Requirements for Practical Model Merge – An Industrial Perspective*

Lars Bendix[1] and Pär Emanuelsson[2]

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

Abstract. All the support tools that developers are used to must be in place, if the use of model-centric development in companies has to take off. Industry deals with big models and many people working on the same model. Collaboration in a team inevitably leads to parallel work creating different versions that eventually will have to be merged together. However, our experience is that at present the support for model merge is far from optimal. In this paper, we put forward a number of requirements for practical merge tools, based on our analysis of literature, merge tool evaluations, interviews with developers, and a number of use cases for concurrent development of models. We found future work to do for both tool vendors and academic research. Fortunately we also uncovered a few tips and tricks that companies using model-centric development can implement on the short term while waiting for better times.

Keywords: Model merge, diff, version control, parallel work, team coordination, industrial experience.

1 Introduction

In industry, the use of models in development is gaining momentum. However, from our experience there are still a number of obstacles that have to be overcome before the use of model-driven development can really take off. Models can be used in many different ways ranging from simple visualization of code to a pure model-centric approach where the model is the sole focus of attention and executable code is generated directly from the model and never looked at or manipulated. For the past couple of years Ericsson AB has started to use the model-centric approach on more and more of its projects and has had some successful experience, but has also suffered from being an early adopter of a "technology" that for some aspects is still not fully mature.

Industrial use of model-driven development means not just creating big and complex models, but more importantly also the involvement of many people working on the same model. A key factor in the adoption of model-centric development is the presence of mature tools that can support the developers when they carry out their

* Empirical results category paper.

tasks. The most basic set of tools necessary consists of a model editor and a model compiler. This will allow the single developer to create and manipulate a model and to compile it into running code. However, this will not scale to an industrial project where many people are involved. Effective collaboration in a team requires that people can share information and the existence of groupware that can help groups communicate, collaborate and coordinate their activities [11].

On a more traditional development project where textual programming languages like C, Python or Java are used, version control tools and functionality often work as the groupware that helps a team manage its coordination. They will address problems like "shared data", "simultaneous update" and "double maintenance" [4] that are intrinsic parts of team collaboration and cannot be avoided, but instead have to be managed to allow the team to work efficiently and without making mistakes. The version control tool supplies a workspace to manage the "shared data" problem and concurrency detection to help manage the "simultaneous update" problem. Merge functionality – that may or may not be part of the version control tool – helps manage the "double maintenance" problem by allowing for easy conflict detection and resolution. Finally, diff functionality allows the developer to get information about how two versions in the repository – or a version in the repository and the version in the workspace – differ.

Our experience at Ericsson AB was that the version control support worked pretty well, though not optimal, when working on model-centric projects. However, the merge/diff tool support was far from optimal. From an informal tool evaluation we did in early 2007, it emerged that even for extremely simple examples, the merge results could often be counterintuitive or downright wrong – in some cases the merge result produced would not even load in the model editor. In a different study covering more tools the authors even went so far as to conclude that the "state of model merge tools is abysmal" [5]. It was decided to investigate more carefully the maturity of model merge with three objectives. First, to find solutions that could be implemented immediately by developers and projects. Second, to discover results from research that could be integrated in the tools provided by vendors. Third, to distinguish and define problems that need to be researched to provide more mature support for model merge.

In a previous paper, we reported on our results from an initial literature survey of academic research on model merge and an initial analysis of similarities and differences between text merge and model merge [6]. In a later paper, we proposed and discussed the consequences of a number of use cases for text and model merge, based on problems and suggestions that emerged from interviews with developers at more sites within Ericsson AB [7]. In this paper, we first present the context for the experience reported and clarify the terminology we have chosen to use. Then we give a more thorough analysis of relevant use cases from [7] with the aim to distil a number of requirements for a practical model merge tool. These requirements are then grouped into related themes that are discussed in more detail, and finally we draw our conclusions. This paper is based on a recent more thorough evaluation of model merge tools [18] and further interviews with developers.

2 Background

In this chapter, we will first describe the context in which we have obtained the experience we report, then we give a brief review of previous work done in this field followed by a clarification of the terminology that we use in this paper and finally the delimitations that will hold for the subsequent analysis and discussion.

Context. Ericsson AB has developed several large systems with millions of lines of code using UML in a model-centric way. This means that executable code is generated directly from the models and only models are considered for work, whereas generated code is never looked at or manipulated. We are able to obtain good reliability of systems and execution speed and code volume is acceptable. Furthermore, we have seen positive effects on code comprehension and on system complexity. As such the use of model technology has proved a success for industrial use and Ericsson AB would like to continue.

However, there are still a number of unresolved problems on the collaborative level because of the immaturity of tool support. We often work in big projects with up to 100 people working on the same model. On some projects people are even distributed on more than one site, making collaboration even more difficult without proper tool support. Without this support people resort to doing manual merges using three screens for the two alternatives and the result; instead of having tool-supported 3-way merge. Or they use a text merge tool on the textual representation of the model; which works in some cases, but requires knowledge of the representation format.

These "solutions" make it possible for the project teams to survive the use of model-centric development, but we would like to see better direct support from tools and processes.

Previous work. For textual documents there has long been mature merge tool support and early on the software configuration management (SCM) research community started to look at widening merge support. Early attempts were on structured documents in the context of structure-oriented environments [22] and syntax-directed editors [3]. However, since neither structure-oriented environments nor syntax-directed editors caught on that line of research died out. Interest has more recently resurfaced with the widespread use of structured texts and documents and thus the need to be able to provide support for collaboration for such structures. One line of research focuses on hypertext systems [17] whereas another line looks at models in general and UML models in particular [19].

People from outside the SCM community have also shown interest in looking at how to support the collaborative work on models. Early research focused on the detection and visualization of differences [23], [13] of diagrams to allow people to understand and analyze the evolution of changes to diagrams. More recently this has been extended to include merging of diagrams [14], [24] to support also the reconciliation of parallel work. This interest from the model community has grown into two lines of interesting workshops – one that looks more specifically at the technical versioning aspects of models [9], [10] and one that treats more general aspects of model evolution [15], [16].

Terminology. Reading through the literature from the model community we encountered some problems with terminology that made it difficult to know precisely what was being talked about and caused us some initial confusion. To avoid similar confusion in the readers of this paper, we find it proper to clarify the terminology we use here. There is a line of research on model merge that has a much more theoretical and mathematical approach [1], [2], [21], [20] and [8] than the more technical approaches mentioned above. Most of that work – though not all – focus on merge of different types of models and not of different versions of the same model. They "borrow" much from the mathematical world and work on defining and using the algebraic properties of operators on models. So we end up with many operators that sometimes have different meanings for the same operator. Here we define our meaning of four of these operators:

diff vs. *compare*: diff computes the differences between two versions of the same type of model (eg. class diagram); compare computes the differences between two models of different types (eg. class diagram and sequence diagram)

merge vs. *union*: merge integrates two (parallel) versions of the same type of model (eg. class diagram) and usually (in this paper) is a 3-way merge with a common ancestor; union integrates two models of different type (eg. class diagram and sequence diagram) and usually is 2-way without a common ancestor

Premise. What we present in this paper does not pretend to be general. It is based on the experience from one company – though from several independent branches – and the analysis and discussions are targeted at the specific needs of that company. However, since we believe that model-centric development and its problems do not vary much from company to company, we are confident that most of our findings – even with the delimitations below – will be generally applicable and of interest also to a wider audience.

In this paper, we treat *diff* and *merge* only as we are interested in working on different versions of the same type model. We detail only (mostly) merge as we consider diff to be a part of and a pre-requisite for a merge and therefore it shares similar problems. We have version control of models so historic versions are available and 3-way merge always possible. We talk about UML – and not models in general – and we can (and do) have UUIDs. All people on a team will use the same tools and processes and we value tools that integrate with other tools over integrated frameworks because that allows for more flexibility in setting up a working environment. Finally, we have a bias for feature-oriented development, which means that more feature teams will have to modify the same (parts of a) model at the same time.

3 From Use Cases to Requirements

In this chapter, we analyse a number of the use cases that were presented in [7]. We use them to distil more detailed requirements for practical model merge support. We briefly describe and motivate each use case. This is followed by an analysis of the use case, where we relate the model case with the traditional support in the text case. Finally we state and briefly discuss the use case's consequential requirements.

At this point we do not discriminate between requirements that are targeted at the version control tool, the model merge tool, the model language or the model work process. A more detailed discussion of the interrelations and dependencies between and the consequences of the requirements will be given in chapter 4 below. The use cases are intended to give the context in which model merge will have to live and as such hints at how it should work and what the requirements are if there is to be the same support for model merge as for text merge.

Some of the use cases from [7] are not used here for several reasons. Use case 3.1.b: *Work in isolation*, because it was meant to highlight collaboration in general and is not relevant to a specific analysis of merge support. It is part of what is supplied by the version control tool through the concept of a workspace – the consequence of which is that we may need to merge the parallel work done in more workspaces. Use case 3.1.c: *Integrate work*, is covered by and detailed in use cases 3.2.c-e that will be treated below. Use case 3.1.f: *Create awareness*, is outside the scope of this paper – it is usually part of the support supplied by the version control tool and is not specific to merge support. Use cases 3.2.a: *Architecture model development* and 3.2.b: *Design model development* are also left out here, because their primary purpose was to show the need for a compare operation and the varying number of people working in parallel – in this paper, we focus on the design setup.

3.1 Put Model under Version Control

Description and motivation: The version control system is the primary source of groupware support for a team. Furthermore, we would be interested in recording the history of evolution of our model.

Analysis: Traditionally when we put a project under version control, we have to select the configuration items (CI), which are the artefacts that we want to version. Usually version control systems handle files as CIs, so we need to supply the system with a set of files that make up the model. One extreme would be to have the whole model in one single file, another extreme to have each single model element in a file of its own.

Requirements: We will need the modelling language to have a mechanism for splitting up a model so it can be placed in several files, and we will need the version control system to support flexible units of versioning:

- *flexible unit of versioning (UV)*. The UV is used by the version control system for concurrency detection. The finer the UV, the better the version control system can decide if parallel changes touch the same or different parts of the model. However, the finer the UV, the more fractioned the model will appear to the developers and the more work they will have in managing the version control. It is important for the developers to have flexibility for the UV, so they can tailor it to their specific needs.
- *modularization mechanisms*. The model language must have a mechanism for physically splitting up a model in smaller parts. If that is not the case, everyone will be working on the same artefact and all parallel changes will create a concurrency conflict triggering a merge situation for the artefact.

3.2 Investigate History

Description and motivation: When we have the historical evolution of a model preserved in the version control system, we would like to investigate that history to discover what changed between two specific versions. More generally, we would like to know what is the difference between any two versions of an artefact whether they are in the repository or in the workspace.

Analysis: We use the version control system to keep track of the versions we create of an artefact and that are committed to the repository. This will give us an overall picture of the evolution of an artefact. In case we want to know the details, we need an operation that given two versions from the repository can tell us exactly how they differ. Such an operation can also be used to tell the difference between a particular version in the repository and the version we have in our workspace. In all cases we will have a 3-way diff as there will always be a common ancestor – also to the workspace version as it has been checked out from the repository. For such a diff to be of practical use, it should present the differences in a way that makes sense to the user.

Requirements: We will need a diff operation, attention to presentational issues, a work process that focus on logical tasks, and flexibility in the unit of comparison:

- *diff operation*. Because we work in a context where we have the same type of model, we do not need a compare operation that can tell the differences between different types of models. The diff operation should be detached from the version control system to allow us more flexibility in selecting tools. Since we are not working on text files, the diff operation should be tailored to the type of model that is addressed.
- *presentational issues*. In order not to create information overflow, only important differences should be shown. For our context we do not consider layout changes to be significant. However, a good filtering mechanism will allow the user to define what he wants to see at any given moment.
- *work process that commits logical tasks*. Once the tool has shown the differences between two versions, we have to make sense of the details. To try to recreate the logical intention behind a number of detailed changes. This task is greatly helped if the work process prescribes that only complete tasks are committed and if each commit has a short log text associated with it.
- *flexible unit of comparison (UC)*. Just as for the presentational issues, we need flexibility in the unit of what is compared. In the textual case we do not want to be told that a file has changed; we want to know what line was changed and sometimes even what changed on that line. Likewise in the model case. Flexibility in the UC will allow us to tailor the diff operation to give us information at the level of detail that we are interested in at the moment.

3.3 Model Update without Merge

Description and motivation: When parallel development has happened we want to synchronize the work at some point. In the case where work has not been done on the same artefacts, we do not need to carry out a merge but can simply take the sum of changes to be the result.

Analysis: The normal way of working of a version control system is that when we want to commit our changes, it first carries out a concurrency check. If something new has arrived in the repository since we last updated our workspace there is a physical conflict on some of the artefact, and we will need to update our workspace version to avoid getting the “simultaneous update” problem [4]. However, if the artefacts that we changed have not changed in the repository, we can do a commit and add a new version. However, this is not always enough to ensure that we will have consistent configurations in the repository as there may be logical conflicts that are not detected by this mechanism.

Requirements: We will need a transaction mechanism that takes into account logical consistency:

- *strict long transactions*. The long transaction model [12] works as described in the analysis above and thus opens up for inconsistencies. However, the strict version of long transactions does the concurrency check at the logical level where we perform the commit. If anything has changed in the repository since we last updated our workspace we are not current anymore and must update – even if changes in the repository only regards artefacts that we have not changed in our workspace. Strict long transactions do not detect inconsistencies, but force us to update and create a new “configuration” in our workspace instead of directly in the repository. This means that we have the possibility to check for inconsistencies in the updated configuration before we finally commit it to the repository. For strict long transactions to be practical, we should be able to commit – and thus carry out concurrency checks – at other levels than the top level. Otherwise we will always be forced to do an update even when changes in the repository regard completely unrelated parts of the system.
- *flexible unit of versioning (UV)*. This will allow the developer to decide the granularity of concurrency detection. In the textual case the UV is always the file, but the developer decides what to put into the file. If that is not the case for models, then we would need the UV to have more flexibility.

3.4 Model Update with Automated Merge

Description and motivation: When work has been carried out in parallel on the same artefact(s), there will have to be performed a merge of the changes as part of the model update. In the simple case, the merge tool will be able to automatically resolve the changes and produce a successful merge result.

Analysis: The issues of concurrency detection were dealt with in the previous use case 3.3, so in this use case there is actually a physical conflict at the level of unit of versioning for at least one artefact. To be able to automatically resolve the conflicting changes we need to go more into details. We look at the internal structure of the unit of versioning to see if there are conflicting changes at the level of the unit of comparison. If not, we will have the same situation as for use case 3.3, but at the level of a single artefact and not of a whole configuration – this includes both the capability of producing a merge result, and the possibility of this result being logically inconsistent.

Requirements: We need to be aware of the semantics of the merge operator and we need flexible unit of comparison to allow better conflict resolution:

- *flexible unit of comparison (UC).* The granularity of the UC decides the level at which we can do conflict resolution. The finer the granularity, the better and more precise we can distinguish differences and decide on how to automatically resolve the merge. If the level of granularity is a class, then changes to different methods by different people will create an irresolvable merge conflict. Likewise, if "action code" is the unit of comparison then any modification to the "action code" will flag the whole "action code" as changed. It would be more helpful if "action code" had a UC at the level of a line of text, as we could then distinguish exactly what lines were changed and have better possibilities for resolving parallel changes to the "action code". However, if the level of granularity becomes too fine, then we can suffer performance penalties – and "incorrect" semantic behaviour at the level of the unit of versioning as discussed below.
- *semantics of merge operator.* Now that we – or rather the automated merge – actually change the internals of our model, it is important that it is clear in which way these changes are done. In the traditional text merge tools the semantics of the merge operator is very simple. If a line of text is changed in one of the alternatives, then it is also changed in the result. For model merge tools the situation will be much more complex. In many cases there will be more than one possibility and only explicit semantics of the model merge operator will make it clear for the user what happens. Furthermore, the text merge tool only guarantees to produce a "correct result" according to its own semantics – which is "lines of text" – even though the real semantics of the contents of the merged artefact is often quite different. From this perspective, it is perfectly reasonable for the merge tool to produce a result in the case where the declaration and use of an identifier has been removed in one alternative and a new use of that identifier added to the other alternative. It is obvious that what is correct semantics for the text merge tool will not be correct semantics for the Java compiler. It is not clear whether it will be possible – or practical – to avoid such a mismatch in semantics for model merge.

3.5 Model Update with Merge Conflict

Description and motivation: When work has been carried out in parallel on the same artefact, there will have to be performed a merge of the changes as part of the model update. In the complex case, the merge tool will not be able to automatically resolve the changes and will announce a merge conflict.

Analysis: From the analysis of use case 3.4 above, it is clear that we have the same needs for clear and explicitly defined semantics in this case. Likewise, the granularity of the unit of comparison is equally important for the possibility or impossibility to automatically resolve merge conflicts. The only difference to the above use case 3.4 is that in case both alternatives have changed the same unit of comparison, it will not be possible for the merge tool to choose which alternative to use – a merge conflict will have happened and the user will have to manually resolve it.

Requirements: We will need clear semantics, flexible unit of comparison, and definition of how to present conflicts:

- *flexible unit of comparison (UC).* Identical to use case 3.4 above, so we refer to that discussion.
- *semantics of merge operator.* When we deal with the simple semantics of the text merge operator, it is clear that parallel changes to the same UC (line of text) will have to create a conflict. However, for model merge the case might not be that simple. However, in the model case there is much more information available than just "some text has changed" and the UCs might not always be of the same type. So it might in some cases be possible to define a reasonable merge result – or a preference to one alternative over the other – even when both alternatives have been changed.
- *presentational issues.* We need to deal with the presentation of merge conflicts at two levels. First, the presentation of the conflict has to be in such a way that it is clear to the user what the conflict consists of. As stated above, the richer semantics of the model merge operator should make it possible to provide that information. Second, the representation of the conflict has to be in such a way that the merged result can be loaded and modified in a model editor. This means that the underlying representation of the model will have to be able to handle and represent conflict markers. In text merge such conflict markers are not a problem for the editor, as they are text too – and standardization of the conflict markers have even made it possible to present conflicts in graphical editors.

3.6 Verify and Validate Merge Result

Description and motivation: Once we have produced an automated merge result, we would like to verify and validate its correctness.

Analysis: It will have become evident from the discussions in use cases 3.3, 3.4 and 3.5 above, that it will be virtually impossible to guarantee always 100% correct merge results. In text merge, that is a well-known fact and it is common practice to always check the result by doing a "build-and-smoke" test after an announced successful merge. Because that test is relatively fast and easy to do for the textual domain, users tend to have a preference for a high recall at the cost of a lower precision in the merge results. When discussing verification and validation of merge results, it is important to notice that also in the case of use case 3.3 above, there is actually performed a merge. The merge is not done at the level of the unit of comparison as in use cases 3.4 and 3.5, but at the level of unit of versioning.

Requirements: We should be able to verify the syntax and semantics and to validate the model logic:

- *verification of "syntax and semantics".* In text merge this is a simple compilation of the program. If our model merge operator is not able to guarantee that the result always respects the syntax and semantics of the model language, we need to do a similar compilation for models too.

- *validation of "program logic"*. For text merge it is common practice to have a small suite of test cases that will catch the most blatant mistakes. It is much faster than complete testing and experience has shown that in most cases it is sufficient for finding merges that mistakably are announced as successful merges. A similar process should be adapted for model merge.

4 Discussion

In this chapter, we will discuss in more detail the requirements that we have identified in the preceding chapter. For ease, we have grouped the requirements into three related themes – semantics of model merge, division of responsibility and presentational issues – that are discussed for possible consequences of the requirements and for what should be taken into consideration when implementing them.

4.1 Semantics of Model Merge

This is in our opinion the most important and also controversial aspect of model merge. The semantics of text merge are really simple, if a line of text has changed it has changed. If the same line of text has changed in both alternatives, there is a conflict. There have been attempts at more fine-grained unit of comparison by looking at the syntax and semantics of the contents. However, lines are still what rules for text merge in practice.

For models the situation is not that simple. It is indeed possible to exploit the underlying textual representation of the model and use a textual merge tool. However, even minor changes to the layout of a model can have big consequences for the order in which things are stored, which will cause insurmountable problems for a textual merge tool. So we see no way around using the implicit structures dictated by the model language for a model merge tool. Furthermore, such an approach will also benefit from the possibility of defining a "richer" semantics, since we will have several different "types" of units of versioning and not just one (lines of text) as in textual merge. There have been early attempts to define such "rich" semantics for merge of structures [3] and [22], that revealed several cases where the desired merge result was open for discussion. The work of [8] is a first step towards model merge semantics, though they are more focused on the algebraic properties and compare and union operators.

That defining model merge semantics is not that easy can be seen from figure 1. One developer restricts the multiplicity of the class' relation to "0..2", while the other developer in parallel restricts it to "1..10". Now what should be the merged result of this: "1..2" (the most restrictive), "0..10" (the least restrictive) – or something third? In our opinion, the developer should never be left guessing, so in case the merge is automatically resolved, the result should never be a surprise – otherwise a merge conflict should be flagged. A recent tool evaluation [18] revealed other "unpleasant" surprises. One setup was that in both alternatives a new class with the same name as the existing class was added and the two alternatives were merged. One tool decided that the names were the same and therefore merged the two classes into one. The other tool decided that the two classes were different and kept both classes in the merge result. One can argue for the correctness of both approaches. In the first case, the merge tool made its decision based on the "similarity" of the two classes whereas the second tool made its decision based on the different UUIDs of the two classes (ignoring that they had the same name). This shows that in some cases model merge semantics are very open for interpretation. Therefore it is very important that the tool vendors make these semantics very explicit – and that the users continue to meticulously read the manuals until common standard semantics are agreed upon.

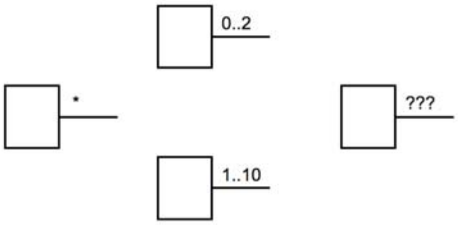

Fig. 1. Simple model merge dilemma

4.2 Division of Responsibility

The merge tool in itself is only a part of the groupware support for collaboration. In the great picture of support for the parallel work of a team we need more than just the physical merge of two artefacts – and we need to decide which tool should take care of which tasks. For parallel work there are two tasks: the concurrency detection and the conflict detection and resolution.

Concurrency detection is the discovery that parallel work has happened. That can be carried out in many different ways, but usually it is the responsibility of the version control tool to do that. It keeps track of the addition of new versions to the repository and should know the status of the files in the user's workspace. Based on that information it is easy to decide whether parallel work has been performed or not. There are different strategies for when to consider parallel work to have happened. The most "relaxed" is to look at each single file and decide on a file-to-file basis. That is, however, a very unsafe strategy as changes to different files in the same commit are usually related. Most common is therefore the long transaction strategy [12] where a concurrency conflict is announced – and the commit aborted – if not all single changes can be committed. Concurrency detection is tightly connected to the unit of versioning. The more fine-grained it is, the easier it is to decide if parallel work has happened on the same artefact.

Conflict detection, on the other hand, is the discovery of an unsuccessful merge of work that has been carried out in parallel on the same artefact (unit of versioning). And conflict resolution is when the outcome of the merge is successful. Conflict detection and resolution is usually the responsibility of the merge tool. As input is gets the two alternatives and their common ancestor from which it tries to create a merge. Conflict detection and resolution is tightly connected to the unit of comparison. The more fine-grained it is, the easier it is to distinguish if the same "thing" has changed in both alternatives, in which case there is a real conflict that might be difficult for the merge tool to automatically resolve. However, the richer set of units of versioning in model merge might provide more information that could allow automated merges even in these cases of apparent conflict, as discussed above.

Both unit of versioning and unit of comparison are very dependent on the nature of what is being versioned and merged. In the traditional textual case, unit of versioning is a file and unit of comparison is a line of text. For models we will have to use the file as unit of versioning if we use traditional version control tools, but depending on the modularization mechanisms of the model language, we can have more or less flexibility in what we are allowed to put into a single file. For the unit of comparison, we are bound by the decision of the merge tool, which in turn will be highly influenced by the syntax and semantics of the model language. A supportive model merge tool should have as fine-grained unit of comparison and as rich and well-defined semantics as possible.

4.3 Presentational Issues

Presentational issues are important also for text merge, but takes on even more importance for model merge. We have to present merges (and diffs) and in particular conflicts to the user in a way that he can understand the nature of the conflict (or change) and such that irrelevant details are left out. We also need to consider how the presence of merge conflicts should be represented in the model itself.

Even in the simple case of text merge, we often have presentational issues. Merge tools are not very good at handling these and users have to aware of that and behave in a way to avoid getting conflicts that are grounded in "irrelevant" layout. A typical example is the indentation of programs that is a frequent cause of "stupid" merge conflicts until people agree on a common setup of their editor. For model merge we would like the tool to be able to ignore layout changes, as they are not our primary focus and can obscure more important changes. This does not mean that a model merge tool should always ignore layout changes, as they may indeed be important too. Just that because it is virtually impossible to avoid layout changes when working with models (as opposed to text), the merge tool has be more supportive and allow us the flexibility to ignore – or consider – layout changes in the merge.

The standard behaviour of text merge tools is to work in batch mode. The tool tries to produce merge results for all files that have to be merged and leaves conflict markers in the files where it does not manage to resolve the merge conflicts. This works well for text, as we are able to open, read and understand the resulting files in our text editor. However, that is not the case for model merge. If we would leave conflict markers in the resulting merged model, we would not able to load it into our model editor – and we would be pretty stuck. Therefore, current model merge tools work in interactive mode and ask the user to manually resolve all the conflicts one by one before the result is created. For some cases that may be a good way of working, but if we would like to leave some flexibility to the user, we should allow for the batch mode as well. This can be done if we include conflict representation into the meta-model, such that models with conflict markers become valid models for the editor. We should also be aware that, since model editors are based on the syntax of the modelling language, they cannot cope with syntactically incorrect merge results in general, so extreme care has to be taken in constructing the merge result.

5 Conclusion

The experience at Ericsson AB from using models – and in particular UML– for model-centric development has been predominantly positive. The technology is mature and we get less complexity and good performance when using models. However, the support from engineering tools is not yet mature. In particular support for collaboration like merge and diff tools.

Based on a recent tool evaluation [18], we can conclude that model merge tools have improved since our initial evaluation and that of [5]. In this paper, we have identified and discussed a number of requirements for making them even better. At the present state it looks like there is too much diversity in principles and mindset for different model merge tools. Such diversity does not exist for text merge tools – and the fact that the principles are most often implicit makes the problem even bigger. However, we see that as a natural thing at this early, immature state and hope that "state of the practice" will now start to converge.

From our analysis and discussion of merge and diff problems a number of requirements emerged that can be dealt with on the long, medium and short term by various actors:

- long term: research issues and challenges
  - semantics of model merge
  - meta-model conflict representation
  - modularization mechanisms
- medium term: research to tool transfer possibilities
  - semantics of model merge
  - presentational issues
- short term: MDD process best practices
  - use a strict long transaction model
  - merge often to avoid irresolvable conflicts
  - verify and validate each merge result
  - educate users in the model merge tool semantics since it is so much more complex and not as "uniform" as for text merge

References

1. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
2. Alanen, M., Porres, I.: Basic Operations Over Models Containing Subset and Union Properties. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 469–483. Springer, Heidelberg (2006)
3. Asklund, U.: Identifying Conflicts During Structural Merge. In: Proceedings of NWPER 1994, Nordic Workshop on Programming Environment Research, Lund, Sweden, June 1-3 (1994)
4. Babich, W.A.: Software Configuration Management – Coordination for Team Productivity. Addison-Wesley, Reading (1986)
5. Barrett, S., Chalin, P., Butler, G.: Model Merging Falls Short of Software Engineering Needs. In: [16]

6. Bendix, L., Emanuelsson, P.: Diff and Merge Support for Model Based Development. In: [9]
7. Bendix, L., Emanuelsson, P.: Collaborative Work with Software Models – Industrial Experience and Requirements. In: Proceedings of the Second International Conference on Model Based Systems Engineering – MBSE 2009, Haifa, Israel, March 2-6 (2009)
8. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A Manifesto for Model Merging. In: Proceedings of the International Workshop on Global Integrated Model Management, Shanghai, China, May 22 (2006)
9. Proceedings of the International Workshop on Comparison and Versioning of Software Models, Leipzig, Germany, May 17 (2008)
10. Proceedings of the International Workshop on Comparison and Versioning of Software Models, Vancouver, Canada, May 17 (2009)
11. Ellis, C.A., Gibbs, S.J., Rein, G.L.: Groupware – Some Issues and Experiences. Communications of the ACM (January 1991)
12. Feiler, P.H.: Configuration Management Models in Commercial Environments, Technical Report SEI-91-TR-7, Software Engineering Institute (March 1991)
13. Girschick, M.: Difference Detection and Visualization in UML Class Diagrams, Technical Report TUD-CS-2006-5, TU Darmstadt (August 2006)
14. Mehra, A., Grundy, J., Hosking, J.: A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In: Proceedings of the 20th International Conference on Automated Software Engineering, Long Beach, California, November 7-11 (2005)
15. Proceedings of the Workshop on Model-Driven Software Evolution, Amsterdam, The Netherlands, March 20 (2007)
16. Proceedings of the Second Workshop on Model-Driven Software Evolution, Athens, Greece, April 1 (2008)
17. Nguyen, T.N., Thao, C., Munson, E.V.: On Product Versioning for Hypertexts. In: Proceedings of the 12th International Workshop on Software Configuration Management, Lisbon, Portugal, September 5-6 (2005)
18. Nåls, A., Auvinen, J.: Model Merge Study, internal Ericsson Technical Report (April 2009)
19. Oliveira, H., Murta, L., Werner, C.: Odyssey-VCS: a Flexible Version Control System for UML Model Elements. In: Proceedings of the 12th International Workshop on Software Configuration Management, Lisbon, Portugal, September 5-6 (2005)
20. Selonen, P.: A Review of UML Model Comparison Approaches. In: Proceedings of Nordic Workshop on Model Driven Engineering, Ronneby, Sweden, August 27-29 (2007)
21. Störrle, H.: A formal approach to the cross-language version management of models. In: Proceedings of Nordic Workshop on Model Driven Engineering, Ronneby, Sweden, August 27-29 (2007)
22. Westfechtel, B.: Structure-Oriented Merging of Revisions of Software Documents. In: Proceedings of the 3rd International workshop on Software Configuration Management, Trondheim, Norway, June 12-14 (1991)
23. Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. In: Proceedings of the 20th International Conference on Automated Software Engineering, Long Beach, California, November 7-11 (2005)
24. Zito, A., Diskin, Z., Dingel, J.: Package Merge in UML 2: Practice vs. Theory? In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, October 1-6 (2006)