working on a specific support for pre-check-in and post-check-in hooks. The hooks for the other commands will be introduced in the next releases.

4. RELATED WORK

Most commercial and open-source version control systems are based on file system data model [5, 12, 32, 35]. Besides being generic and mature, these version control systems have several limitations to manipulate artifacts with complex internal data model, such as UML. These limitations include inability to perform fine-grained versioning of model elements or to consistently merge and detect conflicts among model elements [21]. We do not see these approaches as direct competitors of our approach. Our approach can be used to version UML model elements while these approaches can be used to version source-code artifacts in the same software development project.

There are also other approaches that employ other data models, such as entity-relationship or even object-oriented. However, these approaches work at source-code level and are focused on a specific programming language. For instance, Goldstein et al. [15], Habermann et al. [16], and Render et al. [31] support versioning of Smalltalk, C and Pascal source-code, respectively. Our approach can also be seen as complementary to these approaches since we are focused on UML model elements and these approaches are focused on source-code.

Some few approaches use non file system data model to version analysis and design artifacts. For instance, Ohst et al. [23] propose an approach for versioning analysis and design artifacts via syntax trees stored in XML files. Working at fine grained UML artifacts they can correctly manage structural changes on these artifacts. However, the usage of XML does not mean adherence to modeling standards. Their XML format does not follow XMI specification for UML, leading to incompatibilities with existing UML-based CASE tools. Moreover, they provide no support for workspace management and merging. On the other hand, Nguyen et al. [22] use a hyper-versioning system to apply version control over complex artifacts, including UML analysis and design artifacts. This work has a strong focus on versioning relationships among the elements. However, it is also based on a proprietary UML data model, reducing compatibility with existing CASE tools.

Some CASE tools, such as Enterprise Architect [33], Borland Together [4], Rational Software Architect [17], and Poseidon [14], provide integration with existing file-based version control system. Usually, they break the model apart, into small files, and store these files in a file-based version control system. This strategy allows the use of pessimistic concurrency control by locking these files. Moreover, they usually apply external tools to merge model elements, in the case of optimistic concurrency control. Besides providing some support for model versioning, this strategy heavily relies on client-side tools, making it difficult the replacement or even the concomitant usage of different CASE tools. In addition, the repository is usually spoiled with many unrecognizable model and control files, which cannot be considered textual files by the version control system. This makes almost useless the standard toolset shipped together with the version control systems.

Finally, OMG has defined a specification for MOF versioning [26]. Although not compliant to this specification, it is possible to notice that Odyssey-VCS share the specification philosophy. Similar to the specification, Odyssey-VCS has its own versioning model. Moreover, it stores the versioned elements into separate per-version extents with associated history of changes. It seems feasible to adapt the Odyssey-VCS versioning model to adhere to this specification in the future.

5. CONCLUSION

We presented in this paper the ongoing improvements for the second release of Odyssey-VCS. The main enhancement is the adoption of EMF, allowing us to support UML 2.1 and XMI 2.1 specifications. However, we have also presented some other new features, such as reflective processing, explicit branching and auto-branching, generic merge algorithm, support for pessimistic concurrency policy, and support for hooks.

After finishing the implementation phase, we intent to start the testing phase, together with some performance evaluation. The results of the performance evaluation can be compared with the results obtained by the first release of Odyssey-VCS [21]. We are expecting some performance improvements due to the replacement of MDR to EMF and the use of Ecore reflective API on the merge algorithm.

As future work, we intend to deeply analyze the MOF versioning specification and try to adhere to this specification. Moreover, another future work is to generalize the existing UML versioning model references (most of these references were removed with reflective processing) and transform Odyssey-VCS into a generic

version control system for MOF-based (or Ecore-based) metamodels.

6. ACKNOWLEDGMENTS

Our thanks to the members of the Software Reuse Group at COPPE/UFRJ, especially Hamilton Oliveira, Cristine Dantas, and Luiz Gustavo Lopes, who contributed to the first release of Odyssey-VCS. Moreover, we would like to thank CNPq for the financial support under grant PDJ 150346/2007-7.

7. REFERENCES

[1] Beydeda, S., Book, M. and Gruhn, V. *Model-Driven Software Development*. Springer, 2005.

[2] BitMover, 2008, "BitKeeper". In: http://www.bitkeeper.com, accessed in January 23, 2008.

[3] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. and Orchard, D., 2005, "Web Services Architecture - W3C Working Group Note". In: http://www.w3.org/TR/ws-arch, accessed in April 16, 2006.

[4] Borland, 2008, "Borland Together". In: http://www.borland.com/us/products/together/index.html, accessed in January 16, 2008.

[5] Collins-Sussman, B., Fitzpatrick, B.W. and Pilato, C.M. *Version Control with Subversion*. O'Reilly, 2004.

[6] Conradi, R. and Westfechtel, B. Version Models for Software Configuration Management. *ACM Computing Surveys*, *30* (2). 232-282.

[7] Dirckze, R. Java Metadata Interface (JMI) Specification - Version 1.0, Unisys Corporation and Sun Microsystems, 2002.

[8] Eclipse Foundation, 2008, "Eclipse Modeling Framework (EMF)". In: http://www.eclipse.org/emf, accessed in January 23, 2008.

[9] Eclipse Foundation, 2008, "EMF-based UML 2.x Metamodel Implementation". In: http://www.eclipse.org/uml2, accessed in January 23, 2008.

[10] Estublier, J., Software Configuration Management: a Roadmap. in *International Conference on Software Engineering (ICSE), The Future of Software Engineering*, (Limerick, Ireland, 2000), 279-289.

[11] Estublier, J., Leblang, D., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W. and Wiborg-Weber, D. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *14* (4). 1-48.

[12] Fogel, K. and Bar, M. *Open Source Development with CVS*. The Coriolis Group, Scottsdale, Arizona, USA, 2001.

[13] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[14] Gentleware, 2008, "Poseidon for UML". In: http://www.gentleware.com, accessed in January 23, 2008.

[15] Goldstein, I.P. and Bobrow, D.G. A Layered Approach to Software Design. in Barstow, D.R., Shrobe, H.E. and Sandewall, E. eds. *Interactive Programming Environments*, McGraw-Hill, New York, NY, 1984, 387-413.

[16] Habermann, A.N. and Notkin, D. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering (TSE)*, *12* (12). 1117-1127.

[17] IBM, 2008, "Rational Software Architect". In: http://www-306.ibm.com/software/awdtools/architect/swarchitect/, accessed in January 16, 2008.

[18] Matula, M., 2008, "NetBeans Metadata Repository". In: http://mdr.netbeans.org, accessed in January 23, 2008.

[19] Mellor, S.J. and Balcer, M.J. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[20] Miller, J. and Mukerji, J. MDA Guide Version 1.0.1, Object Management Group, 2003.

[21] Murta, L.G.P., Oliveira, H.L.R., Dantas, C.R., Lopes, L.G.B. and Werner, C.M.L. Odyssey-SCM: An integrated software configuration management infrastructure for UML models. *Science of Computer Programming*, *65* (3). 249-274.

[22] Nguyen, T.N., Munson, E.V. and Boyland, J.T., The molhado hypertext versioning system. in *Conference on Hypertext and Hypermedia*, (Santa Cruz, USA, 2004), 185-194.

[23] Ohst, D. and Kelter, U., A Fine-grained Version and Configuration Model in Analysis and Design. in *International Conference on Software Maintenance (ICSM)*, (Montreal, Canada, 2002), 521-527.

[24] Oliveira, H.L.R., Murta, L.G.P. and Werner, C.M.L., Odyssey-VCS: a Flexible Version Control System for UML Model Elements. in *International Workshop on Software Configuration Management (SCM)*, (Lisbon, Portugal, 2005), 1-16.

[25] OMG. Meta Object Facility (MOF) Specification, version 1.4, Object Management Group, 2002.

[26] OMG. Meta Object Facility (MOF) Versioning and Development Lifecycle Specification, v2.0, Object Management Group, 2007.

[27] OMG. MOF 2.0/XMI Mapping, Version 2.1.1, Object Management Group, 2007.

[28] OMG, 2008, "Object Management Group". In: http://www.omg.org, accessed in January 23, 2008.

[29] OMG. Unified Modeling Language (UML) Infrastructure Specification, version 2.0, Object Management Group, 2006.

[30] OMG. Unified Modeling Language (UML) Superstructure Specification, version 2.0, Object Management Group, 2005.

[31] Render, H. and Campbell, R., An Object-oriented Model of Software Configuration Management. in *International Workshop on Software Configuration Management (SCM)*, (Trondheim, Norway, 1991), 127-139.

[32] Roche, T. and Whipple, L.C. *Essential SourceSafe*. Hentzenwerke Publishing, 2001.

[33] Sparx Systems, 2008, "Enterprise Architect". In: http://www.sparxsystems.com/products/ea.html, accessed in January 23, 2008.

[34] Tichy, W. RCS: a system for version control. *Software - Practice and Experience*, *15* (7). 637-654.

[35] White, B.A. *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction*. Addison-Wesley, 2000.