Author(s): Koivulahti-Ojala, Mervi; Käkölä, Timo

Title: Framework for Evaluating the Version Management Capabilities of a Class of UML Modeling Tools from the Viewpoint of Multi-site, Multi-partner Product Line Organizations

Year: 2009

Version:

Please cite the original version:

Koivulahti-Ojala, M.; Käkölä, T., "Framework for Evaluating the Version Management Capabilities of a Class of UML Modeling Tools from the Viewpoint of Multi-Site, Multi-Partner Product Line Organizations," System Sciences (HICSS), 2010 43rd Hawaii International Conference on , vol., no., pp.1-10, 5-8 Jan. 2010. doi: 10.1109/HICSS.2010.213

Framework for Evaluating the Version Management Capabilities of a Class of UML Modeling Tools from the Viewpoint of Multi-site, Multi-partner Product Line Organizations

Mervi Koivulahti-Ojala & Timo Käkölä
University of Jyväskylä
40014 University of Jyväskylä, Finland
{meelheko, timokk}@jyu.fi

Abstract: UML models are widely used in software product line engineering for activities such as modeling the software product line reference architecture, detailed design, and automation of software code generation and testing. But in high-tech companies, modeling activities are typically distributed across multiple sites and involve multiple partners in different countries, thus complicating model management. Today's UML modeling tools support sophisticated version management for managing parallel and distributed modeling. However, the literature does not provide a comprehensive set of industrial-level criteria to evaluate the version management capabilities of UML tools. This article's contribution is a framework for evaluating the version management features of UML modeling tools for multi-site, multi-partner software product line organizations.

Keywords: Global software development, modeling tool, software product line organization, tool evaluation, UML modeling, version management

1. INTRODUCTION

To succeed in the global markets of software-intensive products, high-tech companies need to shorten the cycle time of new product development while improving product quality and service delivery along with maintenance or reduction of the total resources required [6;20;25]. This concern can be dealt through internal or external strategies. Internal strategies include global software development, where development resources are distributed globally to reap cost benefits, leverage specialized competencies, and address specific needs of geographically-defined markets [7;13;31], and software product line engineering and management, that is, the strategic acquisition, creation, and reuse of software assets [19;24;30]. External strategies include acquiring commercial off-the-shelf components and outsourcing software development, maintenance, and related services to best-in-class service providers [8;18].

This paper focuses on the software product line engineering strategy in the context of global software development. Software product line engineering is an industrially validated methodology for developing software products and software-intensive systems and services faster, at lower costs, and with better quality and higher end-user satisfaction. It differs from single system development in two primary ways [30]:

1. It needs two distinct development processes: domain engineering and application engineering. Domain engineering defines the commonality and variability of the software product line, thus establishing the common software platform for developing high-quality applications rapidly within the line. Application engineering derives specific applications by strategically reusing the platform and by exploiting the variability built into the platform.
2. It needs to explicitly define and manage variability. During domain engineering, variability is introduced in all software product line assets such as domain requirements, architectural models, components, and test cases. It is exploited during application engineering to derive applications mass-customized to the needs of different customers and markets.

Software product line engineering involves higher levels of abstraction than single-system development methods because the platforms require substantial investments, have long life cycles, and have to be generally applicable to a wide range of products. Without appropriate abstractions, such platforms cannot be built and variability cannot be managed effectively. Industrially validated modeling approaches and commercially available modeling tools are critically important to deal with the abstractions. In addition to traditional system modeling, variability modeling is required in product line engineering to explicitly document how the applications within the product line can vary.

To model the variability of a product line, two approaches have been proposed in the literature. The first, traditional approach has been to integrate variability modeling in the systems modeling language such as Unified Modeling Language™ (UML) [12;27] by appropriately extending the metamodel of the language [4]. The second approach, orthogonal variability modeling, distinguishes between a variability model and a system model [30]. Orthogonal variability models are easier to apply in practice and scale better than integrated variability models. They usually describe the variability using a graphical notation. One reason, orthogonal variability modeling is not yet extensively used in the industry, is that there are no commercially available modeling tools to support it. Therefore, this paper will focus on the traditional integrated modeling approach.

System models can be applied, for example, to model static and dynamic aspects of the software product line reference architecture, to conduct detailed domain and

978-0-7695-3869-3/10 $26.00

application design, and to automate software code generation and testing. UML has become the most widely accepted software system modeling language [4]. It can also be used to model embedded, business, and real-time systems.

UML modeling tools supporting sophisticated version management are critical for managing parallel and geographically distributed modeling activities. However, the extant literature does not provide a comprehensive set of industrial-level criteria to evaluate the version management capabilities of UML tools. If modeling tools fail to support version management in multi-site, multi-partner development environments, the modeling process may be ineffective and modeling tools may not be used optimally. Ineffective tool deployment is expensive since there will be substantial costs without realizing the potential benefits.

The main contribution of this paper is a framework consisting of a set of industrial-level criteria for evaluating UML modeling tools. The framework can be used in practice to determine whether particular UML tool instances support collaborative modeling through version management in multi-site and multi-partner product line organizations. The framework has been created based on a literature review and empirical experiences of the first author during a tool evaluation project. The goals of the project for a large global product line organization, which leveraged multi-site, multi-partner practices, were to identify and evaluate commercial UML tools and to select one for global deployment.

The paper is organized as follows. In Section 2, the basic concepts related to UML modeling and modeling tools are introduced. In Section 3, the existing research related to version management capabilities afforded by UML modeling tools is evaluated. In Section 4, the research method and the case organization are described. In Section 5, the role of version management in multi-site, multi-partner product line organizations is discussed and the framework consisting of a set of evaluation criteria is proposed. In Section 6, two commercial UML modeling tools are evaluated using the framework. In Section 7, the validity and usefulness of the framework are evaluated. Section 8 concludes the paper.

2. FUNDAMENTALS OF MODELS AND UML MODELING TOOLS

In this section, the notion of a model is explained and a framework (Table 1) is created to depict how models can be applied for different types of communication in the context of product line engineering. The role of modeling tools in supporting the shared creation and maintenance of models is then discussed.

2.1 A framework for analyzing product line models

The interpretation of models involves the assignment of meanings to the symbols and truth-values to the sentences of the models [33, p.74]. Models can be used for sharing information between humans, between machines, and between humans and machines.

Counterparts in communication	*Example in product line modeling*	*Example reference related to product line modeling*
Human to human	Modeling requirements Modeling the software product line reference architecture	Product line variability modeling with UML 2.0 [4] Software Product Line Engineering with the UML: Deriving Products [35]
Human to machine	Test automation	Product Line Use Cases: Scenario-Based Specification and Testing of Requirements [5]
Machine to human	Reverse engineering	Feature-oriented Re-engineering of Legacy Systems into Product Line Assets – a Case Study [15]
Machine to machine	Model transformations Code generation	Code Generation to Support Static and Dynamic Composition of Software Product Lines [34]

Table 1. A framework for analyzing product line models as means of communication and information sharing.

Human to human communication

Modeling language independent and dependent modeling approaches have been proposed to support human communication. Kruchten [17] argues that software architectures should be depicted from five modeling language and tool independent viewpoints: logical, process, physical, development, and use case. The depictions allow for the separation of the concerns of the various architectural stakeholders (e.g., end-users, developers, systems engineers, and project managers). In the area of software product lines, the modeling languages need to enable the modeling of commonalities and variabilities. For this purpose, Bayer et al. [4] present a consolidated variability meta-model with a unified terminology and representation that enables variability specification during domain engineering and variability resolution during application engineering. The model helps stakeholders to collaborate throughout the life cycles of software product lines and vendors to develop interoperable commercial and open-source modeling tools.

Human to machine communication

Models can be used (primarily during requirements engineering) to codify human knowledge and organizational rules and resources into forms that enable computerized actions. The Object Constraint Language (OCL), being part of the UML standard, is useful in product line engineering for (1) defining rules to which domain model elements must conform, so application models can be derived from the domain models, and for (2) validating the application models that reuse and possibly modify the domain models. Models are also crucial for

validating the codified knowledge. For example, Leppänen [22] has used formal methods for testing models. In the area of product line engineering, models have been used to test application requirements derived from domain requirements [5]. The models have also been used to derive application test cases from reusable domain test cases, which have been created to verify and validate domain requirements [32].

Machine to human communication

Humans routinely use computer-based information systems for decision-making and analysis. For example, reverse engineering tools are used to automatically create architectural and other models based on code. This is especially useful in the context of open source software that is seldom accompanied by detailed design models [2]. The capabilities of reverse engineering tools to generate product line models with explicitly defined variability from application code are limited partly because much of the variability has typically been resolved by the time the code has been created. For example, if an application has been derived that contains no optional features afforded by the product line, the code related to the optional features may be entirely missing from the application code, making it impossible to determine based on the code which optional features may have been available in the product line. To our knowledge there are no reverse engineering tools, which could interpret the code and transform the implemented variable and configurable elements of software product lines into variability models.

Machine to machine communication

Models can be automatically transformed into other models or to code. Model Driven Architecture (MDA) is a framework based on the UML and other industry standards that promote the creation of machine-readable, abstract models [16]. The models are developed independently of the technology platforms; stored in and shared through standardized repositories; and automatically transformed into database schemas, software code, and other assets for various platforms.

Several modeling tools support platform-independent modeling, code generation, XML, and/or database schema generation features. For example, when a product line consists of similar products running on different operating systems, domain engineering can leverage platform-independent modeling to design and implement the common parts of the product line for the operating systems. The platform-independent designs can then be transformed into platform-specific ones to create the operating system-specific products during application engineering [9]. Code generation is also common during application engineering [34].

2.2 UML modeling tools

UML modeling tools offer graphical editors to help architects and developers model requirements, architectures, data structures, dynamic behaviors, and other characteristics of systems. Most tools also support the UML 2 profile mechanism, enabling the creation and use of Domain Specific Languages (DSLs), for example, for variability modeling. Some UML tools can generate software from UML models and UML models from the software. Some modeling tools have a built-in knowledge of UML rules, so they can automatically validate the correctness of UML models. Table 2 presents typical high-level features for the class of UML modeling tools.

UML tools often support distributed software development within and across teams. The traditional approach has been to make model repositories available to the teams through centralized servers. When centralized version management is deployed together with centralized servers, specific locking mechanisms are typically enforced to enable multiple users to simultaneously work with a model and to completely prevent conflicts that otherwise would result from parallel model updates. When more freedom with concurrent model editing is desired, the merging mechanisms of centralized version management systems enable the free concurrent editing of a model, inform developers of possible conflicts when they check their changes into the centralized repository, and merge changes and resolve conflicts automatically or based on developer input.

3. LITERATURE REVIEW

Oldevik et al. [28] propose a set of evaluation criteria for product line modeling tools but the set does not address version management. To our knowledge, no other papers present comprehensive evaluation criteria for product line modeling tools. However, there are a few papers related to the requirements for UML modeling tools [11;23]. This section reviews those papers from the version management point of view to find out how features supporting collaborative work are described.

3.1 General-purpose requirements for the class of UML modeling tools

Funes et al. [11] present a generic set of requirements based on authors' experiences. They provide no references to case studies in particular organizations. They group requirements into the following categories: Features (that are not related to modeling), Modeling support, Customization, Installation and performance, and Tool support. Only three of the requirements relate to the features supporting modeling in collaborative environments (Table 3): (1) Multiple User Support, Access control/sharing, (2) Multiple User Support, Concurrency control, and (3) Versioning. Funes et al. [11] do not explain the requirements in more detail.

Lester & Wilkie [23] propose 15 criteria for UML tool evaluation partially based on the feature lists of existing products. Two of them relate to version management but they are only described as headings and not explained in more detail. The criteria are based on experiences in a software company with only two sites. Therefore, other relevant evaluation criteria might be needed in multi-site and multi-partner organizations.

Feature	*Purpose of the feature is to help*
Modeling & Diagramming	Create, remove, and edit model elements; view the models from different perspectives, and create, remove and edit diagrams.
Hierarchy Management	Create, update, and delete hierarchies (i.e., packages) in which model elements are assigned.
Collaboration and Version management	Multiple concurrent users to manage different versions of assets and to resolve conflicts; integrate the UML tool to version control and/or change management systems as necessary.
Publishing	Compose and publish views of the selected models or model elements; provide data in different formats (XMI, HTML/ODT/PNG/JPG); create reports and documents based on the selected model or model elements.
Tracebility	Create, remove, update, and trace relationships between models or model elements.
Simulation and Validation	Simulate dynamic behaviours of models or interface or integrate the tool to simulation tools; validate UML model correctness and completeness.
Model and Code Synchronization	Generate code based on models; create models based on code (reverse engineering); integrate UML tools to source code systems or Eclipse; integrate UML tools with MDA tools such as oAW, AndroMDA, and BlueAge.
User Management	Manage access and connectivity to the organization's directory services (LDAP, AD).

Table 2. Common features of UML tools.

Funes et al. [11]	Requirements	Multiple User Support 1. Access control/sharing 2. Concurrency control Versioning
Lester et al. [23]	Evaluation criteria	Repository/Version control support Componentization
Oldevik et al. [28]	Evaluation criteria	(None)

Table 3. Version management related requirements for the class of UML tools.

3.2 Summary

The requirements and evaluation criteria for UML modeling tools presented in the papers analyzed in this section do not describe features supporting collaborative modeling in such detail that UML tool evaluations could be completed. In all the papers authors draw upon their own experiences or the feature sets of existing products. Thus, the version management related requirements seem to be based more on the analysis of existing products than on the needs of the users of UML tools. Therefore, this paper will analyze in more depth version management related to product line modeling in multi-site, multi-partner organizations.

4. DESCRIPTION OF THE CASE ORGANIZATION AND THE RESEARCH METHOD

An evaluation framework has been created based on the experiences in the global case organization and the literature review. The case organization is a large multi-site and multi-partner high-tech company using the software product line strategy to successfully operate in highly diverse global markets. The UML modeling tool evaluation project was initiated in 2006 by a department responsible for the development and delivery of global information management solutions and services for R&D units within the case organization. Requirements were gathered during the winter 2007-2008 to understand the features necessary for applying UML modeling tools to model system architectures together with collaborators and partners.

The project was managed according to the internal corporate guidelines for tool evaluation projects. The project team consisted of a project manager, seven architects from major user organizations, and two IT specialists/architects. The first author of this paper was responsible for requirements engineering. Each user organization representative was interviewed by phone during the first phase. Other requirements sources included the industrial best practices reported in journals and in the Internet, modeling tool experts, and IT architects. Requirements were described in writing based on the interviews, reviewed, and prioritized by the project team.

One of the highest priority requirements related to version management was that the UML tools must support sophisticated locking mechanisms. The mechanisms must (1) enable developers to define various parts of a model that they can update independently and (2) prevent conflicts from parallel model updates.

In the first phase, 15 modeling tools from 13 vendors were evaluated. Based on the requirements, three commercial products were selected for in depth evaluations, including detailed vendor liability, financial, and tool architecture evaluations. Details of vendor liability, tool architecture, and financial evaluations are not provided in this paper because the case organization and tool vendors have agreed that the evaluations are confidential. Final evaluations of the three tools were based on feedback from the project team, performance and other tests of the three tool installations in the case organization, reviews with vendors, and the available documentation.

Interestingly, during the evaluation and piloting process it was found out that the modeling tools significantly differed with respect to their version management capabilities. Thus, the case organization became interested in creating a set of more detailed evaluation criteria to enable the detailed analysis of version management capabilities. The criteria presented in Section 5.2 reflect the high-level requirements determined du-

ring the evaluation project and can be used for evaluations of version management capabilities. The detailed questions for each evaluation criteria have been created based on the literature review. The criteria related to the availability of historical traceability information were added to the framework solely based on the experiences in the case organization because historical information has been crucial to ensure proper version management in the organization.

5. TOOL EVALUATION FRAMEWORK

5.1 Assets to be modeled in product line organizations

In a product line organization, the assets the organization creates, maintains, and manages to satisfy market needs constitute systems composed of software and hardware. The hardware and software assets may be managed as products, product lines, and platforms serving several other products or product lines. Other companies, organizations, or individuals may manage and even own the software and hardware components.

Figure 1 depicts a platform shared across two product lines. Both product lines consist of three product variants that are used by markets consisting of individual consumers and/or organizations. Platforms provide common (mandatory) and variable (alternative or optional) features shared across products or product lines. Complex organizational networks can be responsible for owning and sharing the components used within platforms and product lines. In Figure 1, the platform contains three components, which are not managed by the organization responsible for the platform, and product line 1 contains two components, which are not owned or managed by the product line 1 organization.

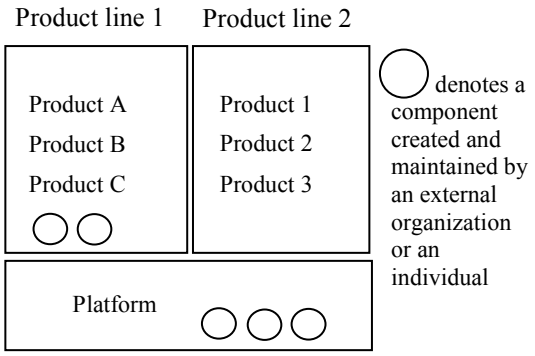

Figure 1. Software and hardware asset design and maintenance responsibility.

The use of models in this network of collaborators and partners would require seamless interactions to share the models. For example, the company responsible for product A may share the understanding of the architecture in the form of models that the partner could further use when planning and implementing the models related to the design of a particular component used in the product.

Different strategies may be implemented to enable modeling of the assets. In this paper we consider a strategy in which one (commercial) modeling tool is used across the product line organization. This strategy minimizes the need for data transfer across tools but may require extensive effort for training and dealing with resistance to organizational change.

5.2 Evaluation characteristics

In this section, the evaluation characteristics are described to define a set of desired version management properties for the class of UML tools (Table 4). The characteristics have been derived from documented version management (e.g., [1]) and product line modeling (Section 5.1) practices and from the requirements, the project team identified in the case organization.

We adopt the following terminology defined by Object Management Group for UML2 [27]:

- A *model* captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose.
- A *package* is used to group elements, and provides a namespace for the grouped elements.
- *Diagrams* are graphical representations of parts of the UML model. UML diagrams contain graphical elements that represent elements in the UML model.
- An *element* is a constituent of a model.
- A *property* is a structural feature.

The justification for each characteristic is indicated by questions to be answered during evaluations. The output domain of permitted answers is also defined for each question. Some questions have *Yes* or *No* as the output domain while others have a range of possible answers.

Two sets of characteristics are defined: one for version management support from the viewpoint of functionality (i.e., which features users can use) and one from the viewpoint of client and server technology to find out whether the technology supporting version management is feasible for large multi-site, multi-partner organizations. Specific questions have been added for each set. Organizations planning to introduce UML modeling tools should carefully consider the questions and add or remove questions according to their specific needs. However, Table 4 serves as a baseline for evaluation. Section 5.2.1 discusses each question in more detail.

Fundamental version management concepts in distributed parallel development of software are check-in/check-out, branching, and merge [1]. System models need to be version controlled in the same manner as software code. For example, when UML is adopted to model the deployment view of software, each model element may have a corresponding element in the software code (see [17] and Section 2.1). As discussed in Section 5.1, different parties may manage the software assets up to the component level. It should thus be possible to manage models up to the element level that corresponds with the component in software. For example, if there is a new version of the component to be branched, the model should reflect this change, so it should be possible to make a branch for the corresponding element in a model. Another example

involves the modeling of a common view of product line architecture (see [4] and Section 2.1). When a team is working on a common view of the architecture through a model and a team member checks out the model, others cannot continue the work until the same person has completed the check-in. During the interviews in the case organization, interviewees reported experiences of model level check-ins and check-outs, which were seen as problematic. The case organization thus determined that it should be possible to check-in and check-out at the element level. Therefore, each version management feature should support operations at the element level.

Version management features	*Evaluation question*	*Evaluation answer*
Check-in/Check-out	Is there support for Model, Package, Diagram and Element level check-in and check-out for multiple users? {Yes/No}	Model, Package, Diagram, and Element level check-in and check-out enable teams to work effectively with the models. Mandatory.
History	Is the Element level history available (who has made what changes)?{Yes/No}	Element level history enables tracing of all the changes. Optional.
Model comparison	Is it possible to compare models at the Element level? {Yes/No}	Element level comparison is a prerequisite for merging. Mandatory.
Merging	Is there support for Model, Package, Diagram, and Element level three-way merge? {Yes/No}	Model, Package, Diagram and Element level three-way merging enables teams to merge models effectively and reliably. Mandatory.
Branching	Is there support for Model, Package, Diagram and Element level branching? {Yes/No}	Model, Package, Diagram and Element level branching enables teams to work effectively with models. Mandatory.
Technologies for version management	*Evaluation question*	*Evaluation answer*
Server-side technology	Are three-tier technologies supported? {Yes/No}	Three-tier technologies enable scalable and reliable solutions.
Client-side technology	Are client installations required? {Yes/No} Are there maintenance needs for the clients? {Yes/No}	Client installations and project-specific needs for tool configurations increase maintenance costs and support needs.

Table 4. Evaluating the version management features and technologies of the class of UML tools.

5.2.1 UML tool features for supporting version management

Check-in/Check-out

Appleton [1] states that most widely used version control tools employ the checkout-edit-checkin model to manage the evolution of version-controlled files in a repository or codebase. Element level check-in and check-out enables completing the necessary tasks effectively in product line modeling. Diagrams, packages, and models could also be useful elements to be checked in and out.

History

Version management requires thorough traceability so users know who has done which changes to the model and can rollback changes if needed. Knowing the history of data helps determine the extents to which the data is trustworthy and up-to-date. Knowing the previous editors also gives points of contact for inquiries. To enable tracebility, log information should be automatically collected and appropriate features should be available to see and analyze the log information. It should be possible to trace back to the element level because users may need to know, for example, who has made changes to a particular component. Diagrams, packages, or models could also be useful elements to trace. However, this feature is optional because users can work without comprehensive tracebility at least as long as their routines and/or tools do not break down. When coordination breakdowns disrupt the routines, it is typically time consuming and expensive to find out and fix the reasons for the breakdowns, if the traceability information is missing [20,21].

Model comparison

Model comparison enables identifying the changes between two models. It can take place in different levels. For example, two models can be compared and their differences can be reported on package, diagram, element, and property levels. Most sophisticated comparison functionalites enable comparisons up to the property level, so users can see the differences between different UML elements' properties. Because efficient merging requires comparisons, this feature is mandatory.

Merging

Merging is the means by which one development line synchronizes its contents with another development line [1]. Merging can be implemented as a 2-way or a 3-way merge. In a 2-way merge, two software artefacts are merged without information about the possible common ancestor. In a 3-way merge, the information about the common ancestor is used. The 3-way merge is more reliable than the 2-way merge because it can detect conflicts better and identify actual changes more precisely. In product line organizations, it should be possible to merge at the element level. For example, users may need to merge models of two branches reflecting changes made to the code. To minimize manual work, diagram merging should also be possible.

Branching

Branching in its most basic form allows development to take place along more than one path for a particular file or directory [1]. Branching can be applied to five different software development situations [1]. Branching of (1) the system's *physical* configuration -

branches are created for files, components, and subsystems, (2) the system's *functional* configuration - branches are created for features, logical changes (bug fixes and enhancements), and other significant units of deliverable functionality (e.g., patches, releases, and products), (3) the system's *operating environment* - branches are created for various aspects of the build and runtime platforms (e.g., compilers, windowing systems, libraries, hardware, and operating systems) and/or for the entire platform, (4) the team's work efforts - *Organizational* branches are created for activities/tasks, subprojects, roles, and groups, and (5) the team's work behaviors - Procedural branches are created to support various policies, processes, and states.

For each category, analogical needs for the branching of product line related models can be identified. (1) Physical branching of models when modeling software for different subsystems as a basis for code generation, (2) functional branching for different products, (3) environmental branching for hardware, software, and related platforms, (4) organizational branching for different projects, and (5) procedural branching to support different product line modeling processes. We see that model branching is analogical to the branching of software and thus it should be possible to branch at least at the package level, as packages provide the mechanism to group model elements. However, optimal support for product line modeling requires branching at the element level. For example, if there is a new version of a component to be branched, it should be possible to make a branch for the corresponding element in the associated model.

5.2.2 Technologies for supporting version management

Version management can be supported by two- or three-tier technologies. Three-tier technologies are more scalable and reliable than two-tier technologies. If client installations are needed, the magnitude of maintenance and support costs incurred to keep the clients updated needs to be considered. If the clients also need to be configured for each modeling project separately, the maintenance and support costs will increase even more.

5.2.3 Summary

This section described a framework consisting of seven criteria to support the evaluation of version management capabilities of UML modeling tools. The criteria were derived from documented version management practices (e.g., [1]) and characteristics needed in product line modeling (Section 5.1). The framework is composed of two sets of characteristics: one from the functional perspective (i.e., which features users are can use?) and one from the technical perspective (i.e., are technologies supporting version management feasible for large multi-site organizations?).

6. USING THE FRAMEWORK TO EVALUATE TWO COMMERCIAL UML MODELING TOOLS

In this section the commercial UML modeling tools Enterprise Architect (http://www.sparxsystems.com/) and Magicdraw (http://www.magicdraw.com/) are evaluated (Table 5) using the framework described in Section 5.2. Commercial UML modeling tools have been selected for the evaluation because global high-tech organizations typically benefit from purchasing commercial modeling tools [26]. Open-source UML tools are not yet as mature as their commercial counterparts are but they have reached a sufficient maturity level to benefit small and medium sized businesses [26]. We chose the two tools for evaluation because they (1) are available for Macintosh, Linux and Windows operating systems, (2) are not too expensive for a large company to deploy for even thousands of users, (3) support SysML, and (4) provide version management features. These four criteria are adequate to simulate a situation where a large company is looking for a UML modeling tool for organization-wide use by both systems and software architects, designers, and other stakeholders.

The term “project” used in both Enterprise Architect and MagicDraw equals to the term “model” adopted in this paper; one project may contain any number of packages, elements and diagrams.

6.1 Enterprise Architect

Enterpise Architect can be used in conjunction with several version management tools such as Subversion, CVS, ClearCase, Visual Source Safe, Accurev, and Perforce. Each package in a model can be version-managed separately as a XMI file, checked-in, modified and checked-out. In addition, *User Security* feature provides means for individual users or user groups to lock, modify, and unlock package(s), diagram(s) or element(s). It is also possible to use several version-managed packages at the same time via *Get-all-latest* feature.

The comparison feature under *Manage Baselines* enables the comparison of models including version-managed packages. Comparison is possible for two models at a time up to the element level including diagrams. Branching can be realized by making a baseline using the *Manage Baselines* feature.

In Enterpise Architect, all clients communicate directly to the centralized version control system via local version management clients. This approach puts pressure on client maintenance because all users need both Enterprise Architect and the version management client installed and configured. Each version managed project needs to be configured separately. Enterprise Architect leverages three-tier technologies; there are three separate processes running (user interface, version management client, and version management server).

Version Management Features	*Evaluation question*	*Enterprise Architect 7.5*	*MagicDraw 16.0*
Check-in/Check-out	Is there support for Model, Package, Diagram and Element level check-in and check-out for multiple users? {Yes/No}	The Model and Package level check-in and check-out and the Element and Diagram level locking and unlocking are supported.	The Model and Package level check-in and check-out and the Element and Diagram level locking and unlocking are supported.
History	Is the Element level history available (who has made what changes)?{Yes/No}	No	No.
Model comparison	Is it possible to compare models at the Element level? {Yes/No}	Yes. Two models can be compared at the Element level including diagrams.	Yes. Two models can be compared at the Element level including diagrams.
Merging	Is there support for the Model, Package, Diagram and Element level three-way merge? {Yes/No}	No. Two-way merge is supported.	Yes. Three-way merge is supported.
Branching	Is there support for Model, Package, Diagram and Element level branching? {Yes/No}	No. Only Model and Package level branching is supported.	No. Only Model and Package level branching is supported.
Version Management Technologies	*Evaluation question*	*Enterprise Architect*	*MagicDraw*
Server-side technology	Are three-tier technologies supported? {Yes/No}	Yes	Yes
Client-side technology	Are client installations required? {Yes/No} Are there maintenance needs for the clients? {Yes/No}	Yes. Yes. Version management tool installation and configuration are needed.	Yes. No. Extra maintenance is needed for Teamwork servers.

Table 5. Comparing the version management features and technologies of MagicDraw and Enterprise Architect.

6.2 MagicDraw

The Teamwork server of MagicDraw allows the assignment of as many developers as necessary to work simultaneously on the same model on multiple workstations. The resulting model is saved and version-managed either on the Teamwork server or in a version management tool connected to the Teamwork server. Currently, MagicDraw can be used with two version management tools: Clearcase and Subversion. Models can be decomposed into sub-models at the package level, enabling model partitioning. Each package can be version-managed separately and checked-in, modified and checked-out. In addition, it is possible to lock, modify, and unlock package(s), diagram(s), and element(s).

Branching is realized at the model level. However, as models can consist of other models (modules), branching can also be considered to work at the package level. Model comparison (*Analyze/Compare projects*) can be used for three-way comparison and merge up-to the element level including diagrams.

No project-specific client configurations are needed for MagicDraw clients. Version management client installations are not needed either. This reduces the need for client maintenance and support. However, the Teamwork servers require extensive maintenance and support. MagicDraw leverages three-tier technologies; there are three separate processes running (the user interface of a MagicDraw client, Teamwork server, and the version management repository server).

7. EVALUATION OF THE FRAMEWORK

Both MagicDraw and Enterprise Architect support the package level check-in and check-out and locking and unlocking up to the element/diagram level. However, in both tools all the changes are saved at the model level. From a technical point of view, check-in/check-out thus requires lots of network traffic, increasing the time needed for check-in/check-out. MagicDraw enables three-way merging while Enterprise Architect provides only two-way merging.

Both MagicDraw and Enterprise Architect enable package level branching. Branching especially in product line organizations should be further studied because package level branching is not seen optimal for product line purposes. For example, if there is a new version of the component to be branched, it should be possible to make a branch for the corresponding element in the associated model. Both Enterprise Architect and MagicDraw lack element level histories. Availability of element level histories would help trace, who has made which changes to a particular element at what time. However, both tools help trace changes by enabling the comparison of models.

From a technical point of view, Enterprise Architect can be used in conjunction with many version management systems, thus being potentially more cost effective. After all, many companies already have comprehensive version management systems. Enterprise Architect requires more maintenance and configuration on the client side (i.e., version management clients need to be installed and configured) whereas MagicDraw requires substantial Teamwork server maintenance. The differences in technology may cause risks in availability and performance and increase maintenance needs. It is thus crucial for organizations to test the real perfor-

mance of the tools by experimenting with a variety of different setups of servers and clients.

Even if the products were quite similar in terms of features, the differences in technologies may increase maintenance needs and pose availability and performance related risks. The use of the framework thus provides essential information to support decision making during the evaluation projects.

Both products can be used in product line modeling because they provide the required basic features. For companies looking for more sophisticated version management features, MagicDraw is the best choice.

7.1 Lessons learnt

The evaluation project in the case organization draws attention to issues, which are general for all organizations considering the adoption of UML modeling tools.

During the project, it was noticed that the usability of the tools' version management features should be further studied because during the piloting phase users need to be spesifically instructed about version management capabilities. Organizations also need to consider the total cost of ownership separately because the possibility to use the already existing version management systems may reduce costs. Organizations planning to introduce UML modeling tools should always consider the framework and, additionally, evaluate usability, efficiency, and the total cost of ownership.

The fact that the two products have similar features also calls for the development of new more innovative solutions. For example, new advances in version management such as Distributed Version Control Systems (DVCS) [29] should be considered.

8. CONCLUSIONS AND FUTURE RESEARCH

The main deliverable of this paper is a framework consisting of a set of criteria for evaluating the version management features of UML modeling tools for multi-site, multi-partner software product line organizations. To illustrate and validate the framework, we applied it to evaluate two UML modeling tools. This study may serve as a baseline to find and implement new product development ideas for improving the UML modeling tools through the design science research [14]. For example, improving the usability of the tools and the capabilities of the users is expected to increase the benefits gained from modeling [3;10].

The results of this study serve as a basis to evaluate features of the UML modeling tools available in the software markets and the relationships between the features and successful deployments. Based on the experiences in the case organization, the deployment projects are more likely to fail if the modeling tools and services do not meet the requirements set in the framework. It is thus crucial to conduct further empirical research to understand better, which tool features will contribute most to the beneficial deployment of the tools.

This paper has focused on evaluating version management features of UML tools that follow the traditional centralized client-server model. However, new tools such as Git have appeared and the dominant ones such as Subversion have been further developed to leverage the Distributed Version Control Systems model challenging the centralized model. These tools operate in a peer-to-peer manner, enabling radical changes in systems development practices. Each developer using such a tool has a copy of the project's entire history and metadata. Developers can share changes in any way that suits their needs, not necessarily through a central server [29]. Although these tools and the enabled practices are not yet robust enough to be used organization-wide by global multi-site, multi-partner corporations, the tools are maturing quickly. Future research is thus needed to assess the applicability of the proposed framework for evaluating DVCS-based UML modeling tools and to revise the framework as necessary.

9. ACKNOWLEDGMENTS

The comments of Andrius Armonas, Erran Carmel, Mitchell Cochran and Rick Kazman greatly improved this paper.

10. REFERENCES

1. Appleton, B., Berczuk, S., Cabrera, R. and Orenstei, R. (1998). Streamed Lines: Branching Patterns for Parallel Software Development. In PLoP '98 conference. Available at: http://www.cmcrossroads.com/bradapp/acme/branching/

2. Arciniegas, J.L., Dueñas, J.C., Ruiz, J.L., Ceron, R., Bermejo, J. (2006). Architecture Reasoning for Supporting Product Line Evolution: An Example on Security. In T. Käkölä & J.C. Dueñas (Eds.), Software Product Lines: Research Issues in Engineering and Management. Springer, 327-372.

3. Arisholm, E., Briand, L.C., Hove, S.E. and Labiche, Y. (2006). The Impact of UML Documentation on Software Maintenance: an Experimental Evaluation, IEEE Transactions on Software Engineering, 32(6), 365 – 381.

4. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J. X, Møller-Pedersen, B., Oldevik, J., Tessier P., Thibault, J-P and Widen, T. (2006). Consolidated Product Line Variability Modeling. In T. Käkölä & J.C. Dueñas (Eds.), Software Product Lines: Research Issues in Engineering and Management. Springer, 195-241.

5. Bertolino, A.., Fantechi, A., Gnesi, S. and Lami, G. (2006). Product Line Use Cases: Scenario-Based Specification and Testing of Requirements. In T. Käkölä & J.C. Dueñas (Eds.), Software Product Lines: Research Issues in Engineering and Management. Springer, 425-444.

6. Brown, S. L. and Eisenhardt, K.M. (1995). Product Development: Past Research, Present Findings, and Future Directions. Academy of Management Review 20(2), 343-378.

7. Carmel, E. and Agarwal, R. (2001). Tactical Approaches for Alleviating Distance in Global Software Development. IEEE Software, 18(2), 22-29.

8. Carmel, E. and Agarwal, R. (2002). The Maturation of Offshore Sourcing of Information Technology Work. MIS Quarterly Executive, 1(2), 65-78.

9. Cusumano, M. A., and Selby, R. W. (1998). Microsoft® Secrets. Free Press, New York, NY.

10. Dzidek, W.J., Arisholm, E. and Briand, L.C. (2008). A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. IEEE Transactions on Software Engineering, 34(3), 407-432.
11. Funes, A., Dasso, A.., Salgado, C. and Peralta M. (2005). UML Tool Evaluation Requirements. Simposio Argentino de Sistemas de Información. http://www.frcu.utn.edu.ar/deptos/depto_3/34JAIIO/34JAIIO/asis/ASIS20.pdf.
12. Gomaa, H. (2004). Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. In R. L. Nord (Ed.): Software Product Lines, Third International Conference, SPLC 2004. Springer LNCS 3154.
13. Herbsleb, J.D. and Moitra, D. (2001). Guest Editors' Introduction: Global Software Development. IEEE Software, 18(2), 16-20.
14. Hevner, A.R., March, S.T., Park, J. and S. Ram (2004), Design Science in Information Systems Research, MIS Quarterly, 28(1), 75-105.
15. Kang, KC., Kim M., Lee J. and Kim B. (2005). Feature-oriented Re-engineering of Legacy Systems into Product Line Assets. In H. Obbink and K. Pohl (Eds.): Software Product Lines, Ninth International Conference, SPLC 2005. Springer LNCS 3714, 45-56.
16. Kleppe, A.G., Warmer, J., and Bast, W. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston, MA.
17. Kruchten, P. (1995). The 4+1 View Model of Architecture. IEEE Software, 12(6), 42-50.
18. Käkölä, T. (2008). Best Practices for International eSourcing of Software Products and Services. In Proceedings of the 41st Hawaii International Conference on Systems Sciences (HICSS-41). IEEE.
19. Käkölä, T. and Dueñas, J. (Eds.) (2006). Software Product Lines: Research Issues in Engineering and Management. Springer.
20. Käkölä, T., Koivulahti-Ojala, M., and Liimatainen, J. (2009). An Information Systems Design Product Theory for the Class of Integrated Requirements and Release Management Systems. Software Process: Improvement and Practice (in press).
21. Käkölä, T. and Taalas, A. (2008). Validating the Information Systems Design Theory for Dual Information Systems. In Proceedings of the 29th International Conference on Information Systems (ICIS), Paris. Association for Information Systems, http://www.aisnet.org.
22. Leppänen, S. (2008). Rigorous Service-Oriented Development of Communicating Distributed Systems, PhD Thesis, Tampere University of Technology, Publication 718, Tampere, Finland.
23. Lester, N. G. and Wilkie, F.G. (2005). Evaluating UML Tool Support for Effective Coordination and Communication across Geographically Disparate Sites. In Proceedings of the 2004 International Workshop on Software Technology and Engineering Practice, 57-64.
24. Van der Linden, F., Schmid, K. and Rommes, E. (2007). Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer.
25. Meyer, M.H. and Selinger, R. (1998). Product Platforms in Software Development. Sloan Management Review, 40(1), 61-74.
26. Norton, D. (2007). Open-Source Modeling Tools Maturing, but Need Time to Reach Full Potential. Gartner Research Report G00146580.
27. Object Management Group (2009). Unified Modeling Language: Superstructure. Formal Specification, version 2.2, 2009.
28. Oldevik, J., Solberg, A., Haugen, Ø. and Møller-Pedersen, B. (2006). Evaluation Framework for Model-Driven Product Line Engineering Tools. In T. Käkölä & J.C. Dueñas (Eds.), Software Product Lines: Research Issues in Engineering and Management. Springer, 589-618.
29. O'Sullivan, B. (2009). Making Sense of Revision-Control Systems. ACM Queue, 7(7). ACM, New York.
30. Pohl, K., Böckle, G. and Van der Linden, F. (2005). Software Product Line Engineering. Springer.
31. Ramasubbu, N., Krishnan, M. S. and Kompalli, P. (2005). Leveraging Global Resources: A Process Maturity Framework for Managing Distributed Development. IEEE Software, 22(3), 80-86.
32. Reuys, A., Reis, S., Kamsties, E., and Pohl, K. (2006). The ScenTED Method for Testing Software Product Lines. In T. Käkölä & J.C. Dueñas (Eds.), Software Product Lines: Research Issues in Engineering and Management. Springer, 479-520.
33. Ronnie, C. (1993). Formal Semantics: An Introduction. Cambridge University Press.
34. Rosenmuller, M., Siegmund, N., Saake, G., and Apel, S. (2008). Code Generation to Support Static and Dynamic Composition of Software Product Lines. In Proceedings of the 7th International Conference on Generative Programming and Component Engineering, 3-12.
35. Ziadi, T. and Jézéquel, J.-M. (2006). Software Product Line Engineering with the UML: Deriving Products. In T. Käkölä & J.C. Dueñas (Eds.), Software Product Lines: Research Issues in Engineering and Management. Springer, 556-588.