Towards a Generic Layout Composition Framework for Domain Specific Models

Jendrik Johannes[*]
Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
jendrik.johannes@tu-dresden.de

Karsten Gaul
Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
karsten.gaul@gmx.net

ABSTRACT
Domain Specific Models with graphical syntax play a big role in Model-Driven Software Development, as do model composition tools. Those tools however, often ignore or destroy layout information which is vital for graphical models. We believe that one reason for the insufficient support for layout information in model composition tools is the lack of generic solutions that are easy to adapt for new graphical modelling languages. Therefore, this paper proposes a language-independent framework for layout preservation and composition as an extension to existing model composition frameworks. We describe the single components of the framework and evaluate it in combination with the Reuseware Composition Framework for layout compositions in two different industrial used languages. We discuss the results of this evaluation and the next steps to be taken.

1. INTRODUCTION
In Model-Driven Software Development (MDSD) different graphical Domain Specific Models defined in different Domain Specific Modelling Languages (DSMLs) are used in combination. MDSD approaches promise high flexibility with regard to the DSMLs that are used and how these are combined. Using metamodelling tools, developers can create new DSMLs when required and integrate them into their MDSD process by defining model transformations and compositions. Different technologies are available for model transformation and composition which are language independent. That is, they can be used with any DSML that is defined by a metamodel they understand.

A drawback of such language-independent approaches is that they handle the semantic models, but rarely support the preservation and composition of layout information. This, however, is an important issue, because the outcome of a model composition is seldom the final system which is (like a compiled piece of code) processed by machines, but another model to be viewed and edited by developers. Thus, we argue that layout preservation and composition is crucial for the acceptance of MDSD. Currently, however, approaches that are easy to adapt for new DSMLs are missing.

This paper proposes such an approach for compositions of models within arbitrary graphical DSMLs (Section 2). It provides an implementation of the approach in an extensible framework that is based on the Eclipse Modeling Framework (EMF) [18]. Our framework can be used with arbitrary graphical DSMLs defined in EMF's metalanguage Ecore [18]. It can be connected to arbitrary EMF-based model composition engines that fulfill a number of properties we will discuss. One example of such an engine is the Reuseware Composition Framework [8] with which we evaluated our approach. We performed an evaluation of our framework with two different DSMLs (Section 3) used in industry. Afterwards, in Section 4, we discuss lessons learned and future extensions to broaden the scope of our framework. We look at related work in Section 5 and conclude in Section 6.

[*]This research has been co-funded by the European Commission in the 6th Framework Programme project MODELPLEX contract no. 034081 (`www.modelplex.org`).

2. LAYOUT COMPOSITION
In this section we introduce the concepts behind our framework and, based on that, the different components of it. First (Section 2.1), we specify the scope of our work by defining criteria for the DSMLs and the model composition frameworks we support. Second (Section 2.2), we introduce the *Mental Map* concept on which we base our approach. Third (Section 2.3), we describe the different steps of our layout preservation and composition process and show variability within the different steps which can be implemented in individual components in our framework. Fourth (Section 2.4), we introduce the components we implemented.

2.1 Criteria for Supported DSMLs and Model Composition Frameworks
A DSML has to fulfill the following properties to work with our approach:
Requirement 1 The DSML has to have a graphical (diagrammatical) syntax.[1]
Requirement 2 The DSML has to be defined in Ecore.[2]

The following is required of a model composition framework to interoperate with our layout composition framework.
Requirement 3 The composition scripts for models have to have a graphical (diagrammatical) syntax.[1]
Requirement 4 The composition framework needs to be able to expose which item in a composition script refers to which input model.[1]

[1]Section 4 discusses how these restrictions can be loosened
[2]This restriction applies if our implementation is reused directly. Conceptually, our framework can be ported to another modelling environment.

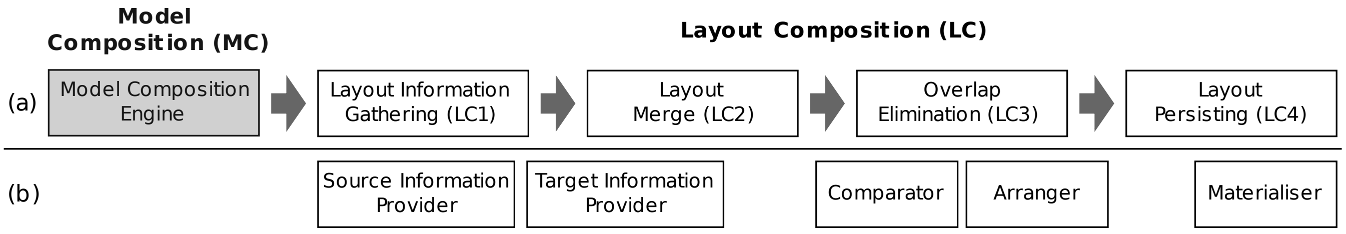

Figure 2: (a) Model and layout composition process and (b) Layout composition components of our framework

2.2 Mental Map

Naturally, when different diagrams are composed some adjustment of the layout is required because in many cases nodes of the former separated diagrams will overlap in the composed diagram. A naive solution would be to perform a complete relayout of the diagram using a layouting algorithm such as planarity [17] as shown in Figure 1.

This however, destroys the original neighborhood relationships between nodes. The literature calls these relationships the user's *Mental Map* [4] of the diagram. The importance of the Mental Map in MDSD is also stressed in [20]. One can think of the Mental Map as a road map, where the scale might vary, but the relations between elements do not. A user subconsciously creates his Mental Map of a diagram when arranging the icons in a certain way. Thus, when the layout is adjusted to eliminate overlaps the Mental Map should be preserved. There are three rules to be met in order to preserve the Mental Map [4]:

Goal 1: disjointness of nodes
Goal 2: keep the neighborhood relationship of the nodes
Goal 3: compact design

Naively applying layouting algorithms often violates one or more of these goals. In Figure 1, for example, the neighborhood relationship is not kept and, therefore, the result leaves the user disoriented.

2.3 Layout Composition Process

Figure 2a illustrates the model and layout composition process. The input to the process consists of one or more graphical models and one graphical composition script (Figure 3a). In the first step, the model composition engine—in our case Reuseware—interprets the composition script to perform the semantic model composition (MC). After that, our framework performs the layout composition (LC) with adjustment in four major steps. First (LC1), it collects the layout information from the input models and the composition script. Second (LC2), this information is merged in a Mental Map preserving fashion. For this, our framework needs to gather information from the underlying modelling and model composition frameworks (Requirement 4). Third (LC3), the merged layout data has to be adjusted to remove overlaps in a way that preserves the Mental Map (Goals 1–3). Fourth (LC4), the adjusted layout information has to be connected to the composed model, which again requires access to the underlying modelling technology.

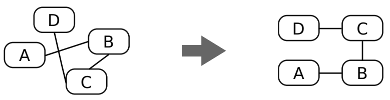

Figure 1: An application of the planarity algorithm that destroys the developer's Mental Map

In the first layout composition step (LC1) we collect all layout information. The collected information consists of (1) the layout information of each input diagram, (2) the layout information of the composition script and (3) the relation between nodes in the composition script and the input diagrams (Requirement 4).

The merging process (LC2) is steered by the layout of the composition script. The developer expects the composed model to be laid out according to his Mental Map of the composition script (cf. Figures 3a and 3b). Thus, using the information about how the nodes of the composition script relate to input diagrams, we move all the nodes of each single input diagram in correspondence to the node representing that diagram in the composition script. This is illustrated in Figure 3b where the element sets are arranged according to the composition script in Figure 3a. Because all nodes of one diagram are moved with the same vector, the Mental Map of the individual diagrams is preserved. Therefore, Goal 2 is reached. Since the positioning is based on the composition script, Goal 3 is also reached. The composed diagram however, may contain overlaps since the nodes representing models in the composition script are much smaller than the models themselves, which violates Goal 1.

To meet Goal 1, layout adjustment is performed in the next step (LC3). Here, we can make use of existing layout algorithms, where we treat all nodes that belong to one input model as a whole rather than adjusting each node individually (cf. adjustment from Figure 3b to Figure 3c). This is similar to the scaling of node clusters presented in [20].

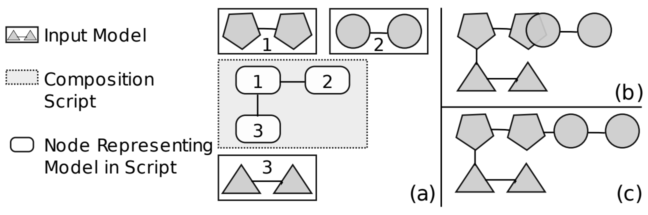

Figure 3: (a) Input of a model composition: 3 input models and 1 composition script (b) Composition result without overlay elimination (c) Composition result after the application of Horizontal Sorting

While a number of algorithms could be used (e.g., the ones discussed in [4]) we implement *Horizontal Sorting* [11] and *Uniform Scaling* [4] so far. Horizontal Sorting, as its name implies, starts at the left side of the diagram and moves overlapping fragments in x direction until they do not overlap anymore (Figure 3c). Uniform Scaling is based on the following equation: *(a+s*(x-a), b+s*(y-b))*. The point *(a,b)* is the center point and *(x,y)* is the location of the model element set that needs to be moved. The factor *s* is a scale factor used to define the distance the model elements are moved by. For automatisation purposes, *(a,b)* should not be chosen by the user—it can be computed from the merged diagram before adjustment (outcome of LC2, cf. Figure 3b).

In the last step (LC4) the computed layout information has to be materialised in the diagram of the composed model. Here, access for modifying this diagram has to be provided by the modelling technology that was used.

2.4 Framework Components

The previous section described the steps our layout composition framework performs. These steps can be implemented in individual components, which could be exchanged depending on specific demands of one composition. The different components are illustrated in Figure 2b. In the following, we give details of the functionality of these components and present what we have implemented so far.

Different component combinations can be used to achieve different results. We summarize possible combinations at the end of this section. In Section 3 we then evaluate what the benefits and drawbacks of certain combinations are.

2.4.1 Source Information Provider (LC1)

An input model consists of an arbitrary number of nodes and, for a user friendly layout algorithm that obeys the rules of the Mental Map, we have to know about the width and height of these nodes. More precisely, width and height of the bounding box of the whole input model is needed (x and y values are not important here). An *Information Provider* walks through the diagram structure and gathers the required data. The *Source Information Provider* depends on the layout format used for the input diagrams.

We implemented two *Source Information Providers* for two layout formats commonly used in EMF, which are the GMF Notation Model [7] and the TOPCASED Diagram Interchange format [19]. The GMF Notation Model is widely spread, because it is used by all DSMLs created with the GMF—a generative DSML development framework. TOPCASED is an alternative framework with similar functionality which currently offers a set of high quality UML editors. There are efforts to align both frameworks to obtain a common layout format in the EMF (possibly aligned with an upgraded version of the Diagram Interchange OMG standard [15]). In general, the Information Providers implemented by us already cover a huge amount of diagram syntaxes used in EMF. Our experience showed that an Information Provider can be implemented within hours.

2.4.2 Target Information Provider (LC1 and LC2)

Another *Information Provider* is required to obtain the layout information of the nodes in a composition script that represent input models. We call this a *Target Information Provider* because it determines the main structure of the composed diagram (cf. Figures 3a and 3b). It gathers the x and y values of the geometrical shapes that represent the input models in the graphical script. Height and width are not that important here. This Information Provider depends on the layout format used for composition scripts in the supported composition engine.

In Reuseware, composition scripts (called composition programs) are created in a graphical editor which was developed with GMF. Consequently, we implemented one *Target Information Provider* that depends on the GMF Notation Model for layout information and on Reuseware to obtain the relationship information (Requirement 4) between nodes in a composition script and input models.

2.4.3 Comparator (LC3)

A *Comparator* ensures that layout composition is performed in a deterministic order. It is required when the semantic model composition does not depend on a deterministic order, but the layout adjustment does.

We implemented one Comparator that sorts input models according to their x position in the composition script (i.e., the one given by the Target Information Provider). This is needed for the *Horizontal Sorting* algorithm but was also used for the *Uniform Scaling* algorithm to have a deterministic order (although any other deterministic Comparator could be used here).

2.4.4 Arranger (LC3)

An *Arranger* does the actual layout adjustment if overlaps exist. Therefore, it first checks for overlaps and decides if additional adjustment is required. An Arranger could do these steps repetitively, depending on the adjustment algorithm. That is, if after one adjustment overlaps do still exist, it can do another algorithm run.

As mentioned in Section 2.3, we implement *Horizontal Sorting* and *Uniform Scaling* as two different Arrangers. Depending on the concrete composition, both algorithms yield results of different quality, as we will discuss in Section 3.

2.4.5 Materialiser (LC4)

The last step is materializing the computed layout in an actual diagram. This is realized by a *Materialiser*.

Materialisers also have to be implemented for each layout format that should be supported. Thus, we implemented one for GMF and one for TOPCASED.

In the next section we evaluate our framework in combination with Reuseware using two graphical DSMLs, where one is utilising GMF and one TOPCASED as layout format. For that, different combinations of the mentioned component implementations are used. We call such a combination a layout composition *strategy*. The Source Information Provider and the Materialiser are always determined by the format used by the corresponding DSML. As Target Information Provider and Comparator we always use the only implementation we provided so far. The Arrangers however

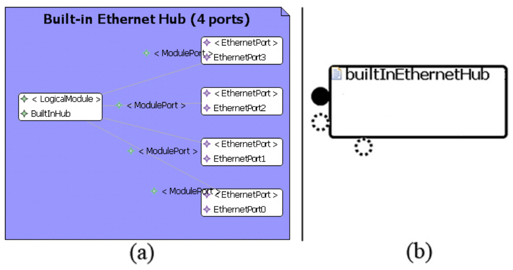

Figure 4: (a) A CIM model (b) The same model represented in a Reuseware composition script

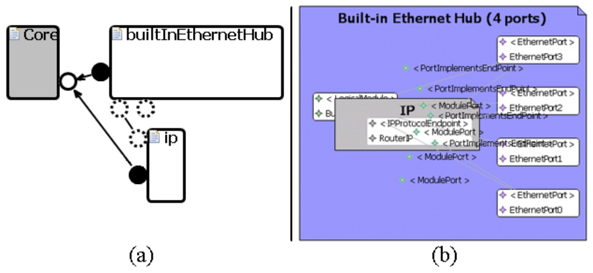

Figure 5: (a) Composition script for the `EthernetIP-Interface` model (b) Composed model

can be varied (no Arranger, Horizontal Sorting, Uniform Scaling). We compare the different possible combinations and discuss the results.

3. EVALUATION

In this section we evaluate our layout composition framework on two different model compositions that were realized with the Reuseware Composition Framework in earlier works: [8] in Section 3.1 and [10] in Section 3.2. We apply different configurations of our layout composition framework and compare the results. Afterwards we discuss what we have achieved so far and what the next steps towards a generic layout composition framework are in Section 4.

3.1 Common Information Model DSML

The first model composition uses models from the telecommunications domain defined with a DSML that implements the Common Information Model (CIM) standard [3]. A metamodel defined in Ecore and a graphical GMF-based editor for the language were developed by Telefonica R&D and Xactium in the MODELPLEX research project [5]. In the following we concentrate on the layouting aspects of the model compositions. More details of the semantic model composition can be found in [10] and online[1].

Figure 4 shows (a) the CIM model `BuiltInEthernetHub` in the CIM GMF editor and (b) the representation of that input model in a composition script in Reuseware's composition script editor. The node in the composition script has different circles attached to it which are called *Ports* in Reuseware. Each Port points at a number of model elements in the input models that are modified during the model composition. For more details please consult [8] and the Reuseware website[2].

CIM models are composed in different stages, where each stage represents a different level of abstraction. The original input models (e.g., Figure 4a) developed with the mentioned GMF editor reside on Level 1. A composition script that composes these models defines a Level 2 composition. A composition script, that uses the results of Level 2 compositions as input models is located on Level 3 and so on.

The `BuiltInEthernetHub` (Figure 4a) is a model of Level 1. To compose the network model `EthernetIPInterface`, a composition script on Level 2 was created which is depicted in Figure 5a. In addition to the `BuiltInEthernetHub` the script contains the input models `Core` and `IP`. The `Core` is an empty model into which CIM model elements are composed. Thus, it holds no graphical representation and layout information. The `IP` contains only one model element and consequently one graphical node.

We execute the composition defined in Figure 5a with three different layout composition strategies. Each strategy uses the Source Information Provider and Materialiser developed for GMF, the Target Information Provider that works for Reuseware's composition scripts and the Comparator. The first strategy applies no layout adjustment, the second uses Horizontal Sorting and the third Uniform Scaling.

No layout adjustment Figure 5b shows the diagram that results from the composition of Figure 5a without layout adjustment. We observe that the elements overlap (Goal 1) since the diagrams are bigger than the icons in the composition script (Figure 5a). This destroys the positioning in the developer's Mental Map (Goal 2) and only Goal 3 is reached.

Horizontal Sorting In Figure 6a Horizontal Sorting is used for layout adjustment. We observe, that the overlap has been removed. The overlapping nodes have been moved along the x-axis. While Goal 1 is reached here, Goal 2 is not completely satisfied. The `IP` model which is located below the `BuiltInEthernetHub` in Figure 5a is now located on the right of it.

Uniform Scaling In Figure 6b we utilise Uniform Scaling (with a scale factor s=2). Here, the mental map is well preserved and all three Goals are satisfied.

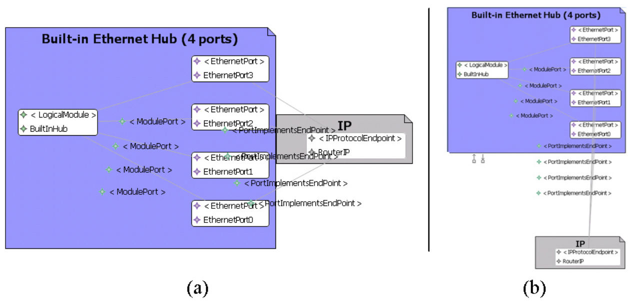

Figure 6: Adjusted layout: (a) Horizontal Sorting (b) Uniform Scaling

[1]http://reuseware.org/index.php/Abstract_CIM_DSLs
[2]http://reuseware.org

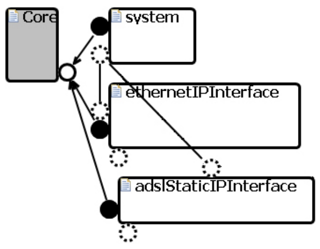

Figure 7: Level 3 composition script

We have seen that the layout adjustment is necessary even for a small model to avoid overlaps while preserving the Mental Map. While Horizontal Sorting performs worse than Uniform Scaling concerning the exactness of neighborhood relations, it yields a more compact design. Thus, it could still be an acceptable option here.

A more complex composition is shown in Figure 7. It is a Level 3 CIM abstraction that reuses results of earlier compositions on Level 2 which are the `EthernetIPInterface` from above as well as the models `ADSLStaticIPInterface` and `System`. We apply two different layout strategies using the two different algorithms to examine how they behave for larger models and how our framework behaves in a staged model composition.

Figure 8a (Horizontal Sorting) and Figure 8b (Uniform Scaling) show the different composition results. In principle, the same observations as above can be made, but the mentioned issues become more obvious. In the case of Horizontal Sorting, everything is aligned along the x-axis, while it was aligned along the y-axis in Figure 7. In the case of Uniform Scaling, the problem of less compact design increases. Although we used only a small scale factor (s=2), the diagram is getting relatively large. This is due to the fact that all element sets are moved uniformly in different directions, resulting into unused spaces between smaller element sets.

In Figures 8a and 8b we can also observe that a layout composed in an earlier step (i.e., the layout of `EthernetIPInterface` which is composed by Figure 5a) is not modfied anymore. Thus, different strategies can be applied at different stages of a composition. In Figures 8b, Horizontal Sorting was applied to compose `EthernetIPInterface` which keeps

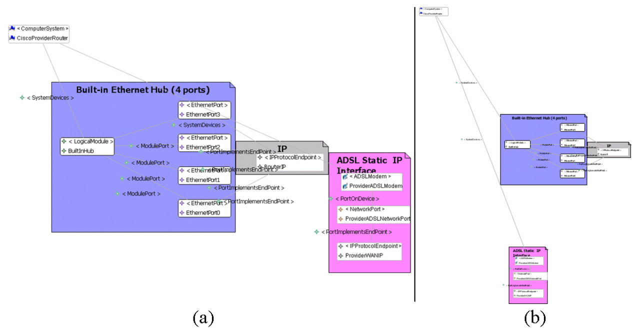

Figure 8: Composed Level 3 diagram: (a) Horizontal Sorting (b) Uniform Scaling

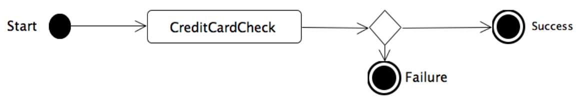

Figure 9: A business process extension modelled as UML activity in TOPCASED [19]

the overall layout more compact compared to the case where Uniform Scaling is applied everywhere (not shown).

3.2 UML Activities for Business Processes

Business processes as presented in [6] can be modelled as UML activity models. A core process can be extended with new sub-processes by composing those models with Reuseware as we did in [8]. An example of a sub-process is shown in Figure 9. When it is composed into a larger core activity, the *initial* and *final* nodes (black circles) are replaced with other nodes in the core—integrating both activities.

We tested our layout composition framework with the models of [8]. This time, we had to use the TOPCASED specific components, since the diagrams were created with the TOPCASED UML editor. The adjustment worked in the same manner as for the CIM models confirming the results about strength and weaknesses of the algorithms and demonstrating that the framework can be used with other DSMLs.

4. NEXT STEPS

This section discusses the results of the last section and points out future work to improve our layout composition framework. We have seen throughout the evaluation that there is not one best strategy for layout adjustment. Which is the best strategy rather depends on many factors from the sizes of the input models up to the personal taste of the developer and how he uses the DSML at hand. A possibility is to make the developer aware of the different strategies and let him experiment with different ones—as we did in the evaluation. However, if many compositions are defined, this extra work of evaluating (and re-evaluating) all strategies each time a composition or one of its input models changes can become a tedious task. It should be possible to select strategies automatically based on further analysis of the input models or by allowing the developer to specify criteria for this selection. This requires analysis of a broader example space in the future.

Since we made certain assumptions about the models and the model compositions when we decided how to preserve the Mental Map, there are cases that are not so well supported by our framework at the moment. Consider the UML activity example (Figure 9). Here the nodes `Start`, `Success` and `Failure` are replaced by others during a composition. Currently, the layout information about these replaced nodes is always discarded. However, there are also examples where it seems to be more intuitive to position the replacing node at the position of the replaced node—for instance, if only one node is inserted and not a whole diagram. This however highly depends on the concrete kinds of compositions that are performed. If and how the best strategy can be determined automatically will have to be explored by evaluating different kinds of compositions. In addition, if a replacing node should take the position of the replaced node, the composition framework needs to reveal the relationship between such nodes (extension of Requirement 4).

Another thing we have not considered yet are diagrams that do not follow the simple node and edge paradigm but are more restrictive (e.g., UML sequence charts). In such cases, the layout adjustment possibilities are limited on the one hand, but might also not be necessary on the other hand (because a "good" layout is enforced by the nature of the graphical formalism). More investigations are required here.

A point that might hinder the combination with other model composition engines is the requirement for a graphical composition script (Requirement 3), since many such tools come with textual specification languages. In principle, such languages could also be handled by translating text positions (e.g., the order in which input models are referenced) into graphical layout information (by a specific Target Information Provider). Consequently, to support a textual language, a useful translation has to be found.

5. RELATED WORK

Many modelling tools do not pay proper attention to layout information today. Graphical modelling tools and frameworks such as GMF [7], TOPCASED [19], Rational Software Architect [9], Borland Together [2], MagicDraw [13] or Fujaba [12] offer facilities which apply layout algorithms to whole or partial diagrams. Despite the fact that these algorithms do not consider the Mental Map of the existing layout and often fail to produce viable results for large diagrams, the tools also do not offer facilities to preserve or transfer layouts from one diagram to another. MDSD process tools such as openArchitectureWare [14] or AndroMDA [1] completely ignore layout information when composing or transforming models.

Our work focuses on model compositions that are performed between models defined in one DSML. Another important discipline is model transformation between different DSMLs. Here, layout information is also seldom handled and also not considered in standardization efforts such as QVT [16]. Pilgrim et al. [20, 21] used trace links created during a model transformation to obtain layout information from the source diagram to layout the target diagram. They however do not discuss what the limitations for the model transformation are they support and only considered Uniform Scaling to remove overlaps so far.

6. CONCLUSION

We presented a generic layout composition framework to improve layout preservation in MDSD. The architecture of the framework and the components we implemented were introduced and utilised in several examples. These experiments showed that the provided solutions are a great improvement over current practice. They also showed, however, weaknesses and limitations of our work so far. Based on this, we identified challenges as a base for further work to improve the quality and genericity of the presented layout composition framework. In the future, we will tackle these challenges and perform more experiments on different DSMLs with distinct graphical syntaxes.

7. REFERENCES

[1] AndroMDA Development Team. AndroMDA. http://www.andromda.org/, 2009.

[2] Borland. Borland Together. http://www.borland.com/us/products/together/, 2009.

[3] Distributed Management Task Force Inc. (DMTF). Common Information Model Standards. http://www.dmtf.org/standards/cim/, 2008.

[4] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. *Research Report IIAS-RR-91-16E*, 1991.

[5] A. Evans, M. A. Fernández, and P. Mohagheghi. Experiences of Developing a Network Modeling Tool Using the Eclipse Environment. In *Proc. of ECMDA-FA'09*, volume 5562 of *LNCS*. Springer, 2009.

[6] M. Fritzsche, W. Gilani, C. Fritzsche, I. T. A. Spence, P. Kilpatrick, and T. J. Brown. Towards Utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis. In *Proc. ECMDA-FA'08*, volume 5095 of *LNCS*. Springer, 2008.

[7] GMF Development Team. Graphical Modeling Framework. http://www.eclipse.org/gmf/, 2009.

[8] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On Language-Independent Model Modularisation. In *Transactions on Aspect-Oriented Development*, LNCS. Springer, 2009. To Appear.

[9] IBM. Rational Software Architect. http://ibm.com/software/awdtools/architect/swarchitect/, 2009.

[10] J. Johannes, S. Zschaler, M. A. Fernández, A. Castillo, D. S. Kolovos, and R. F. Paige. Abstracting Complex Languages through Transformation and Composition. In *Proc. of MoDELS'09*, LNCS. Springer, 2009.

[11] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Research Report ISAS-RR-94-6E*, 1991.

[12] U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. In *Proc. of ICSE'00*. IEEE, 2000.

[13] No Magic, Inc. MagicDraw. http://www.magicdraw.com/, 2009.

[14] oAW Development Team. openArchitectureWare. http://www.openarchitectureware.org/, 2009.

[15] Object Management Group. Diagram Interchange Specification, v1.0, 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-04.

[16] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation, 2008. http://www.omg.org/cgi-bin/doc?formal/08-04-03.

[17] R. C. Read. A New Method for Drawing a Planar Graph Given the Cyclic Order of the Edges at Each Vertex. *Congressus Numerantium 56*, 1987.

[18] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework, 2nd Edition.* Pearson Education, 2008.

[19] TOPCASED Development Team. TOPCASED Environment. http://www.topcased.org, 2009.

[20] J. von Pilgrim. Mental Map and Model Driven Development. In *Proc. of LED'07*. EASST, 2007.

[21] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and Visualizing Transformation Chains. In *Proc. of ECMDA-FA'08*, volume 5095 of *LNCS*. Springer, 2008.