Experiences with Modeling in the Large in Fujaba

Thomas Buchmann
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
thomas.buchmann@uni-bayreuth.de

Alexander Dotor
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
alexander.dotor@uni-bayreuth.de

Bernhard Westfechtel
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.de

ABSTRACT

Model-driven software development intends to reduce development effort by generating code from high-level models. However, models for non-trivial problems are still large and require sophisticated support for modeling in the large. Experiences from a recently launched project dedicated to a model-driven modular SCM system confirm this claim. This paper investigates modeling in the large support provided by the object-oriented CASE tool Fujaba and discusses potential extensions of Fujaba based on UML package diagrams.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software architectures—*languages*; D.2.11 [**Software Engineering**]: Management—*software configuration management*

Keywords

version control, packages, imports

1. INTRODUCTION

Software configuration management (SCM) is the discipline of controlling the evolution of large and complex software systems. A wide variety of SCM tools and systems has been implemented, ranging from small tools such as RCS [10] over medium-sized systems such as CVS [11] or Subversion [4] to large-scale industrial systems such as ClearCase [12].

Version control is a core function of any SCM system. Version control is based on *version models*, many of which have been implemented in research prototypes, open source products, and commercial systems [5]. While there are considerable differences among these version models, it is also true that similar concepts such as revisions, variants, state- and change-based versioning appear over and over again. Unfortunately, version models are usually implicitly contained in implemented systems.

Thus, the SCM domain is characterized by a large number of systems with more or less similar features incorporating hard-wired version models which have been implemented with considerable effort. This observation has motivated us to set up a project dedicated to a *model-driven modular SCM system* (abbreviated as *MOD2-SCM* [2]):

First, *version models* are defined *explicitly* rather than implicitly in the code. This makes it easier to communicate and reason about version models. Second, modeling comprises both *structure* and *behavior*. Furthermore, behavioral models are executable. Third, productivity is improved by replacing programming with the creation of *executable models*. Fourth, version models are not created from scratch. Rather, reuse is performed on the modeling level by following a *product line* approach [3]. Finally, the product line is based on a *model library* which is composed of reusable and loosely coupled architectural units.

In MOD2-SCM, we decided to use the object-oriented modeling language and environment *Fujaba* [15] because it supports generation of executable (Java) code from a UML model. To date, only a few approaches have been dedicated to *model-driven development of versioning systems* [14, 13, 7]. However, all of these approaches are confined to structural models inasmuch as the behavior is hard-coded into the respective system.

In this paper, we investigate *modeling in the large* with and beyond Fujaba. As to be demonstrated, the model currently being developed in the MOD2-SCM project is fairly large. Furthermore, the success of the project heavily depends on a carefully designed *model architecture* with loosely coupled architectural units [2]: The product line should support a set of variation points which may be combined in an orthogonal way as far as possible. To this end, the coupling between architectural units has to be reduced to a minimum.

2. MODELING IN THE LARGE

In object-oriented modeling, modeling in the large is an area which has not attracted sufficient attention so far. In the following, we will first discuss support for modeling in the large as far as it is provided in the current version of Fujaba. Subsequently, we will show how external tool support may be used to complement the functionality of Fujaba by creating package diagrams from generated Java code. Finally, we will discuss package diagrams in UML 2.0.

2.1 Support in Fujaba 5.1

On a coarse-grained level, Fujaba models are organized into *projects*. A project stores a model in a single file. A model stored as a project may be self-contained, or it may reference models stored in external projects. These references imply *dependencies* between projects. In this way, both physical decomposition and model reuse are supported.

Within a project, model elements are created in diagrams. *Class diagrams* serve as the primary means for structuring. When a class is created, it may be assigned to a *package* in the dialog window for class editing. Fujaba maintains a tree view of diagrams and model elements, but packages can be neither defined nor displayed in the tree view. Furthermore, Fujaba does not support *package diagrams*.

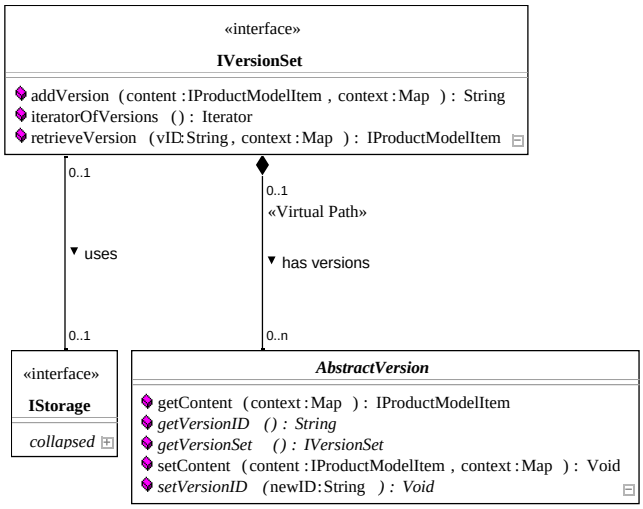

Figure 1: Excerpt of a Fujaba class diagram

In a class diagram, the owning package of a class is not displayed. Furthermore, the class diagram itself is not owned by a package. As a consequence, there is no distinction between references and declarations of classes, and it is not obvious from which packages the classes originate.

To some extent, conventions may be used to structure the model. Such conventions may state e.g. that exactly one class diagram is introduced for each package, the package name is used to identify the class diagram, and all external classes are displayed with collapsed attribute and method sections. The latter is demonstrated in Figure 1, which shows an excerpt of the class diagram for maintaining version sets (package core.versions, see later). Interface IStorage was imported from another package.

2.2 Reverse Engineering with eUML

To complement the modeling facilities of Fujaba, we used the *eUML* plugin of Eclipse to generate a package diagram from the Java code created by the Fujaba compiler. The package diagram for the model in its current state of evolution is displayed in Figure 2. eUML distinguishes between different kinds of dependencies, resulting e.g. from imports, specializations, instantiations, and method calls in the Java code. The eUML user may configure the kinds of dependencies displayed in the diagram.

While the eUML package diagram is helpful, it still suffers from several limitations. First, since it is reverse engineered from the generated Java code, it cannot be used for designing the model architecture up front. Second, the resulting graph is rather dense since it also includes "secondary" dependencies (e.g., to call a method, the class of a parameter may have to be imported, as well). Third, the diagram displays implementation-level dependencies, i.e., dependencies in the generated code, which may differ from conceptual dependencies.

Let us give an example for the latter: In Figure 1, a version set (interface IVersionSet of package core.versions) makes use of a storage (interface IStorage of package core.storage). This association is introduced in the package core.versions: The storage stores a set of versions using deltas, but it is independent of the way how the version set is organized on a conceptual level. On a conceptual level, core.versions depends on core.storage but not vice versa. However, for a bidirectional association Fujaba generates methods for navigation at both ends, introducing cyclic dependencies in the generated code[1].

2.3 UML 2.0 Package Diagrams

Let us briefly recall the concepts which UML 2.0 offers for structuring large models [8, 9]: A model may be structured into hierarchically organized *packages*. Each model element is owned by exactly one package. *Private elements* are not accessible from other packages, while *public elements* are visible. Each package defines a *namespace* in which the names of declared model elements have to be unique. Public model elements from other packages may always be referenced through their full qualified names. A model element from an enclosing package may be referenced without qualification unless it is hidden by an inner declaration.

Apart from nesting, UML 2.0 introduces the following relationships between packages: *Imports* merely serve to extend the set of those elements which may be referenced without qualification. UML 2.0 distinguishes between *public* and *private* imports, which are denoted by the stereotypes <<import>> and <<access>>, respectively. A private import makes the imported elements visible only in the importing package, while a public import simultaneously adds those elements to its exported namespace. Public imports are transitive; this property does not hold for private imports. Apart from nesting and imports, UML 2.0 offers *package merges*, which will not be discussed in this paper.

Figure 3 shows a UML package diagram for the current MOD2-SCM model. Please note the differences to the eUML diagram of Figure 2: First, the UML diagram visualizes nesting of packages. Second, only conceptual relationships are shown (e.g., core.versions imports core.storage but not vice versa). Finally, the number of relationships is significantly reduced due to the transitivity of public imports.

There are fundamental differences between imports in UML 2.0 and imports in *modular programming languages* such as Modula-2 and Ada. In these languages, each — separately compiled — program unit (called module in Modula-2 and package in Ada) may reference only its own local declarations unless the namespace is extended by an import. Depending on the kind of import, imported elements may be referenced with or without qualification. Furthermore, an imported element may only be *used* but not modified. In contrast, the UML 2.0 standard states ([8], p. 143): "An element import ... works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported." For example, an imported class may be inserted into a class diagram of the importing package and extended with attributes, methods, and associations; these extensions apply to the imported package where the imported class is declared.

Altogether, in UML 2.0 the rules for referencing and modifying elements are liberal and make it convenient to access external elements. Unfortunately, these rules may threaten the *modularity* of a UML model. In the case of an undisciplined modeling process, it is fairly easy to create an archi-

[1]While we could have used a unidirectional association in this example, it would be far too restrictive to enforce unidirectional cross-package associations in general.

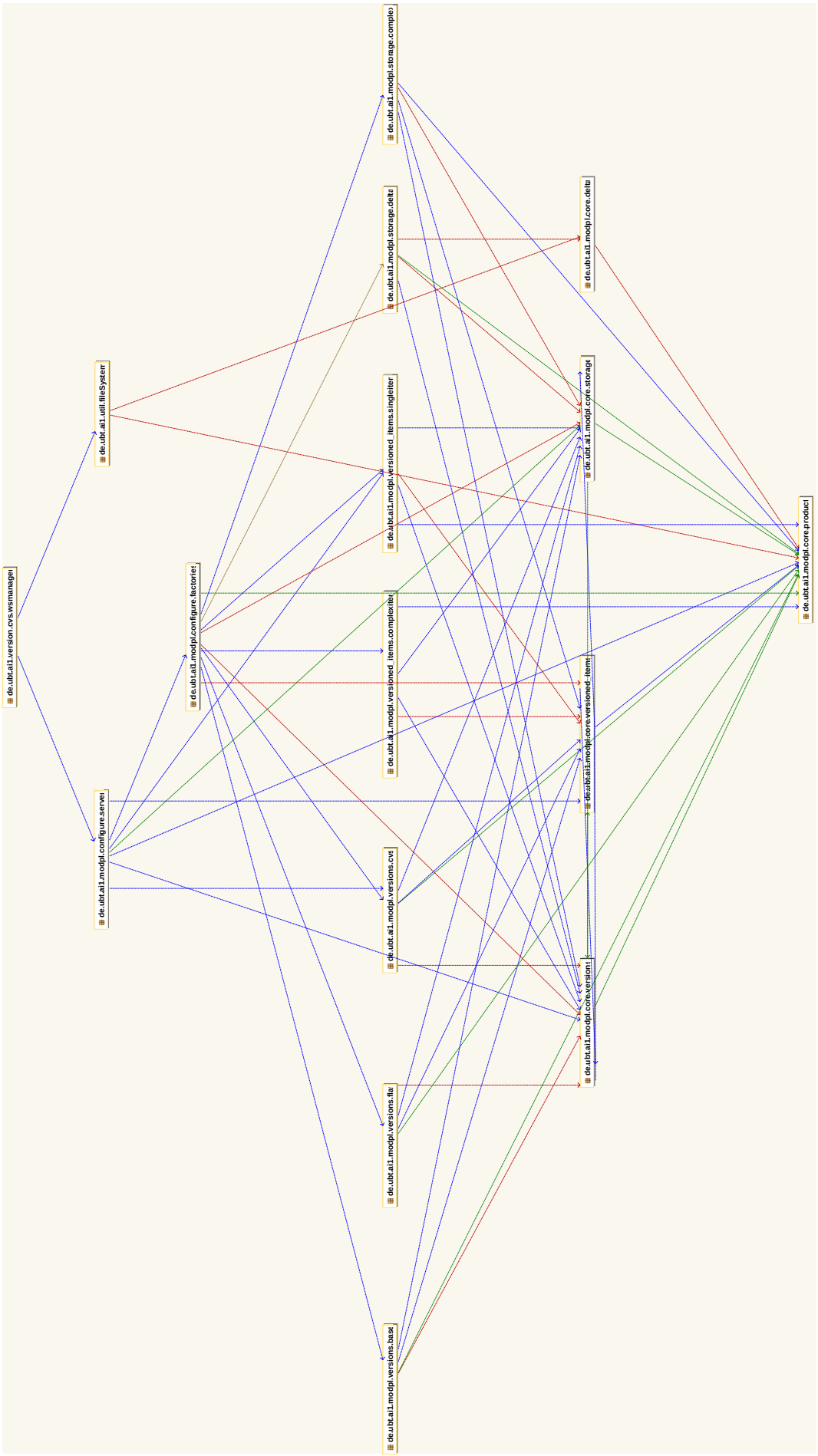

Figure 2: eUML package diagram extracted from generated Java code

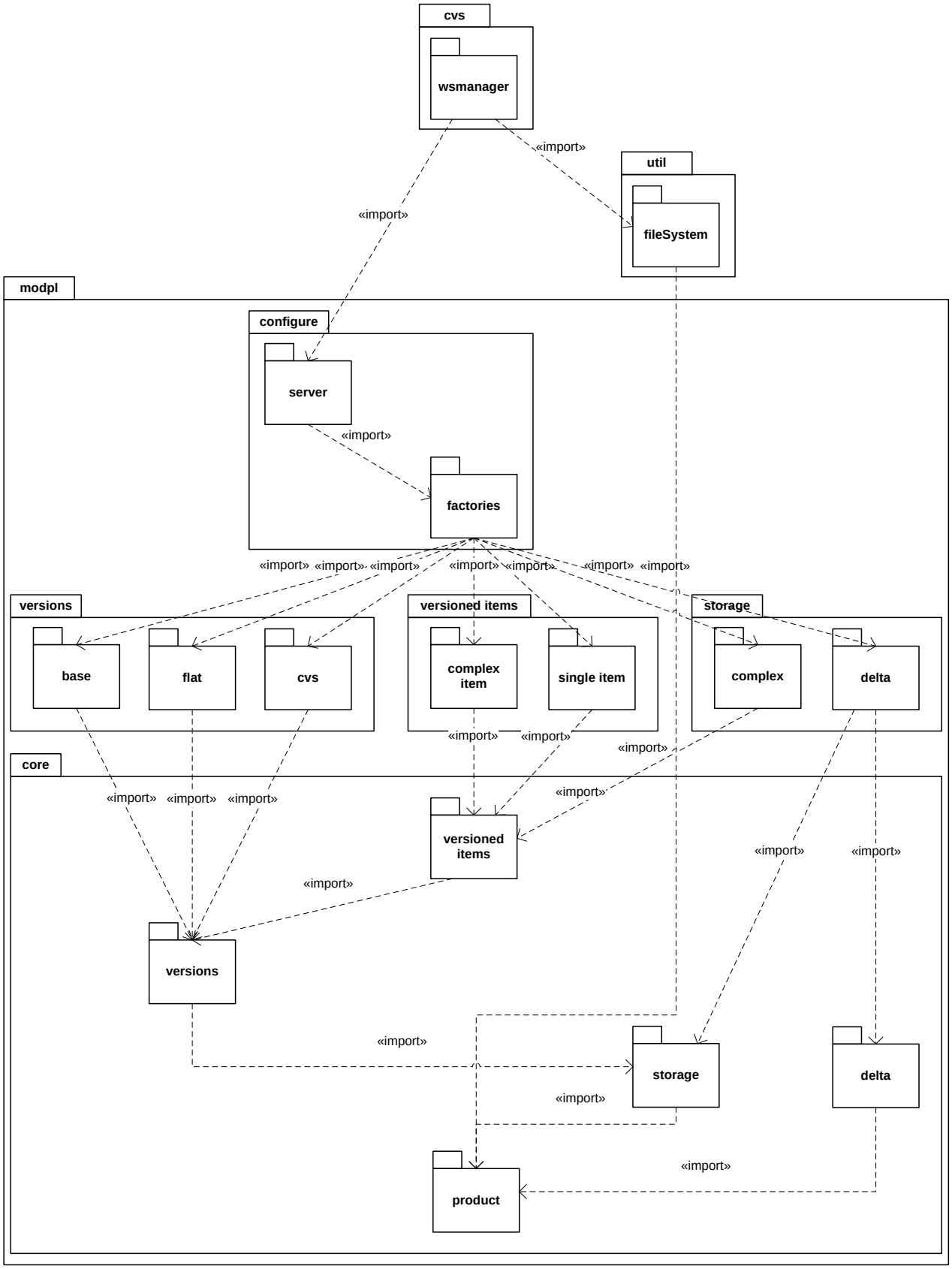

Figure 3: UML package diagram using nesting and public imports

tecture with tightly coupled packages — which would violate the goals we pursue in the MOD2-SCM project.

In particular, there is no guarantee that a package diagram such as shown in Figure 3 shows the actual dependencies between packages in the architecture. First, it is possible to reference external elements without imports by using fully qualified names; the implied dependencies would not be displayed in a package diagram showing only import relationships. Second, public imports are transitive. Thus, when importing some package, all packages of the transitive closure of public imports are visible, as well. A package diagram with public imports does not tell which of these packages are actually referenced.

For example, in Figure 3 the package configure.server imports configure.factories and thus may reference all packages below. If the actual dependencies were so comprehensive, we would have to be concerned about the modularity of the system. Figure 2 shows that only a few dependencies emanate from the server package. Still, there is one dependency on the package versions.cvs which appears to be suspicious: Why should a configurable server depend on a specific version model such as the CVS model?

This example demonstrates that the package diagram with public imports appears to be elegant, but is too imprecise: There is no way around inspecting all actual dependencies emanating from a package. To support such analyses, private imports would be more useful: As Java imports, private imports are not transitive, forcing the client to explicitly import all packages on which it depends. Of course, private imports imply a much denser diagram similar to the one shown in Figure 2.

3. CONCLUSION

In this paper, we reported on experiences with modeling in the large in Fujaba, referring to a recently launched project for developing a model-driven product line for SCM systems. Our experiences indicate that reverse engineering of the model architecture with an external tool is not sufficient and thus improved support for modeling in the large in Fujaba itself is urgently needed. We also discussed UML 2.0 package diagrams as a notation for model architectures, focusing on package imports (the fairly complex concept of package merge goes beyond the scope of this paper, see e.g. [6]). Please note that package diagrams are available in MOFLON [1], which has been built on top of Fujaba. However, MOFLON currently supports only public imports, while our experiences demonstrate that private imports are needed, as well.

To conclude, we give a few suggestions for extending Fujaba with support for modeling in the large: First, the tree view should be revised such that it allows to define a package hierarchy. Second, not only model elements such as classes and associations, but also class and story diagrams should be assigned uniquely to one package. Third, a graphical editor for package diagrams should be offered (supporting nesting of packages as well as public and private imports). Fourth, model elements may be referenced only where they are visible. In addition, we recommend to implement some restrictions which deviate from the UML 2.0 standard: First, qualified references should be disallowed to avoid hidden dependencies. Second, it should be prohibited to modify imported elements.

4. REFERENCES

[1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *LNCS*, pages 361–375, Heidelberg, 2006. Springer.

[2] T. Buchmann, A. Dotor, and B. Westfechtel. MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management*, Toulouse, France, Oct. 2008.

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Boston, Massachusetts, 2005.

[4] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly & Associates, Sebastopol, California, 2004.

[5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[6] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software and Systems Modeling*, Dec. 2007. Online First.

[7] J. Kovše. *Model-Driven Development of Versioning Systems*. PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany, Aug. 2005.

[8] Object Management Group, Needham, Massachusetts. *OMG Unified Modeling Language (OMG UML), Infrastructure, V 2.1.2*, formal/2007-11-04 edition, Nov. 2007.

[9] Object Management Group, Needham, Massachusetts. *OMG Unified Modeling Language (OMG UML), Superstructure, V 2.1.2*, formal/2007-11-02 edition, Nov. 2007.

[10] W. F. Tichy. RCS – A system for version control. *Software: Practice and Experience*, 15(7):637–654, July 1985.

[11] J. Vesperman. *Essential CVS*. O'Reilly & Associates, Sebastopol, California, 2006.

[12] B. A. White. *Software Configuration Management Strategies and Rational ClearCase*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 2003.

[13] E. J. Whitehead, G. Ge, and K. Pan. Automatic generation of hypertext system repositories: a model driven approach. In *15th ACM Conference on Hypertext and Hypermedia*, pages 205–214, Santa Cruz, CA, Aug. 2004. ACM Press.

[14] E. J. Whitehead and D. Gordon. Uniform comparison of configuration management data models. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management: ICSE Workshops SCM 2001 and SCM 2003*, volume 2649 of *LNCS*, pages 70–85. Springer, 2003.

[15] A. Zündorf. Rigorous object oriented software development. Technical report, University of Paderborn, Germany, 2001.