Collaborative Work with Software Models – Industrial Experience and Requirements

Lars Bendix[1] and Pär Emanuelsson[2]
[1]*Department of Computer Science, Lund Institute of Technology,*
Box 118, S-221 00 Lund, Sweden
[2]*Ericsson AB, S-583 30 Linköping, Sweden*
[1]*bendix@cs.lth.se,* [2]*par.emanuelsson@ericsson.com*

Abstract

When the initial problems of introducing and adopting model-driven development in a company have been handled, we want to go to work. That means that we have to supply our team of developers with a development environment and tools and processes that allow them to work efficiently. In many cases the team would like to work with models as if it was "just another programming language" and use the same techniques and processes for team collaboration and coordination they are used to from traditional development. Unfortunately some of the traditional tools and processes that work so well for traditional development do not work at all for model-driven development. Version control functionality is usually a key part in coordinating the parallel work in a team and through a set of use cases, we arrive at a number of requirements to a model-driven development environment that must be available for a team of developers to work efficiently.

1. Introduction

For many different reasons, the use of model-driven development becomes more and more widespread. The model "technology" per se seems to have matured and can bring several advantages – amongst which is higher developer productivity. However, while the model-driven approach in itself is now becoming mature, there still seems to be many problems with the tools and processes that are needed to support the developers in their work. Such support is already in place with development environments for more traditional development, but seems to be absent – or in its infancy – when it comes to model-driven development.

For the early adopters of model-driven development there have been many obstacles to overcome in order to switch from working with programming languages to working with models only. Now that this first line of problems from introducing models has been dealt with, we start to see a second line of problems that has to do with how to work with models. The developers would like their day-to-day work to be as much as possible just "business as usual" with all the support they had when doing traditional programming.

The "general" problem is how to handle collaboration when working with models. Much of this has to do with how a team of developers can coordinate their work [1]. However, there are also other aspects, like how to capture the history of the development to facilitate maintenance (understanding the model through its evolution history) – or to facilitate adding new functionality based on what has happened in the past.

We also need an organised and disciplined way of sharing what is being produced by the team. No team can work efficiently without team members working in parallel. Likewise we will need to carry out maintenance of older versions of the system in parallel with new development. These needs are usually satisfied through the use of a version-controlled repository. In both cases, branching and merging capabilities of such a repository can be used for this. Often we understand the present by looking at the past, so we would like to inspect the differences between the current version and some earlier version. This is all functionality that is at our service for traditional code-driven development, but which needs special tools for model-driven development.

In the present phase we have investigated collaboration in real projects. We have interviewed developers to see how they find working solutions by adapting available tools and processes. We are currently not looking for general solutions to general problems, but solutions that solve our specific problems. Future work will then look at how such

results can be generalised. The modelling tool used in the projects we have studied generates production quality (C++) code from UML models. The UML models consist of class and state charts diagrams combined with action code (attributed to state transitions and method bodies). The code generation is "forward" only, which means that the C++ action code is a part of the model and full code generation is done from the model (UML and action code). There is no round-trip process that allows edit of generated code. One of the main advantages with this set-up that it guarantees the consistency of model and generated code, which has been a problem for many other model-based approaches. In Figure 1 below, this corresponds to "model centric" development.

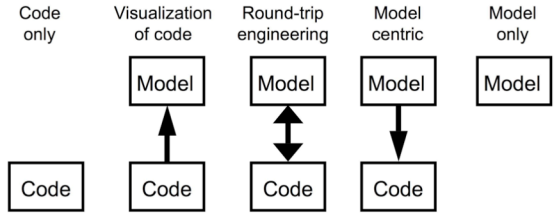

Figure 1. Ways of using models in development.

The outline of the rest of our paper is as follows. In the next chapter, we state the background for our work – both with respect to related work as to the context in which our problems occur. In chapter 3, we develop a number of use cases for collaborative work in general as well as for model-driven development in particular. This is followed by a more in-depth discussion of the consequences of these use cases with regards to general requirements to useful tool and process support for teamwork in model-driven development. Finally, in chapter 5, we draw our conclusions and sketch some future work to be done.

2. Background

In this chapter, we will give you an introduction to our problem domain. We do that by first giving an overview and discussion of related work, thus delimiting what parts of model-driven development we want to target. Then we give a description of the experience Ericsson has with using model-driven development, thereby highlighting the problems we have encountered and the wishes we have for the future.

2.1. Related work

There is very little previous work on the requirements to useful environment support for model-driven development – and to the best of our knowledge, there is virtually nothing on the collaboration aspects of a team and how they can use version control techniques and processes in their day-to-day work as a team.

The work of Meijler [9] seems to be focused on requirements to a meta modelling environment primarily for domain specific modelling languages. Storage and retrieval of models is mentioned but no details developed. There are also requirements to runtime semantics and incremental extensibility and ad-hoc adaptation of models, but again focus is on what to provide single developers and not on what consequences these requirements could have in a team setting.

The primary focus of Staron [12] is on the requirements for making the introduction of model-driven development into industry work smoother. Whereas this means that many aspects deal with overcoming what we called the first line of problems of introducing model-driven development, there are also a few aspects related to collaborative working with model-driven development. It should be possible to develop a system in cooperation between several units at the company. It should be possible to use models without a complete redefinition of current processes; and availability of tools is seen as an important issue. However, once again these requirements are of a quite general nature – though touching also on collaborative aspects – and their detailed consequences are not discussed.

The work that is most related to ours, is that of Selonen [11]. It is not a direct attempt at finding the requirements for support for teamwork using a model-driven approach. The primary goal is to make a review and comparison of different approaches to model comparison. The provided framework for the review makes use of a list of desirable qualities for model comparison techniques and a list of potential usage scenarios. In particular we find the usage scenarios interesting as they contain more detailed considerations of how model comparison is used and thus also more details for the requirements. However, as it is not a primary goal, there is no extensive discussion of the consequences for the requirements.

In Figure 2 a) below, it is shown how tools make it possible for a team to work and coordinate using sophisticated branching and merging strategies when working with text (traditional programming languages). However, because of lack of the same tool support in the model-driven domain, our teams of developers have to work in the much more restricted way that is shown in Figure 2 b), where branches are few and merges back to the mainline are manual and remain "invisible".

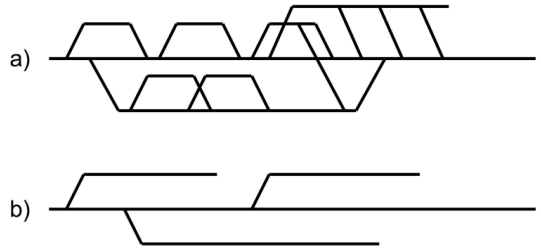

Figure 2: a) how we would like to work; b) how we are "forced" to work.

The situation in Figure 2 a) is the one that our developers would like to be able to use, both because it is less restrictive and because it is what they are/were used to when working with traditional development. This means that we should look at use cases from traditional development and in particular for use cases related to collaborative development. The drawback of this method is that we will not capture ways of working that are particular to model-driven development. However, we have also conducted interviews with developers who carry out model-driven development to be able to discover differences from the traditional case.

Several papers from the workshop on Comparison and Versioning of Software Models address topics that are related to providing support for collaborative work with models [2, 6, 10 and 14]. However, they do not address the topic in a systematic way and are more focused on providing solutions or possibilities than at looking at and analysing what are the actual needs to enable efficient collaborative work with models. Even our own paper at that workshop [3] focused more on describing the problems than actually distilling use cases and analysing and discussing their consequences for the detailed requirements. We do that here with the goal to be able to provide tool providers with a list of requirements for which they can look for existing results – and researchers with an agenda of problems for which industry looks for and needs to find solutions if model-driven development has to be efficient.

2.2. Experience

Ericsson has developed and deployed several large systems containing millions of lines of code that have been generated from UML models. Experience has largely been very positive. It is easier to understand the system for project participants with various roles. It is easier to handle complexity in the problem domain. Also system reliability, speed of execution and code volume has been satisfactory.

However many of the projects have not found a good way to handle collaborative aspects of the development and evolution of UML models. Of course a solution has to be found, but available solutions severely restrict several aspects of the development. One common solution is to avoid parallel development as far as possible in order to avoid merges. This can result in heavy planning work and static code owner assignments. Of course – a reasonable amount of planning is needed anyway and coordination of updates has to be done in some way. But to manage teams of hundreds of developers and guarantee that conflicting changes are not made can indeed be very complex.

Many projects would like to work in a feature-oriented way. In this kind of development a multi-disciplinary team develops a single feature of a system from start to end through all development phases. This development is quite different from having one team for requirements, one for design, one for test etc. All the feature teams would work on the same model and they should not have to do detailed synchronisation and planning between the teams. This kind of development requires efficient and reliable model merge.

Projects that really want to work feature oriented can be forced to choose programming instead of modelling. That would be very unfortunate and something that we would like to see changed – modelling should be compatible with any kind of technique that results in increased productivity and quality.

We will not take the position of "blame the tools". Rather we will study how projects divide models into manageable pieces and study how the different merge-avoiding strategies interact. Tools will not solve all problems and we will have to find strategies that work with non-perfect tools.

For our specific needs, we are only looking for a solution to a limited problem:

- All developers work on the same coordinated project, so everybody works in the same way if needed.
- Everybody is using the same set of tools.
- Historic versions of the models are always available (possible to make 3-way merge).
- Layout changes without semantic significance need not have an elegant solution, but can be essentially manual.

There are many problems when merging results done in different modelling languages or modelling tools with different goals (architecture models, testing models, production code generation). However, we think there are significant problems also when using the same modelling language and tool, with a complete history of development (making 3-way merges

possible), and when the team has frequent coordination meetings and there is a common "purpose and culture" in place. These problems are simple and already solved one might think, however there are many unsolved problems:

- Differences between models are often displayed in terms of a meta model that is not familiar to the user. Many of the meta model concepts do not have a straightforward mapping to the model elements that are the daily work items for the developer.
- Even when only a few changes have been done the compare/merge tools report many differences. There may be up to 10-20 differences (on meta model level) resulting from one change (on user level).
- Models may contain large textual parts. Model compare/diff tools may be weak on the textual parts although they should be as good as textual compare/merge tools.
- Some tools report on model layout changes and offer merge of these changes as well. It may be hard to distinguish layout changes from semantic ones.
- Layout differences are often a consequence of semantic changes. If for example an attribute name is changed to a longer name, the whole layout of a class might be changed, which results in a large number of layout meta model attributes being changed.
- A compare/merge tool often considers the content of a single unit even if there are associations to model elements in other units. It is hard for the user to know when such associations have been repaired or if strange situations will appear much later when the units are loaded into the same workspace.

In text based merge there is a totally semantics-free, language-free assumption: if there are two parallel edits on the same source-line there is a conflict otherwise there is no conflict. Is there a corresponding concept for models that we have not discovered yet?

The text-based merge assumption works really well for all languages (C++, Lisp, 1st order predicate logic) and even though we can construct examples where it fails miserably, it works fairly well in practise. Since UML language extensions and DSLs are more and more common, maybe something more language independent is needed?

We want to have the possibility to do large-scale development of models in parallel by large teams of developers and still be able to guarantee a high level of consistency and productivity. It should be possible to merge models effectively and **not** be forced to do:

- Excessive planning work as for example:
  - An extreme amount of planning in order to serialize work to prevent conflicts. That takes too much time for work that is non-productive.
  - Divide the modules into small fragments of controlled units in order to avoid conflicts.
- or, if conflicts occur, extraordinary measures such as:
  - Manual merges with the model editor.
  - Manual edits of the file representation of the model. That is of course a terrible thing to be forced into, but is often done in practise.

3. Use cases

In this chapter, we describe the use cases for collaborative development we have distinguished so far. First we describe more general use cases from traditional development and then more specific use cases from model-driven development. The template we use for describing our use cases is as follows:

Goals:
Actors:
Summary description:
Pre-conditions:
Post-conditions:
Subordinate or related use cases:
Discussion:
Sample scenarios:

Not all entries will be used for all use cases. In some cases we have not arrived at the proper refinement or definition yet and there are some use cases that are still on the "future work" list.

3.1. Use cases related to textual domain

The use cases described in this section have been inspired in part by [4] and in part by our own experience from traditional development.

a) Put project under version control.

Goals: Identify and structure configuration items and place them in a shared repository.

Actors: System architect/designer and Configuration Manager.

Summary description: As a part of the Configuration Identification activity [7] artefacts that are important for the project are identified, a structured for organising the artefacts is established and created in a repository.

Pre-conditions: System architecture/design is in place.

Post-conditions: All important project artefacts are in the repository.

Discussion: For traditional development, source code is usually the primary type of artefact. However, it is not unusual to include also requirements, design, test cases, documentation and more.

b) Work in isolation.

Goals: Carry out independent work without influence from others.

Actors: Developer (in the case of source code).

Summary description: A copy of the whole system is created that can only be modified by one single person.

Pre-conditions: A defined task and a defined state of the system.

Post-conditions: A system that has only been changed by a single developer.

Discussion: A common problem when many people change the same system is that it feels like shooting at a moving target. Furthermore, one person cannot be certain that a problem is due to him or some other person's changes. So in some situations it is best to be sure that only one person can make changes.

c) Integrate work.

Goals: To integrate work that has been done with the current state of the repository.

Actors: Developer – and the repository.

Summary description: After the completion of work that has been done in isolation, it has to be integrated with the current state of the repository.

Pre-conditions: A set of local changes and a repository (that may have changed).

Post-conditions: A repository where the local changes have been integrated.

Subordinate or related use cases: Related to 3.1 b) Work in isolation.

Discussion: Several people can carry out work in isolation in parallel. So when it is time to integrate, the repository may not be in the state it was when the isolation was established. In this case a merge between the local changes and the current repository must be done. In most cases it can be done by a tool, but in case of conflicts human intervention is needed.

d) Verify and validate merge result.

Goals: To make sure that the proposed merge result is correct.

Actors: Developer, compiler and test suite.

Summary description: In case a merge result is produced, we build the system from the result and smoke test it.

Pre-conditions: A proposed merge result.

Post-conditions: A checked proposed merge result.

Subordinate or related use cases: Related to 3.1 c) Integrate work.

Discussion: We cannot be sure that automated merge tools produce correct results. In particular, they have problems because they do not take into consideration neither syntax and semantics of the programming language, nor the logic of the system we are developing.

e) Investigate history.

Goals: To visualise how a component or system has changed in the past.

Actors: Developer (or maintainer).

Summary description: When trying to understand some code it is a great help to be able to see how it has evolved through time.

Discussion: Usually the history is shown as the difference between two versions of the same component.

f) Create awareness.

Goals: To make the developer aware of what is happening and potential consequences.

Actors: Developer – and team.

Summary description: The developer may be interested in knowing who is working on what, which things have changed, and what would be the consequences of an integration at the current time with (parts of) the system.

Subordinate or related use cases: Related to 3.1 b) Work in isolation.

Discussion: If we want to distinguish between different developers, they need to have different branches in the repository. It should be possible to show differences between not just the repository and a workspace, but between two workspaces. Likewise for merges that should be "hypothetical" and not actually carried out unless the developer decides so.

We have a number of additional use cases that we intend to look at in the future for a more refined analysis: g) Maintain old version h) Release system; i) Rename or move of file (as part of refactorings) – exotic/rare, here just to show previous textual work to an equivalent model use case; j) Split or combine of files – exotic/rare, here just to show previous textual work to an equivalent model use case.

3.2. Use cases related to Model-Driven development

The first two use cases describe development of different kinds of models, namely architecture and design models. The following three use cases describe more detailed merge situations that may occur both in development of architecture and design models.

a) Architecture model development.

Goals: Create an architecture model of a system.

Actors: System architect (develops the architecture model). System engineer (receiver of the architecture model).

Summary Description: Provides the system context to project participants. Describes important system interfaces that are the base for system development. A model of the system architecture is developed. It is the basis for many planning activities, such as resource planning and change impact analysis. It can also be verified for consistency with models used for design and test.

Pre-conditions: System requirements exist.

Post-conditions: An architecture model has been developed.

Discussion: There are not that many people involved in the development (compared with design) and architecture models are primarily graphical (no action language code is needed for example).

b) Design model development

Goals: Create a design model for a system.

Actors: System engineer.

Summary Description: Provides the detailed system description that can be used to make full code generation. A design model of the system is developed.

Pre-conditions: System requirements and system architecture models exist.

Post-conditions: A detailed and verified system model exists.

Discussion: There are many system engineers (hundreds) involved in the development. The models are very large. Development is often done at distributed sites. Models have both graphic and textual parts. Accuracy of results from compares and merges is very important.

c) Model update without merge

Goals: Create a new version of a number of configuration items of the model.

Actors: Model developer.

Summary description: When the items of the model are checked in, the version control tool discovers that there have been no other updates of these items so they can be stored as they are.

Pre-conditions: The model is available in a repository.

Post-conditions: The updated and verified model is in the repository.

Discussion: This is the simplest case and no model compare/merge tool is needed as the "comparison" is done from file system information (time stamp).

d) Model update with simple merge

Goals: Create a new version of a number of configuration items of the model.

Actors: Model developer.

Summary Description: The model developer checks out a number of items. The model editor is used to update the model. The model developer checks in the items. The version control tool discovers that there have been other updates to some items. A model merge tool is then activated and the two updated items and a common ancestor are identified by the version control tool and sent over to the model merge tool. The model merge tool compares the three items and automatically generates a merged model since all conflicts can be resolved.

Pre-conditions: The model is available in a repository.

Post-conditions: The updated and verified model is in the repository.

Discussion: The user should be able to inspect and understand the changes that have been made and verify that there are no real conflicts (there are always conflicts that a tool cannot discover and that need to be manually verified). It should be possible to double-check (in cases of uncertainty) that the result is what the user expect.

e) Model update with complex merge

Goals: Create a new version of a number of configuration items of the model.

Actors: Model developer.

Summary Description: The model developer checks out a number of items. The model editor is used to update the model. The model developer checks in the items. The version control tool discovers that there have been other updates to some items. A model merge tool is then activated and the two updated items and a common ancestor are identified by the version control tool and sent over to the model merge tool. The model merge tool compares the three items but cannot perform an automatic merge. The user is asked how to combine results from the two conflicting models.

Pre-conditions: The model is available in a repository.

Post-conditions: The updated and verified model is in the repository.

Discussion: It is essential that the user can inspect and understand the changes (as in use case 3.2 d). Additionally the model merge tool should not allow inconsistent model fragments to be selected. Model inconsistency that results from the combination of several edits should be discovered.

We have a number of additional use cases that we intend to look at in the future for a more refined analysis: f) Quality checks/inspections g) Migration of model to another language/tool; h) Create a new project/model (if it is not captured by 3.2 a)); i) Model verification and validation (if it is not captured by 3.2 f)).

4. Discussion

The discussion in this chapter is based on the use cases we identified the previous chapter. For the use cases from section 3.1 we will consider how it works in the traditional domain and how we would like it to work in the model domain – and what we would need to obtain that. For the use cases from section 3.2, we will consider how we would like it to work and what we would need to obtain that.

We really want to do not just model-driven development, but also feature-based development, which means that it is an absolute must that people can work in parallel everywhere in the code and in a very dynamic and flexible way. This goes in particular for the different feature teams but also for the developers within one specific feature team. So we cannot use a static split-combine solution [8], as feature teams will work in different "areas" for different features.

This leaves us with the fact that we will have to provide support for an optimistic copy-merge way of working in parallel. Our developers are used to working with such a strategy and would like to be able to do that also for model-driven development. As a consequence good merge support becomes a fundamental requirement. Because this merge support is not optimal today, our developers do implement merge-avoiding strategies, like the one shown in Figure 2 b). It has some advantages, because it keeps the developers from excessive branching, which can be a problem in itself. However, it also keeps the developers from adopting useful team development processes like Continuous Integration [5].

When looking for proper merge support, we are very interested in tools that provide us with correct merge results – and that find as many merges as possible without giving up and signalling a merge conflict that requires human intervention. In the traditional textual domain, research has already considered to include the syntax and semantics of the used programming language to provide better merges. However, that has never made its way into commercial tools. Most probably because the cost-benefit is not right. There are high costs involved with adapting the merge tool to any language that needs to be supported – and to keep up with the evolution of the language. The pay-offs also seem to be marginal, as "simple" merge tools do fairly well. In many cases when human intervention is needed, the conflict can be resolved pretty easily. In the remaining few cases, even an "intelligent" tool would not have been able to solve the conflicting changes.

When we consider the quality of a merge tool or algorithm, we have to focus on two concepts from statistics: precision and recall. The higher the precision, the fewer times the tool will report a false successful merge. The higher the recall, the less times the tool will give up and report a conflict that requires human intervention.

In the ideal case, we would look for 100% precision and 100% recall, but in traditional development, people are satisfied with less and put more emphasis on recall than precision. The reason why we look for a high recall is that we want the tool to automatically do as many merges as possible. Often this happens at the cost of the precision – it will more often report a merge as being successful when indeed it is not a correct result. However, as long as we have other ways of telling that the tool was wrong, this is not a big problem. Indeed, in traditional development, the merge process consists of three steps: the merge, build check, and smoke check. When the second and third steps are not too heavy, there is not a high cost to pay for the lower precision.

In the case of model-driven development, there is currently a different trade-off. It is very time consuming to carry out the build and smoke checks. We would like to see work on how to quickly verify models to see if they are still syntactically and semantically correct. Likewise, we would like to see work on how to quickly validate the logic in the system we are developing. If that can be obtained, then we can also relax our requirement that merge algorithms for models should have 100% precision.

With regards to the recall – the ability to find as many of the merges as possible – we believe that model merge should have better possibilities than in the simple textual case. The reasons for this are that there is much more information available when the merge result has to be produced.

A final aspect regarding merge, is the presentation of the merge result. In the good old days of simple tty screens writing lines of green text, merge and diff visualisation was horrible. However, today there are nice GUIs showing information in a very intuitive way – an example of this is shown in Figure 3. We are presented with both alternatives to be merged (at the top left and top right) and also the common ancestor (at the top middle). We also see the proposed merge result (at the bottom) – and colour is used to make evident from where the proposed result originated and where something has been merged and where the code has remained unchanged. It is possible in each case to choose whether to include a change or not and in case of a merge conflict to choose which alternative we want. It is even possible to manually edit the proposed result before we finally save it. Something similar would be much appreciated also for model-driven development.

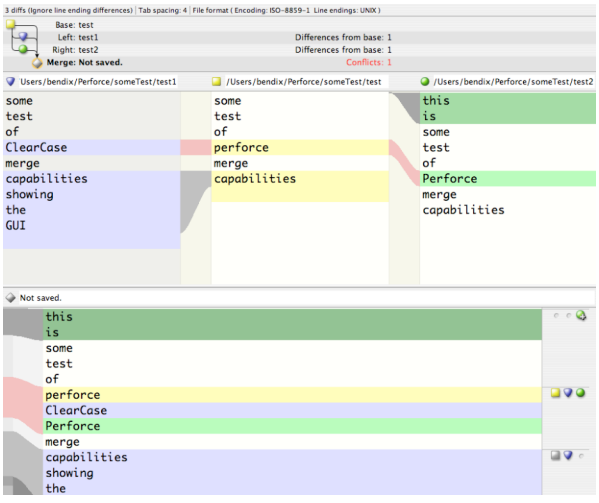

Figure 3. Merge in the textual domain with a nice graphical user interface.

5. Conclusions

Some of the requirements we have identified are already satisfied to some degree at Ericsson today. Merges can be done and differences shown, but the quality lacks a lot. From our requirements we can see where improvements can be made.

We propose that tool vendors and researchers try to solve the special case (we use UML) before they try to attack the general case. This also means that we can "accept" a constraint of universally unique IDs in the first case – and get the "similarity"-based stuff [13] later.

6. References

[1] Wayne A. Babich, "Software Configuration Management – Coordination for Team Productivity," Addison-Wesley, 1986.

[2] Christian Bartelt, "Consistency Preserving Model Merge in Collaborative Development Processes," in Proceedings of the International Workshop on Comparison and Versioning of Software Models, Leipzig, Germany, May 17, 2008.

[3] Lars Bendix, Pär Emanuelsson, "Diff and Merge Support for Model Based Development," in Proceedings of the International Workshop on Comparison and Versioning of Software Models, Leipzig, Germany, May 17, 2008.

[4] Lars Bendix, Otto Vinter, "Configuration Management from a Developer's Perspective," in Proceedings of the EuroSTAR 2001 Conference, Stockholm, Sweden, November 19-23, 2001.

[5] Martin Fowler, "Continuous Integration," http://www.martinfowler.com/articles/continuousIntegration.html, May 2006, retrieved January 3, 2009.

[6] Maximilian Kögel, "Towards Software Configuration Management for Unified Models," in Proceedings of the International Workshop on Comparison and Versioning of Software Models, Leipzig, Germany, May 17, 2008.

[7] Alexis Leon, "Software Configuration Management Handbook (second edition)," Artech House, 2004.

[8] Boris Magnusson, Ulf Asklund, Sten Minör, "Fine-Grained Revision Control for Collaborative Software Development," in Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, Los Angeles, California, December 8-10, 1993.

[9] T. D. Meijler, "Requirements for an Integrated Domain Specific Modeling, Modeling Language Development, and Execution Environment," in Proceedings of the 3rd Nordic Workshop on UML and Software Modeling, Tampere, Finland, August 29-31, 2005.

[10] Leonardo Murta, Chessman Corrêa, João Gustavo Prudêncio, Cláudia Werner, "Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System," in Proceedings of the International Workshop on Comparison and Versioning of Software Models, Leipzig, Germany, May 17, 2008.

[11] Petri Selonen, “A Review of UML Model Comparison Approaches,” in Proceedings of Nordic Workshop on Model Driven Engineering, Ronneby, Sweden, August 27-29, 2007.

[12] Miroslaw Staron, “Adopting Model Driven Software Development in Industry – A Case Study at Two Companies,” in Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, October 1-6, 2006.

[13] Christoph Treude, Stefan Berlik, Sven Wenzel, Udo Kelter, “Difference Computation of Large Models,” in Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, September 3-7, 2007.

[14] Sven Wenzel, “Scalable Visualization of Model Differences,” in Proceedings of the International Workshop on Comparison and Versioning of Software Models, Leipzig, Germany, May 17, 2008.