Supporting Collaborative Work by Preserving Model Meaning when Merging Graphical Models

Keith Phalp[1], Frank Grimm[2], Lai Xu[1]

[1] Bournemouth University, Fern Barrow, Poole, Dorset, BH12 5BB, UK
[2] ScopeSET Technology Deutschland GmbH, Germany

Abstract. An important aspect of support for distributed work is to enable users at different sites to work collaboratively; models need to be accessible by more than one user at a time allowing them to modify them independently from each other supporting parallel evolution [1]. As design is a largely creative process users also use layout to convey meaning. However, tools for merging such models tend to do so from a purely structural perspective, thus losing an important aspect of the meaning conveyed by the modeller. This paper presents a novel approach to model merging which allows us to preserve such layout meaning when merging. We first present evidence from an industrial study, which demonstrates how users use layout to convey specific meanings. We then introduce an approach to merging which will allow for the preservation of meaning and finally describe a prototype tool.

Keywords: UML class models/diagrams, model merging, diagram merging model-driven, distributed, software engineering

1 Introduction: The Need for Merging Models within Collaborative Development

This paper presents a novel approach to model merging [2] which is intended to bring gains to those working on collaborative software development. Whilst, in our case the primary objects (in the wider rather than software sense) are UML models, the lessons learned here have implications for collaboration more widely, where any shared artefact may be developed in a similar collaborative manner (based on diagrammatic modelling notations).

The rest of the paper is organised as follows: section two gives background to model merging and the industrial context, section three outlines our findings on the importance of layout and section four then discusses the need for a different approach to merging. Section five discusses our 'semi-automatic' approach to merging and finally section six offers some conclusions.

2 Model Merging and Context

The context for this study was the production of software for automatic gearbox controllers, using a model driven approach [3, 4]. Hence, modifying, the software was achieved by first modifying the model, then the respective implementation (source code) modifications followed automatically [5]. Modifications could only be carried out in a sequential manner: before starting to work on the model and realise their modifications, developers at one site had to wait for the developers at the other site to finish their modifications, which was clearly an inefficient form of collaboration [6]. Hence, the main motivation for the research presented here was to remove the limitation of only one user modifying a model at the same time [7] and to enable a genuinely collaborative approach. However, when evolved models are modified independently from each other, the same model elements might have been modified in different and potentially contradicting ways [8]. These 'merge conflicts' usually cannot be solved automatically by a merge tool, since such a tool cannot decide which element version to use in the merged model [8]. Hence, modellers (in our case software engineers) have to manually resolve conflicts and reason about conflicting changes [9, 10]. It is important to re-iterate that, for model-driven software development, models are not just a means of visualisation and communication since source code can be derived automatically. Hence, the need to understand how modellers interpret the models, so that we could understand fully the impact of merging, as this will directly impact the resultant software artefacts.

3 The Importance of Layout

Initial results of this analysis are presented in [11]; the results having come from examination of two substantial projects [12]. The following lists some of the ways in which we found that the software developers used layout to convey meaning (in our case for class diagrams). Notably, this, often domain-specific meaning, is neither formally defined in the model nor the diagram itself. The interested reader is referred to Grimm [12] for an exhaustive list.

- The absolute position of a class symbol was meaningless [20, 22], though the symbol’s proximity (diagram context) and relation the other class symbols was important for the modellers’ mental-map [13] of a diagram.
- Class symbols did not overlap (a fundamental requirement of readability) [14], were often ordered according to their semantics in the software design domain, and UML class diagram layout guidelines often [15] ignored. Symbols of closely related classes were then positioned in close proximity to each other; for instance in containment (whole-part) and inheritance hierarchies [16].
- Diagrams dealing with similar domain concepts, i.e., representing classes whose semantics were closely related, often exposed a similar layout structure, supporting the finding that diagram layout conveys inherent information important to modellers.

- Elements placement was based on modellers' knowledge of semantic relationships among elements and how they wanted to represent this knowledge in a diagram. For instance, sometimes two subclasses were placed on the left hand side close together, while another subclass on the same inheritance level was placed on the right apart from its semantically related variants. This concurs with Petre [16] who found that placing unrelated elements close to each other led to the misinterpretation that they were semantically related.
- The position of class symbols was more important than being able to draw connection as straight lines. So, positioning class symbols in their semantic context overweighted the connections the class had to other classes in the respective diagram [17]. However, there was no preferred direction of connections; though if a diagram depicted classes in a clear hierarchical context, then a top-down direction of connections was preferred [18].

Moody [19] argues that the layout guidelines given by the UML standard [15] are flawed in several ways, and, as the results of our diagram analysis show, those guidelines were not followed rigorously. Hence, diagram layouts can, and do, differ and are subject to the interpretation of the modellers who create or modify them.

The main generic finding is that the layout that modellers choose for a diagram is intentional and follows informal, unspecified rules. Elements (mainly class symbols) were placed in accordance with the element’s semantic (i.e., domain) meaning and the engineer's understanding of this meaning. Hence, elements that are closely related in terms of their domain semantics are likely to be positioned close together in a digram as well. Thus, adjacent diagram symbols usually reflect a close relationship of the semantic concepts and their layout in the diagram conveys this meaning visually .

4 Implications: A Different Approach to Merging

It was clear from our study that layout heuristics were being used in the construction of models and allocation of classes to models. These findings strengthened the conviction that merging was vital, but needed to take account of, or at least try to preserve, as much of the meaning that layout conveyed as possible.

However, having conducted a thorough analysis of existing automatic diagram layout approaches (typically based on automatic graph layout algorithms) it became clear that these did not meet our needs because they merely preserved the connections (in a topological sense) rather than dealing with the layout itself, and, similarly ignored many of the heuristics suggested above [20].

In addition, for UML, automatic layout algorithms are based on UML model elements, i.e., the semantic elements like packages, classes, and inheritance and association relations among classes [21]. Since automatic layout algorithms focus on creating aesthetically pleasing layouts, they try to optimise diagram layouts with respect to edge crossings and bends [22], but they do not take the mental map of a diagram into account. When symbols are added to or deleted from a diagram, an automatic layout approach might create a completely different layout. Hence, users working with the diagram would have to re-learn the diagram.

Given those issues related to conventional layout algorithms, the challenge was how to enable efficient model and diagram merging whilst still allowing modellers to preserve the domain-specific information. Ideally, a diagram merge approach would automatically merge diagrams in a meaningful way and burden users only with solving “real” diagram merge conflicts. The ideal scenario would be to allow modellers to create diagrams the way they want with all possible layout freedom, but still be able to rely on mental-map-preserving automatic layout. These two objectives of course contradict each other – layout freedom and automatic layout cannot be combined without one limiting the other.

The authors suggest that a certain degree of automatic layout is desirable, for creating diagrams in the first place and for merging them. When model elements depicted by diagram symbols are updated, a modelling tool has two possibilities, (1) update the diagram symbols' graphical properties (including its size) or (2) leave them as they are and let the modellers take care of manually updating the diagrams. Given the above drawbacks of fully automatic diagram layout, but also given that automatic layout is useful to some extent, and given that merging fully manual diagram layout in a meaningful way is not possible, a semi-automatic layout is described briefly in the following section.

5 Implementing a Merging Tool

Since a diagram can be independently modified by different modellers, in parallel, the diagrams should ideally be combined without user interaction if there are no diagram merge conflicts (and the resulting diagram layout should still be meaningful). Therefore, a semi-automatic layout approach is presented which allows modellers to make the grouping and ordering of class symbols explicit.

As discussed above, these two layout features were found to be most important with respect to defining and conveying domain-specific meaning; thus, when modellers create diagrams, they can explicitly define the order of class symbols. In our approach this is the only layout information that can be defined manually. The more layout features modellers can influence, the more diagram merge conflicts can occur because the features were conflictingly changed in parallel in both evolved diagrams. Those conflicts then have to be resolved manually. This additional diagram information is then taken into account when class diagrams are laid out automatically. The extra information is leveraged in order to position class symbols according to the manually defined order. Thus, for example, modellers are able to explicitly define the principal horizontal and vertical ordering of class symbols – which are then automatically laid out as trees in a top-down manner. Being able to automatically re-arrange diagram symbols during the diagram merge process relieves modellers from having to deal with unimportant layout merge conflicts (e. g., symbol overlapping) and allows them to automatically create uncluttered diagrams during the merge.

Fig. 1 shows a merge example. The screen-shot shows four UML class diagrams: the initially merged diagram is shown in the upper-right corner, both evolved diagrams are in the lower half, and their common ancestor diagram is shown in the upper-left corner. The latter three diagrams are immutable, only the merged diagram and its underlying model can be modified by the modeller. Modifications are necessary to resolve merge conflicts. Both evolved diagram versions and the ancestor diagram are annotated with change and conflict information. Diagram symbols and model elements deleted in one or both evolved diagrams are highlighted and annotated in the ancestor version - since they are not part of evolved diagram (in which they got deleted). Any other changes are highlighted and annotated in the evolved diagrams. Conflicting changes are highlighted in a different colour to non-conflictingly ones.

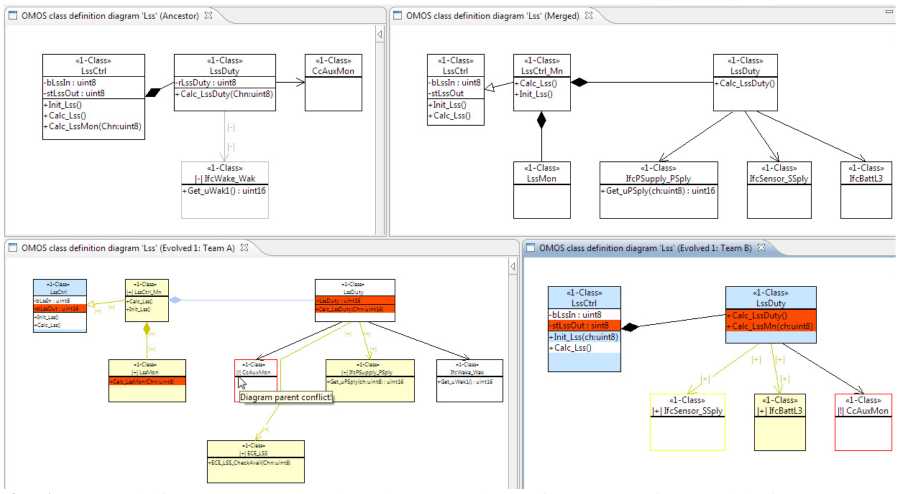

Fig. 1: Merged diagram example (also shown: evolved diagram versions and their common ancestor with change and conflict annotations)

A brief description of our algorithm now follows (again see Grimm [12] for a more detailed treatment). As a first step, the changes between both evolved models and the common ancestor are calculated by comparing the states of equivalent model elements. Equivalent elements in different model versions are determined by means of globally unique identifiers and it is then decided, for each change, whether or not it can be accepted. Conflicting changes are rejected. For model elements with conflicting containments this means that the model element is not part of the initially merged model. Then, so-called existence conflicts exist, and the modeller has to manually decide which parent element contains the element. If an element is not included in the initial merged model, its children elements are also omitted. Referencing any such element from other elements is not possible. Thus, such references are also marked with merge conflicts. As a second step, the actual merged model is created. Any model element which does not have an existence conflict becomes part of the merged model. Of course, these model elements might have merge conflicts, too. However, these conflicts do not prevent the element from becoming part of the merged model, though they would need to be resolved manually by the modeller.

The merge tooling also provided modellers with the possibility to resolve merge conflicts by accepting and rejecting model and diagram changes. Not only could modellers modify the merged model (diagram) by means of accepting or rejecting changes, but also they could also modify it in any way. Therefore, even model elements or symbols which were not changed at all (not even non-conflictingly) could be modified. Hence, the editing capabilities of the implemented model merge tool were those specific ones required for dealing with changes, in addition to the common editing functions provided by an ordinary modelling tool and used when models and diagrams are created in the first place. The dedicated merge tooling took care of updating the acceptance status of changes when the merged model or diagram was updated – so that modellers could learn whether a change made in one model (diagram) was (still) part of the merged model.

In contrast to other automatic UML class diagram layout approaches, no layout heuristics or iterative layout were applied for the implemented layout approach. Such approaches are used to create more aesthetically pleasing and potentially more readable diagram layouts, but they have the drawback that the resulting layout might 'look' different every time a diagram is laid out and when the model is updated (and thus the information used to calculate the layout changed). Hence, the semi-automatic layout approach implemented here is a trade-off between diagram mergablity and manually creating UML class diagrams with all the freedom with respect to positioning / laying out of diagram symbols.

Hence, in our approach, the 'freedom' of manual layout was reduced in favour of being able to efficiently merge class diagrams, while the most important layout features (regarding embedding domain-specific information into the layouts of class diagrams) can still be defined manually by modellers. That is, the layout approach implemented here has as a priority keeping a stable and predictable layout. This means that the order of class symbols is not altered so long as the modeller does not change it. The layout of connection symbols depicting relationships among classes is done completely automatically; a connection symbol's layout is not changed as long as the order of the connection’s class symbols does not change.

6 Conclusions

This paper examines support for collaboration across multiple sites when developing automotive software, focussing on the issue and importance of model merging.

In order to understand the way developers used layout we examined two substantial industrial projects (see section four). The main generic finding was that modellers use layout to convey meaning, often in a way that is not defined by given model heuristics (such as guidance on the production of UML class diagrams). Having established the importance of layout we then wanted to enable modellers to work independently on certain models in parallel.

Therefore, we present an approach for laying out models (class diagrams) in a semi-automatic fashion that allows modellers to manually define the order of class symbols and at the same time allows diagrams to be merge-able. This approach provided a trade-off between (1) the amount of layout freedom modellers had regarding the position of diagram symbols and (2) the ability to automatically create 'meaningful' merged diagrams whose layout was untangled - and preserved the manually defined class symbol hierarchy. In addition, an approach to visualising differences and conflicts between 'to-be-merged' UML models and class diagrams was implemented. This allowed the developers to work with merged models in the same way that modellers work with them when they create them in the first place, and crucially allowed developers to exchange partially merged models.

In summary, this paper has provided evidence for the importance of layout in models and has presented a 'semi-automatic' approach to merging which allows modellers to retain a greater recognition and understanding of their work when models across sites are merged. In addition, by allowing the exchange of partially merged models conflicts between versions can be resolved effectively. We contend that such merging is a vital cog in the support for collaborative development processes.

References

1. Mens, T., Buckley,J. , Zenger, M., Rashid, A.: Towards a taxonomy of software evolution. In:. Proceedings of the Workshop on Unanticipated Software Evolution (2003)
2. Westfechtel, B.: Structure-oriented merging of revisions of software documents. In: Proceedings of the 3rd international workshop on Software configuration management, pages 68–79, New York, NY, USA, ACM Press (1991)
3. Hermsen, W., Neumann, K.-J.: Application of the object-oriented modeling concept OMOS for signal conditioning of vehicle control units. Technical report, SAE 2000 World Congress, March 2000, Detroit, MI, USA (2000)
4. Schweizer, M., Benkel, M.: Development of product families - an example from the automobile industry. In: Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3) (2005)
5. Kleppe, A. G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (2003)
6. Harrison, W. H., Ossher, H., Sweeney, P. F.: Coordinating concurrent development. In Proceedings of the 1990 ACM conference on Computer-supported cooperative work, CSCW '90, pages 157–168, New York, NY, USA, ACM (1990)
7. Mens. T.: A state-of-the-art survey on software merging. In: IEEE Trans. Softw. Eng., 28(5):449–462 (2002)
8. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv., 30:232–282, (June 1998)
9. Ohst, D., Welle, M., Kelter, U.: Difference tools for analysis and design documents. In: ICSM '03: Proceedings of the International Conference on Software Maintenance, page 13, Washington, DC, USA, IEEE Computer Society (2003)
10. Kelter, U., Wehren, J., Niere, J.: A generic difference algorithm for uml models. In: Proceedings of the SE 2005, Essen, Germany (March 2005)
11. Grimm, F. Phalp, K, Vincent, J.: Enabling multi-stakeholder cooperative modelling in automotive software development and implications for model driven software development. Ist International Workshop on Business Support and MDA (MDABIZ) a Tools 2008 Workshop, Zurich (July 2008)

12. Grimm, F. Enabling collaborative modelling for a multi-site model-driven software development approach for electronic control units, PhD thesis, Bournemouth University (2012)
13. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. In: Journal of Visual Languages and Computing, 6(2):183–210 (1995)
14. Tamassia, R., Di Battista, G., Batini, C.: Automatic graph drawing and readability of diagrams. In: IEEE Trans. Syst. Man Cybern., 18(1):61–79 (1988)
15. UML Notation Guide. Object Management Group (2003)
16. Petre. M.: Why looking isn't always seeing: readership skills and graphical programming. In: Commun. ACM, 38(6):33–44 (1995)
17. Eichelberger, H.: Nice class diagrams admit good design? In: SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization, pages 159–167, New York, NY, USA, ACM Press (2003)
18. Purchase, H.: Evaluating graph drawing aesthetics: defining and exploring a new empirical research area. In: DiMarco, J. (ed.), Computer Graphics and Multimedia: Applications, Problems and Solutions, pages 145–178. Ed. Idea Group Publishing (2004)
19. Moody, D. L.: The "physics" of notations: a scientific approach to designing visual notations in software engineering. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pages 485–486 (2010)
20. Eichelberger, H.: Aesthetics of class diagrams. In: Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis, pages 23–31. IEEE (2002)
21. Eiglsperger, M., Gutwenger, C., Kaufmann, M., Kupke, J., Jünger, M., Leipert, S., Klein, K., Mutzel, P., Siebenhaller, M.: Automatic layout of UML class diagrams in orthogonal style. In: Information Visualization, 3(3):189–208 (2004)
22. Eichelberger. H.: On class diagrams, crossings and metrics. In: Jünger, M., Kobourov, S., Mutzel, P. (eds.), Graph Drawing, Dagstuhl Seminar Proceedings (2006)