Enabling multi-stakeholder cooperative modelling in automotive software development and implications for model driven software development

Frank Grimm[1,2], Keith Phalp[1], Jonathan Vincent[1], Georg Beier[2]

[1] Software Systems Research Centre, Bournemouth University, UK
[2] Generative Software Engineering Zwickau, University of Applied Sciences Zwickau, Germany

Summary. One of the motivations for a model driven approach to software development is to increase the involvement for a range of stakeholders in the requirements phases. This inevitably leads to a greater diversity of roles being involved in the production of models, and one of the issues with such diversity is that of providing models which are both accessible and appropriate for the phenomena being modelled. Indeed, such accessibility issues are a clear focus of this workshop.

However, a related issue when producing models across multiple parties, often at different sites, or even different organisations is the management of such model artefacts. In particular, different parties may wish to experiment with model choices. For example, this idea of prototyping processes by experimenting with variants of models is one which has been used for many years by business process modellers, in order to highlight the impact of change, and thus improve alignment of process and supporting software specifications. The problem often occurs when such variants needed to be merged, for example, to be used within a shared repository.

This papers reports upon experiences and findings of this merging problem as evaluated at Bosch Automotive. At Bosch we have different sites where modellers will make changes to shared models, and these models will subsequently require merging into a common repository. Currently, this work has concentrated on one type of diagram, the class diagram. However, it seems clear that the issue of how best to merge models where collaborative multi-party working takes places is one which has a significant potential impact upon the entire model driven process, and, given the diversity of stakeholders, could be particularly problematic for the requirements phase. In fact, class diagrams can also be used for information or data models created in the system analysis step. Hence, we believe that the lessons learned from this work will be valuable in tackling the realities of a commercially viable model driven process.

1 Introduction

This study examined class diagrams within industrial projects, involving multiple stakeholders located at different sites, and used to model the static structure of software applied in cars. The authors have analysed models of seven projects, all of which belong to the automotive domain. In these projects, models are created in a cooperative development style. The size of the models varied from 100 to more than 1000 classes. Each diagram within the models contained between 10 and 20 classes. The models are created using the benefits of visual representations, i.e., using diagrams. Each project consists of 20 to 120 class diagrams. These diagrams were created manually by two development teams. Each team consists of about 15 software engineers. These two teams are located in two different countries.

2 Observations about diagrams

2.1 Model organisation and separation into diagrams

As mentioned above, each of the seven analysed projects consists of a considerable number of class diagrams, varying from 20 for the smallest project to 120 for the largest project. Hence, class models, represented by the diagrams of a certain project, were separated into different 'subordinate' diagrams. There were two different categories of class diagrams. The first category of diagrams is used to show the structure and hierarchy of UML packages of the software system under development. Depending on the number of packages, one to three diagrams were used to define the package structure. Each *package overview diagram* shows top-level packages and, if they exist, sub-packages. In addition, dependencies between packages are defined using *dependency relationships*. Package overview diagrams did not show any other elements than packages.

The second category of diagrams are detailed class diagrams. A *detailed diagram* defines the details of a number of classes, i.e., a class's attributes and operations and, using aggregation and associations, its relationships towards other classes are defined. Each detailed diagram consists of one to four main classes whose details are defined. According to the class documentation, theses classes are strongly related in terms of their semantics. The main classes shown in the same diagram either belong to the same inheritance hierarchy or belong to the same whole-part hierarchy. Using either inheritance or aggregation relationships the hierarchies are defined in the diagram as well. When inheritance hierarchies were shown in the diagram, the complete hierarchy, as seen from the sub-class that is one of the main classes in the diagram, was shown. Hence super classes whose details were defined in other detailed diagrams were shown as well. Branches of the inheritance tree that did not belong to the direct inheritance chain of a main class in the diagram were omitted. Operations and attributes of base classes defined in other detail diagrams were shown as well, no matter if they were accessible by the sub-classes or not.

Each main class is shown in its containing package. An observation supporting the close relation between main classes is that all main classes either belong to the same packages or a number of main classes belong to a certain package and the other main classes belong to one or more sub-packages. Hence, the main classes of a certain detailed diagram belong to—in terms of containment hierarchy—closely related packages.

In addition to inheritance and whole-part hierarchies, association relationships are defined for the main classes. The associated classes are shown in the diagram. Their operation and attributes are shown as well, no matter if they are publicly accessible or not. Uni-directional associations are always navigable from a main class which is defined in the details diagram to the associated class which is defined in another detail diagram. When bi-directional associations are shown, they are shown in both details diagrams defining the main classes involved in the association. Associated classes that are not main classes of the diagram at hand were not shown in their packages. Classes are only shown inside their package in the details diagram defining the class.

Based on the above observations about detailed diagrams it can be argued that, in order to create a large class model, software engineers use a large number of class diagrams and focus on defining a small number of classes and all their internals in each diagram. Models are separated into diagrams in a top-down manner, from package diagrams and diagrams representing the top-level classes to classes representing very detailed parts of the system.

Each diagram deals with a certain aspect of the model. This aspect is reflected by the name of the diagram. For instance, the main classes `DriveProgramme` and `DriveSituation` are defined in the diagram "Drive Situation Detection". Details of the newly defined part-classes (*i.e.*, aggregated classes) of the main classes in this diagram were defined in another diagram called "Gear selection." By looking a the names of the diagrams, software engineers already know what information the diagrams include, *i.e.,* it is obvious on which classes a certain diagrams focuses and that additional information like super classes, aggregated and associated classes are shown in the diagram as well. Thus, navigating the large diagram repository becomes possible. As the observation in this section shows, their thoughtful organisation and structured nature, makes diagrams first-class citizens of the project under development. That is, they mean far more to the project stakeholders than just documentation.

2.2 Layout of class detail diagrams

In addition to the clear hierarchy of diagrams that is used to organise a large model into comprehensible parts, each diagram itself show as clear layout that focuses around its main classes. The main classes are shown in the centre of the diagram. Super classes, associated and aggregated classes that are not main classes of this diagram are positioned around the main classes. Inheritance hierarchies are usually shown in vertical order—super classes above sub-classes. There is no such layout rule for association and aggregation relationships. Associated and aggregated classes are positioned in any direction around a main class. Though it is usually preferred to position associated and aggregated classes below or besides a main class, it is also often the cases that they are positioned above a main class. Because of alignment and the use of gaps it is possible to distinguish between the main classes which are defined in the diagram and the classes that are related to these classes, but defined in other diagrams. The layout possibilities of the visual language of class diagrams provide a secondary notation, in terms of flow, hierarchy, ordering and alignment for users.

2.3 Diagram layout and semantics

In addition to applying aesthetic criteria for laying out diagrams, the semantics of the depicted elements play an important role when elements are positioned in diagrams. Not only the decision of how to separate a model into diagrams, but also the way elements are laid out in a diagram reflects that a class's meaning in the domain is important for its position in a diagram. Semantically related classes were positioned close to each other in the diagrams. For instance, a class called `Battery` was positioned next to class `BatteryManager` in the diagrams containing both classes, or classes belonging to the same semantic aspect of the system are positioned in close proximity to each in a kind of class cluster. The absolute position of these elements does not seem to as important as the fact that related elements are positioned in close proximity to each other. For example, the classes `Battery` and `BatteryManager` are sometimes located on the left side of a diagram, sometimes at the bottom. But both classes are always positioned next to each other.

The authors observed that users are, to a certain degree, willing to sacrifice aesthetic rules for the semantic integrity of classes. For instance, lines depicting relationships were longer than necessary if an associated class would have been located outside of its semantic cluster.

As described above, associations of main class of a diagram to other classes are defined in the class's details diagram. But, when associations between classes that are not in the focus of a diagram, *i.e.,* are not main classes of this diagram, are important for the main classes of this diagrams, these associations are shown in this diagram as well. Hence, as class and inheritance hierarchies, associations can appear in more than one diagram.

2.4 Evolving models and diagrams

The authors analysed four to seven different versions of the seven projects. These versions show how the projects evolved over the course of four to six months. When a class is added to the model, it is added in the details diagram that contains classes related to the new class. When a new aspect is introduced into the model or an existing aspect is split into several sub-aspects, and new diagrams are created. If existing classes belong to the aspect of a newly created diagram, they are moved into this new diagram.

In addition to defining the new class in its detailed diagram, the class is added to all diagrams that contain classes which associate the new class or contain a inheritance hierarchy to which this class belongs. When a newly added class is added to a diagram where it is associated by one or more main classes, it is positioned in the semantic cluster containing the class to which this class is semantically related.

When a new package is added, it is added to the detail diagrams containing the package's classes and to the package overview diagram.

3 Enabling co-operative modelling

As the observations presented in Section 2 demonstrate, class diagrams play an important role for model-driven software development. Unfortunately, when it comes to co-operative development, the advantages of the visual language for class diagrams in terms of comprehensibility by human readers turn into disadvantages. The problem with this visual language is that the UML does not standardise how a diagram should be laid out and that there exist only few informal conventions about the layout of class diagrams. For instance, super classes are drawn above their sub-classes. Hence, users are completely free when arranging the layout of a class diagram.

Two different approaches to co-operative development exist. The *pessimistic* approach grands one user exclusive access to a certain resource by locking this resource in a shared repository. After the resource has been locked, it can be modified by the user. The optimistic approach is not based on locking, instead access to a certain resource is granted to any user. But instead of modifying a resource directly in the shared repository, a local copy is created which can then be modified. In order to make modifications available to all users, the modified resource has to be transferred back into the repository. But since every user can access a certain resource simultaneously, this resource might have been modified by more than one user at the same time. Thus, it might become necessary for the user to deal with, possibly conflicting, modifications made by other users.

The authors are currently developing a tool for the co-operative development of class models and diagrams using the optimistic approach. Hence, models can be modified by more than one user at a given time. When a modified model is transferred back into the shared repository, it is likely that other users have submitted modifications to the repository before. Therefore, a model *merging process* is necessary. It is discussed in the next paragraph.

3.1 Related work

During the merge process two modified model versions are compared to their common ancestor version. As discussed in Section 2, for the industrial case presented in this paper, class diagrams are the one and only way to create class models. Hence, the merge process has to focus on these diagrams. When two versions of a diagram are to be merged, it has to be considered that each version might have been modified independently from the other version. Since the two diagram versions to be merged originate from a common ancestor version, it is likely that some elements are contained in both current diagram versions. But it is also very likely that the two current diagram versions to be merged were modified in different ways and their layouts and contents differ to a certain degree.

The authors know of two existing academic approaches to class diagram merging and two industrial approaches. The industrial CASE tools that take the graphical representation of models into account are IBM's Rational Software Architect 6 (RSA) and No Magic's MagicDraw UML (Teamwork Server Edition). Changes made to diagram elements are highlighted in the original diagram. Diagram merging is not done on a graphical level; even worse, all changes of diagram elements have to be handled by the user in a textual form. Considering the advantages of diagrams as discussed in Section 2, it follows that tools should allow for the merging of such models based on their graphical representation.

One academic approach [5] uses automatic graph layout to avoid the problem of layout merging. Hence, the original, manually created diagram layout is not taken into account and layout changes are not considered. A prototype tool was created for the other academic approach [4]. This approach does not use automatic diagram layout. The layout of one of the compared versions is used as the main diagram version, the elements of other diagram version are included in the main diagram version as overlays. Although preserving the original layout of one version, this approach has a major drawback as well: the user has to accept or reject every individual layout change, additionally elements may even overlap.

Laying out class diagrams fully automatically is not always desirable because the generated layout can differ significantly from the original, manually created, layout of the observed diagrams. Unfortunately, as discussed by Eiglsperger *et al.* [2], it is inherent to fully automatic layout algorithms that they produce good results only when exactly one hierarchy criterion, *e.g.*, inheritance or whole-part relationships, is used. The reason why the original diagram layouts differ so much from the automatically generated ones is that in most of the original diagrams more than one hierarchy criterion was used. Therefore, the result of automatic layout algorithms differs significantly from the original. The original diagram layout conforms to the user's *mental map* [1] of the elements presented in this diagram. Therefore, the need to preserve this mental map is as important as the need for graphical merging in the first place. Given the shortcomings of both the automatic and the manual approaches, the authors advocate a semiautomatic approach for graphical model merging which is described in the following sections.

3.2 Merging two versions of a diagram

The following constraints have to be taken into account when developing a merge approach for class diagrams: (1) diagrams are main view on the underlying model, (2) these diagram are created manually, (3) their layout is intentional and should be preserved as much as possible, and (4) they cannot be merged fully automatically, but (5) the effort required during the merge process should be kept at a minimum.

In order to merge two versions of a diagram, one diagram version, the *base diagram*, is used as a starting point. All elements that were added in the other diagram version, the *fitted diagram*, are included in this diagram as well. The result is a *pre-merged diagram* that shows the elements of in both diagram versions.

Since the layouts of the diagrams are likely to differ, elements added in the fitted diagram cannot be positioned in the pre-merged diagram using their original coordinates. In another publication by the authors [3] it is described how elements added in the fitted diagram version are positioned in the pre-merged diagram. The essential concept used for the positioning is an element's *neighbourhood*. This means when an element from the fitted diagram is positioned in the pre-merged diagram, it is tried to position it next to its original neighbour elements. Hence, the neighbourhood of elements is preserved as much as possible and so is the mental map of the diagram.

The approach has two advantages. Firstly, the manually created layout of the diagrams is preserved since no fully automatic layout algorithm is applied. Secondly, the pre-merged diagram is presented to the users in the same way as it will finally look like *after* it has been merged. No overlay mechanism is used which, at the cost of time and effort, requires the user to rearrange to diagram elements in order to eliminate overlapping elements.

3.3 Visualising and handling of differences and conflicts

The following differences can appear when comparing a current version of a diagram and its direct ancestor version: (1) new elements can be added, (2) elements that existed in the common ancestor diagram version can be removed, (3) elements can be moved, and (4) elements can be re-sized.

As explained in 3.2, elements are positioned according to their relationships towards other elements in their neighbourhood. Therefore, differences between added, moved and re-sized elements in both diagram versions do not automatically result in merge conflicts. With the proposed neighbourhood-based approach when elements are positioned in the pre-merged diagram, they are positioned according to their neighbourhood, and not according to their absolute position in the original diagram. Hence, no conflicts can arise when the absolute position of each version of an element differs in the two diagram versions to be merged. Conflicts only appear when an element is positioned in totally different neighbourhoods in both versions, *i.e.,* when the mental map of the element is very different. In such circumstances this conflict is communicated to the user by annotating the element in the pre-merged diagram. The user can then show the element in its alternative neighbourhood as given in the other original diagram (the first element version in the pre-merged diagram was positioned according to its neighbourhood in the base diagram). Again, the neighbourhood-based approach is applied to preserve the element's graphical context and the mental map of the diagram. The user can then chose between the alternative element positions. This choice is then reflected in the merged diagram as the element is positioned as chosen by the user.

The above description makes it apparent that three features are required for the pre-merged diagram. Firstly, its elements have to be annotated to present difference and conflict information to the user. Secondly, the user can, based on the the difference and conflict annotations, show an alternative version of a modified element. For example, when an element was moved to another parent element, *e.g.,* a class was moved to another package, the element can be shown as a sub-element of the old parent. Deleted elements, too, can be shown in the pre-merged diagram in order to allow the user to undo this deletion. Again, deleted elements are positioned in the pre-merged diagram according to their original neighbourhood. And thirdly, the layout of the pre-merged diagram has to be modified dynamically according to these user choices.

The merge process is completed when all conflicts are resolved by the user. Then a new diagram is created that contains, according to the differences accepted and rejected by the user, some or all elements of both original diagram versions.

4 Conclusions

This paper has presented reasons why mechanisms for merging diagrams (such as those commonly used within CIM & PIM phases of MDA) is vital to supporting collaborative modelling activity, particularly over distributed sites. A detailed study of diagrams produced within automotive software engineering industries reveals that modellers use layout (and related rules) in meaningful ways, and thus, that existing approaches, both manual and automated, to such merging, still have significant drawbacks. Hence, a method is presented which attempts to preserve a greater degree of meaning whilst supporting diagram merging. The authors suggest that in order to further the adoption of model driven approaches, where even greater diversity of users exists, and often across both sites and organisations, such considerations will be vital. Furthermore, that support for meaning preservation via a sympathetic merging process will further the MDA goals of reducing the gap between business needs and supporting software systems.

References

1. P. Eades, W. Lai, K. Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics '91*, pages 24–33, 1991.
2. Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. Automatic layout of uml class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004.

3. Frank Grimm, Keith Phalp, Jonathan Vincent, and Georg Beier. Towards semi-automatic, mental map preserving visual merging of UML class models. In *Proceedings of the IADIS International Conference Applied Computing 2007*, 2007.
4. Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2005. ACM Press.
5. Dirk Ohst, Michael Welle, and Udo Kelter. Difference tools for analysis and design documents. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.