Generic tool for visualization of model differences

Mark van den Brand
Eindhoven University of Technology
Den Dolech 2
5612 AZ Eindhoven
m.g.j.v.d.brand@tue.nl

Zvezdan Protić
Eindhoven University of Technology
Den Dolech 2
5612 AZ Eindhoven
z.protic@tue.nl

Tom Verhoeff
Eindhoven University of Technology
Den Dolech 2
5612 AZ Eindhoven
t.verhoeff@tue.nl

ABSTRACT

Model comparison includes three major concerns: presentation, calculation, and visualization of model differences. In this paper we address the concern of visualization of model differences in the context of model configuration management systems. Since models are considered the main artifacts in model configuration management systems, we require that the differences between models are represented by means of a *differences model*, which conforms to a differences metamodel.

The traditional approaches to visualization of model differences based on a textual, tree-like, or even diagrammatic representation of differences do not scale well in the presence of large differences models. The cause for this is that it gets harder to comprehend the meaning of differences as the size of the differences models increase. We focus on this problem and propose a solution that extends and combines two existing approaches, namely polymetric views and a generic visualization framework for metamodel-based languages. Polymetric views offer good overview, zoom, and filtering capabilities. A visualization framework for metamodel-based languages is used to visualize differences details. By using the combination of these two approaches, it becomes easier to comprehend the meaning of differences even in large models. This paper describes both the details of our solution, and a generic tool that implements the described solution.

Categories and Subject Descriptors

D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.7 [**SOFTWARE ENGINEERING**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.9 [**SOFTWARE ENGINEERING**]: Management—*Software configuration management*

Keywords

Metamodeling, Model comparison, Model differences visualization

IWMCP '10, July 1, 2010 Malaga, Spain

1. INTRODUCTION

Model Driven Software Engineering (MDSE) is a field of Software Engineering which focuses on models as main design artifacts, and uses model transformations as means of relating models. Consequently, mature model configuration management systems are required to manage the complexity of modeled systems in MDSE environments. One of the major functions of model configuration management systems is model comparison. Model comparison (model differencing) is a complex process which consists of three concerns: representation, calculation, and processing of differences [9]. The rationale behind this separation of concerns is that usually it is not only required to calculate differences, but it is required to store, process, and visualize them in the context of a model configuration management system.

In this paper we consider visualization of model differences. In [15] it was observed that traditional difference visualization approaches using text-based, tree-based or even diagrammatic visualization techniques, poorly scale with the size of the differences model. There are two reason for this behavior. The first reason is based on the fact that model differences are considered information content. Thus, we believe that their visualization should be based on the following idea of information visualization specified by Shneiderman [12]: overview first, zoom and filter, then details-on-demand. However, the existing approaches are not suited to visualize model differences in a way that completely supports the above mentioned idea. The second reason can be derived implicitly from the first reason: the traditional approaches use only one technique. However, in order to provide the best insight into the meaning of differences, more than one technique should be used. In Section 2 we give a detailed discussion on these two reasons.

In order to improve the model differences visualization capabilities provided by traditional approaches, we extend and combine two existing techniques: polymetric views [10] and a generic visualization framework for metamodel-based languages. The first technique provides good overview, zoom, and filtering capabilities, and the other technique supports the semantically rich detailed representations of differences. A recent approach, described in [11], which combines a tree-based visualization technique provided by *EMF Compare* [3] with a visualization framework called *GMF* [5], uses similar ideas and makes a step beyond traditional approaches, but is tightly coupled with the Eclipse framework [2]. Our approach, and an associated tool, are generic, and both are metamodel and framework independent. The preliminaries required for understanding our approach are given in Sec-

tion 3. Thereafter, in Section 4, a detailed description of our approach is presented. Next, in Section 5, details of a prototype tool that implements the described approach are presented. Finally, in Section 6 we discuss the results and propose some ideas for further research.

The main contributions of this paper are:

- We discuss two reasons why traditional differences approaches poorly scale with the size of the difference models
- We specify a new visualization approach that combines polymetric views and a framework for visualization of metamodel-based languages, and thus scales excellently in the presence of large difference models
- We present the details of a model difference visualization tool based on the specified approach

2. MODEL DIFFERENCES AS INFORMATION CONTENT

In our approach to visualizing model differences, we adopt the idea of information visualization proposed by Shneiderman [12]: Overview first, zoom and filter, then details-on-demand. The reason for adopting this idea is based on the fact that model differences are information that needs to be visualized. Thus, it is required to have overview capabilities, such that the global meaning of differences can be comprehended. Next, it should be possible to zoom in and filter differences, such that the user of configuration management systems (referred to as user in the rest of this section) can syntactically and semantically associate the differences to the parts of the models that those differences are related to. Finally, the selected differences should be rendered by using a sufficient level of detail to provide the most insight into their meaning.

Traditional approaches to difference visualization do not completely follow Shneiderman's idea. It was observed that they scale poorly with the size of the differences model [15]. One of the reasons for this is that traditional visualization techniques cannot fully support Shneiderman's idea. In order to support this claim, we will provide a formalized definition of overview, zoom, filtering, and details-on-demand in the context of model differences. Thereafter, we will show why the traditional approaches have problems in satisfying these requirements. The important thing to know is that the ultimate goal of all the described concepts is providing the *maximum insight into the meaning of model differences*.

Useful overview techniques in the context of model differences should provide an overview of one or both models used to calculate the differences, and should relate the elements of those models to the elements in the differences model. Thus, an overview should allow the user to get a first grip on the meaning of model differences in the context of the models that they are calculated from. The zoom should allow the users to focus on the interesting details. The filtering should allow the user to quickly navigate to the parts of the difference model that are important to them. The filtering should be based on the meaning of the differences, not only on their structure. The details-on-demand should allow the user to extract and focus on the details of selected differences.

Having defined the concepts of overview, zoom, filtering, and details-on-demand, we will provide an explanation for why the traditional approaches to visualization of model differences do not satisfy all of these concepts.

For example, in text-based visualization, it is hard to provide overview, because the text usually does not fit the display. Modern text-based visualization techniques, for example the technique described in [13] provide means for overview, zooming, and filtering by having a slider on which the position of changed elements in relation to initial models is marked. However, the overview, zooming and filtering in this case are syntax-based and do not provide much insight into the meaning of differences. Furthermore, in order to get insight into the details of selected differences a user needs to spend quite some time interpreting the textual representation of the differences.

In tree-based visualization, for example in [3], getting the overview of the differences is still not so easy for larger models because the size of the visible part of the tree is limited by the size of the display. However, the combination of overview and inherent zooming allows for a much better insight in the meaning of the differences. The filtering in tree-based visualization approaches is also easier than in text approaches, since the hierarchical structure of the tree allows for an easier interpretation of the meaning of differences. However, the filtering still does not provide the means to extract just the required combination of parts of the model and parts of the differences model. Also, the details of the differences are still not easy to comprehend, since the user needs to interpret the tree representation of the differences.

In the diagrammatic visualization approaches, for example in [11], the differences are represented in a natural visualization environment of the metamodel of the models used to calculate the differences. The combination of the overview and the zoom allows the differences to be examined at the appropriate level of details, thus providing clear insight into their meaning. For example, a small overview is combined with a larger zoom view, such that the portion of the system that is currently in the focus of the zoom is also outlined in the overview. However, the filtering of differences based on their meaning is still hard; for example, it is hard to extract all element types having a certain property, because these element types might be in different parts of the system, and thus they might only be visualized in different diagrams.

The mentioned examples uncover another reason why traditional approaches poorly scale with the size of the differences model—the traditional approaches use only one visualization technique. Thus, approaches that aim to provide a visualization of model differences that scales well, have to combine multiple techniques.

3. PRELIMINARIES

As already mentioned, the three main concerns in the process of comparing models are the representation of, the calculation of, and the processing (e.g., visualization) of model differences. In this section we first describe our approach to the representation of model differences, which specifies the difference presentation format used in our visualization technique. This approach is generic and metamodel-independent, and is similar to approaches like EMFCompare [3] or the approach presented in [7]. Next, we briefly describe our approach to the calculation of model differences that produces differences models that conform to the specified presentation format. The details of both approaches can be found in [14].

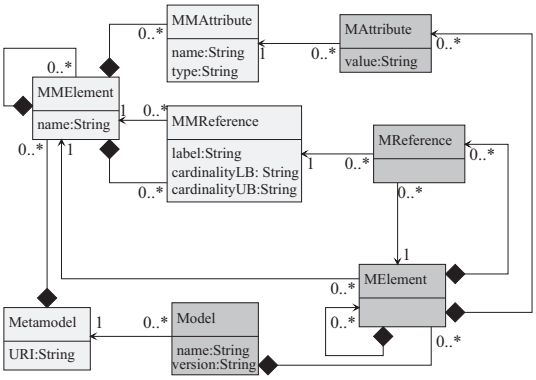

Figure 1: Metametamodel that models used in the calculation of differences conform to

3.1 Representation of model differences

Our approach to the representation of model differences allows those differences to be used in Model Driven Engineering environments. Thus, the differences between two models are represented by a difference model which conforms to a differences metamodel. The differences metamodel is based on the metametamodel that the models (and metamodels) used in the process of calculating differences conform to. This metametamodel is depicted in Figure 1. This metametamodel describes both metamodels and models. Metamodels are obtained by instantiating the *Metamodel* element, and models are obtained by instantiating the *Model* element. This metametamodel can be considered as a *domain specific* metametamodel which is geared towards representation of model differences, and not towards general modeling like MOF or Ecore. Thus, it is more comparable to the cores of MOF and Ecore, and does not contain some of the advanced modeling concepts like packages or inheritance. However, the ideas presented using this metametamodel also apply to other, more complete, metametamodels.

The differences metamodel is an extension of the introduced metametamodel and is depicted in Figure 2. The difference models are instances of the *DifferencesModel* element. The main building blocks of the difference models are instances of *ChangedElement*, *DeletedElement*, and *AddedElement*. Assuming that the differences model represents the differences between models A and B, then the instances of the *AddedElement* are elements that are in model B and not in model A, the instances of the *DeletedElement* are elements that are in model A but not in model B, and the instances of the *ChangedElement* are elements that represent the *same entities* in both models but are not structurally identical.

3.2 Calculation of model differences

Traditional approaches to the calculation of model differences are based on tree-matching algorithms. These algorithms *match* the nodes of two trees that represent two models being compared and based on this matching the differences are calculated. Several types of matching are recognized: static-identity, signature-based, similarity based or language-specific. Our algorithm for calculating differences is also based on tree-matching algorithms. Unlike traditional approaches that support only one type of matching, our algorithm is defined in such a way to support all four types of matching. In order to allow such a highly configurable calculation process, we extend the differences metametamodel with additional elements. The extended metametamodel is interpreted as a calculation metamodel and is depicted in Figure 3.

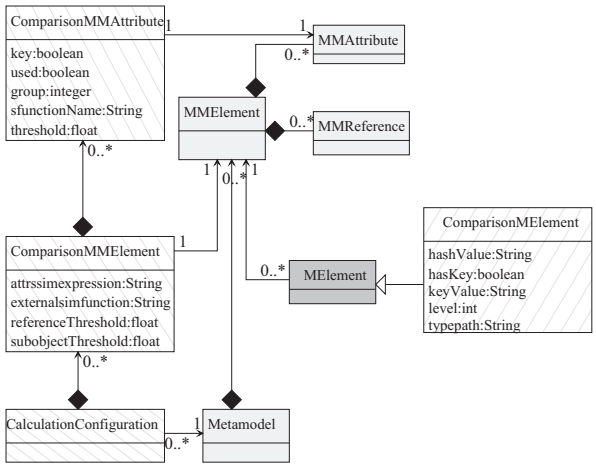

Figure 3: Calculation metamodel

Calculation models are used by our algorithm and they have two important features. The first feature is represented by instances of the *CalculationConfiguration* element. This feature of the calculation model opens the possibility of specifying the metamodel-specific configurations that are used to influence the calculation process of models related to the specific metamodel. Thus, for all models that conform to a specific metamodel, only one calculation configuration needs to be set. The second feature is represented by instances of the *ComparisonMElement*. The instances of the *ComparisonMElement* represent nodes in model trees, and are generated for each model element in a preprocessing step, with the help of the metamodel-specific configuration.

3.2.1 Model comparison algorithm

The input to our comparison algorithm are two models A and B and the metamodel-specific configuration. Our comparison algorithm consists of three steps. In the first step the similarities between objects in models A and B are calculated. We say that two objects are similar if they can be considered the *same* entity. We define similarities by using a similarity function which returns *true* if two objects are similar, and *false* otherwise.

In the second step, based on the similarities found, a matching of objects is calculated. The matching is performed by traversing the tree top-down. At the first level, based on the similarities found, some objects may be matched. For all matched objects at the first level, the matching process continues recursively until the bottom of the tree is reached or there are no sub-objects that can be matched.

In the last step, the calculation of differences is done based on the matchings found. This step consists of three substeps. The first sub-step is the calculation of differences in terms of deleted or changed sub-elements and attributes (for all matched and not-matched elements in model A). The second sub-step consists of the calculation of added elements (for all non-matched elements in model B). The third substep consists of the calculation of changes in references.

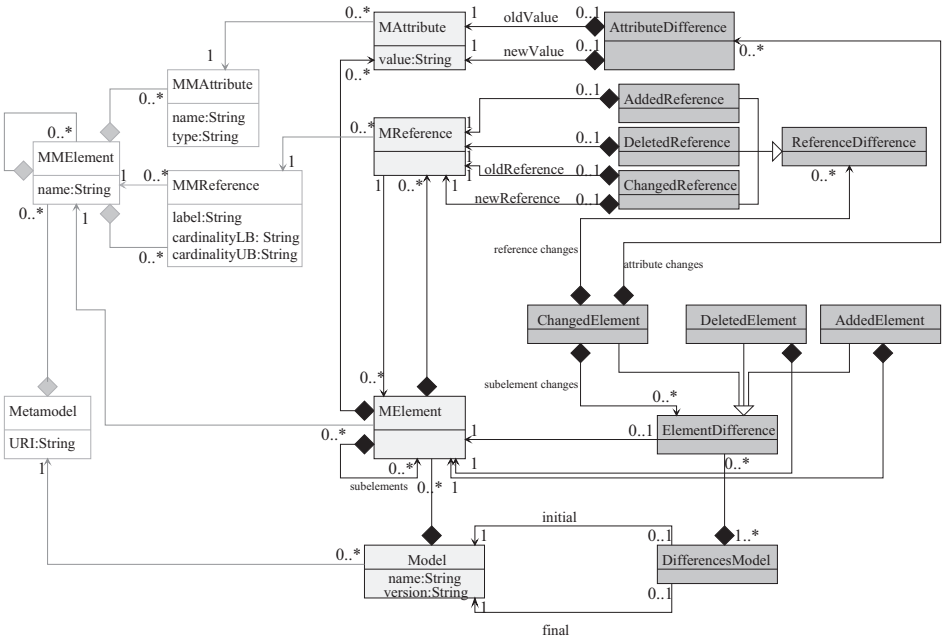

Figure 2: Differences metamodel

4. DIFFERENCES VISUALIZATION

As already noted, our approach to the visualization of model differences extends and combines two other approaches. The first approach involves *polymetric views*, which were first described in [10]. A polymetric view is a lightweight visualization component, which gives insight into a certain aspect of the system by combining simple visualization and metric information. The idea of [10] was used for visualizing model differences in [15]. In the approach of [15], two trees representing two models being compared are represented in a *unified* form. This unified form represents *matched* elements in models being compared as the same node in the tree. This representation allows the definition of metrics related to model differences based on the unified tree. These metrics can then be used to specify views that provide insight in the relation of model differences and model elements.

We follow the approach of [15], and use a unified tree as a representation of both models used in comparison. However, since we do not impose a restriction that a calculation process is included in the visualization process, we do not use calculated similarities between objects as a basis of a metric calculation. The metrics in our case are calculated only by using the initial model (used in calculating the differences) and the differences model. Also, we do not presuppose the significance of *changes*, but this significance can be encoded in the metrics calculation functions.

Our approach to polymetric views is generic, in a sense that we allow user-defined views and metric functions. However, as noted by Lanza et al. [10], since the users rarely define their own views and metrics, we defined a small set of metrics, and by using those metrics we defined a default set of views (based on the set of *cluster* views defined in [10]). The set of metrics we defined is presented in Table 1. The metrics apply to instances of MElement, which are also called *objects* in this context. The *subobjects* are instances of MElement contained in another instance of MElement.

Name	Description
MA	Number of attributes
MR	Number of references
MSO	Number of subobjects
NA	Number of changed attributes
NR	Number of changed references
NSOD	Number of changed direct subobjects
NSOT	Number of changed subobjects which takes into account the transitive closure of the subobject relation
NC	Sum of attribute, reference and subobject changes
RNA	Relative number of changed attributes
RNR	Relative number of changed references
RNSOD	Relative number of changed direct subobjects
RNSOT	Relative number of changed subobjects which takes into account the transitive closure of the subobject relation
RNC	Relative number of sum of attribute, reference and subobject changes
MMName	Encoding of the MMElement instance names

Table 1: The defined set of metrics

Based on this set of metrics, we defined a set of views. For example, the *SYSTEM HOTSPOTS* view specified below defines an overview of the system: all elements in the model are visualized, the width of each element is relative to the number of attributes, the height of each element is relative to the number of references in the element. The color of each element is based on the relative number of changes to the element (the color gradient from white to red is used to represent the relative number of changes), the elements are sorted by the color, and the elements are presented in the form of a checker table (a checker table is a simple matrix representation, where elements are visualized in the cells of a matrix based on the specified sort criteria).

- SYSTEM HOTSPOTS - Layout: Checker. Target: Objects. Scope: All. Width: MA. Height: MR. Color: NC. Outline: -. Sort: Color.

The attributes of the views are almost the same as defined in [10]: *Width*, *Height* and *Color* of an icon representing a selected model element. The exception is an extra attribute: *Outline* adapted from [15], which is defined as a colored outline of the icon representing a selected model element. Each of the attributes is related to a defined metric, and this connection is used to provide visual characteristics to the icon representing a model element in a view.

Polymetric views provide a good overview of the changes, the zoom capabilities are also supported by selecting the type of elements to be visualized, and the filtering is done by variations in dimensions and colors of visualized elements. However, the meaning of details of differences is not easily grasped. An example that reveals this problem is depicted in Figure 4. In this example, a simplified *INHERITANCE CLASSIFICATION* view of an example model, and a semantically rich view of the same model are presented.

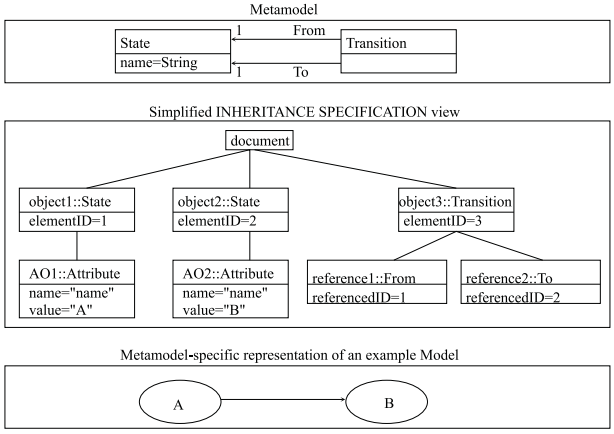

Figure 4: An INHERITANCE SPECIFICATION view and the metamodel-specific representation of the same example model

In order to provide better insight to the meaning of the differences details, we extend the polymetric views approach with the possibility of defining special "views" that expose the details of the differences in a natural (metamodel-specific) way. Thus, for example, a user can select an icon in a polymetric view, and choose to visualize its details in the metamodel-specific way. In software language terminology, the metamodel specific way of visualizing the model corresponds to a *concrete syntax* of a model. The example is given in Figure 5.

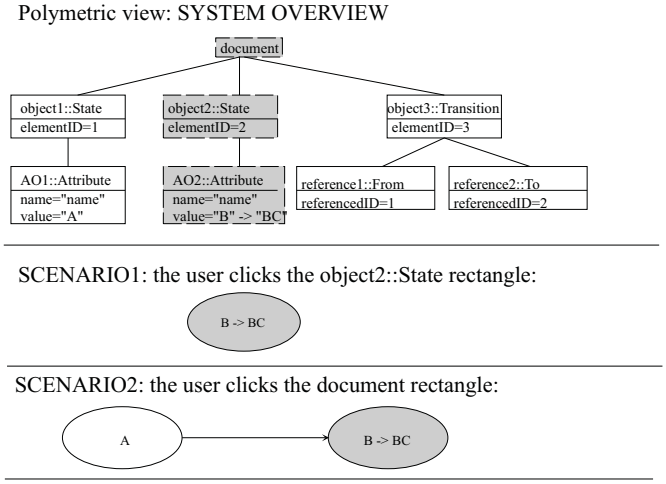

Figure 5: Example of combination of polymetric views and metamodel-specific visualization approaches

The second approach borrows the ideas of an "Open Visualization Framework for Metamodel-Based Modeling Languages" specified in [8], and is also quite similar to the automated approach to generation of model editing environments called EuGENia [4], which allows the users to declaratively map metamodel elements to GMF [5] elements which are used to visualize and edit Eclipse-based models [2]. This permits semi-automatic creation of model editors. However, unlike the authors of EuGENia (and GMF) we are not focused on creating complete editors. Our goal is a framework-independent approach to visualizing model differences that fits into the idea of polymetric views.

In our approach we specify a small set of rule types for defining rules to map an arbitrary metamodel onto a graphical metamodel based on *dot* [1]. This allows visualization of models conforming to the mapped metamodel. By using a *unified* representation of models and differences, it also allows the visualization of model differences. Since the *dot* framework is able to automatically layout graphs, we get the layout of the visualized model differences for free. The rules also allow for the definition of a metamodel-specific differences layout. In the following sections we will first present the mapping, the rationale behind it, and the mapping language. Then we will present our approach to the visualization of differences by using the specified mapping.

4.1 Metamodel to dot mapping

In order to define a generic metamodel-to-dot mapping, we specify a requirement that all elements and references in a metamodel are identifiable. The example metametamodel that we use in this paper supports this requirement (see Figure 1).

The mapping is represented by a set of rules. Each rule specifies a mapping of one metamodel element to one or more *dot* graphical elements. Each *dot* graphical element is described by a simplified metamodel depicted in Figure 6.

We define five rule types. The first rule type defines a

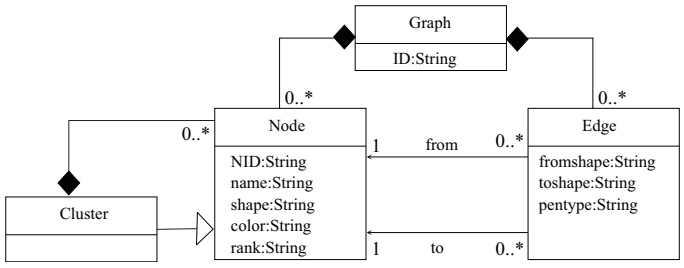

Figure 6: Simplified *dot* metamodel

mapping of an instance of an *MMElement* into a dot node. All instances of the *MElement* related to the mapped instance of an *MMElement* will be graphically represented by a dot node of shape and size as specified by the mapping. The second rule type defines a mapping of an instance of an *MMElement* into a dot cluster. The dot cluster is a rectangular area that groups a set of dot nodes. This rule enables the creation of a *compositional* hierarchy of elements. The third rule type defines a mapping of an instance of an *MMElement* into a dot edge. This rule type enables the creation of connections between objects. This can be used to visualize objects that represent, for example, associations or generalizations in UML class diagrams or transitions in UML state machine diagrams. The fourth rule type defines a mapping of an instance of an *MMReference*, into a dot edge. This rule type enables the creation of *tree-like* hierarchical structures. The fifth rule type defines a mapping of an instance of an *MMElement* into a *nexus*. The nexus is a dot node which has one or more incoming dot edges, and one or more outgoing dot edges. The nexus can be used to visualize elements like, for example, pseudostates in UML state machine diagrams. The reasons for specifying these rule types, the details of those rule types, and example mappings, can be found in the Appendix A.

4.2 Using the defined mapping to visualize the differences

Consider the differences metamodel as defined in Section 3. There are three basic types of differences: added, deleted or changed objects or parts of objects. We will use *coloring* as a way of expressing differences, and thus we assign three colors to these differences. The added objects or parts of objects will be colored green, the deleted objects or parts of objects will be colored red, and the changed objects or parts of objects will be colored blue. The visualization uses both the initial model and the differences model to create a visualization of differences.

The visualization of differences depends both on the mapping, and on the type of difference. We will now describe all the possible combinations of types of differences and mappings of model objects.

If the difference is the addition of an object, and the object is mapped to a node or a cluster, the node or a cluster is presented colored green. If the difference is the addition of an object, and the object is mapped to an edge, the edge is colored green. If the difference is the addition of an object, and the object is mapped to a nexus, the nexus is colored green with all incoming and outgoing edges also colored green.

The coloring principle for the added objects also holds for deleted objects, which are colored red, and for changed objects, which are colored blue.

5. TOOL

The implemented visualization prototype tool is part of the Java-based framework that also supports the representation of models and metamodels, as well as the representation of and the calculation of model differences [6]. In a default setting, the differences are visualized by using a set of predefined polymetric views. This set of predefined views can be changed and extended easily by the users of the tool. The predefined views use a set of predefined metrics. This set of metrics can be extended by new metrics conforming to the predefined interface (new metrics are implemented as Java methods). In order to visualize differences between models conforming to a specific metamodel in a natural way, i.e., in order to use a framework for visualization of metamodel-based languages, a mapping between that metamodel and a *dot* metamodel should be defined by the user. However, this mapping needs to be defined only once for a specific metamodel, and can then be reused for visualizing all model differences obtained by comparing models conforming to that metamodel.

The implemented tool is capable of visualizing the differences between models, completely metamodel independent, as long as models and metamodels provided conform to the metametamodel specified in Section 3.1.

5.1 Use case

To provide more insight into the possibilities of our approach, this section provides a small use-case scenario.

Assume that the first designer has created model A, and that afterwards a second designer has changed that model to a model A'. Next, the first designer would like to inspect the changes to the initial model. In order to do that, he would like to have an overview of the system, such that the elements of the system and the degree of change to each element are given in a form of a checker table, such that he can discern the most changed parts of the system, and he can get an overall impression of changes to the system. Also, he would like to select a changed element and to check the changes to that element in a natural environment for that element (for example to examine the class diagram of the most changed class). Assuming that the models A and A', as well as their metamodel (M_A) conform to the required metametamodel (see Figure 1), the first designer should first invoke the differences calculation tool to calculate the differences. The difference calculation tool initially uses a predefined metamodel-independent calculation configuration, however this configuration can be changed to obtain more precise results.

Next, in order to visualize differences in a metamodel-specific way, the first designer needs to define the mapping between the metamodel M_A and the *dot* metamodel. Then, the first designer can invoke the visualization tool and choose an appropriate view (e.g., *SYSTEM HOTSPOTS* view), to get an overview of the differences. Zooming is done by selecting the appropriate type of elements to visualize in a view, and filtering is done by consulting the color of the elements (e.g., more red are more changed elements). Thereafter, the first designer can select a glyph present in this view and visualize it in a metamodel-specific way, in order to obtain details-on-demand of a specific model element.

Two example models A and A', conforming to the meta-

model in Figure 4, and the visualization of their differences (combination of the screenshots from the tool), are presented in Figure 7. The simplified *SYSTEM HOTSPOTS* view is used to visualize an overview of the differences, and a metamodel specific representation is used to provide insight into the details of the differences.

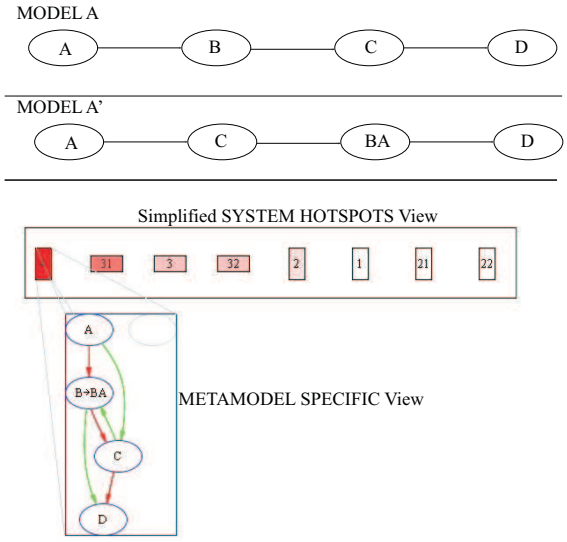

Figure 7: Example differences visualization

6. CONCLUSIONS

6.1 Discussion

The traditional difference visualization approaches using only text-based, tree-based or diagrammatic techniques, scale poorly with the size of the difference models. Using polymetric views to visualize model differences, as presented in [15], was a step beyond traditional approaches. However, insight into the meaning of the details of the differences is not easily obtained by using only polymetric views. Thus, we combined polymetric views with a framework for visualizing metamodel-based languages, to obtain more insight into the details of differences. As already noted, a similar approach as ours is described in [11], where a tree-based visualization technique provided by *EMF Compare* is combined with a visualization framework for visualizing model differences by reusing *GMF*. However, since polymetric views efficiently deal with overview, zooming, and filtering, we believe that polymetric views are better suited for providing an overview of larger difference models than the tree-based visualization technique. Also, our approach and accompanying tool are metamodel and framework independent, and thus easily adaptable for a wide spectrum of modeling environments.

6.2 Future Work

We are currently performing a validation of our approach in an industrial setting. We intend to use the results of that validation for researching new visualization techniques that can be used specifically for the visualization of model differences. Another possible research direction is investigating the applicability of our approach in a setting of an existing model configuration management system. However, our current approach is focused on differences between two models, and configuration management systems usually require a difference function between three models. Thus, research into the applicability of our approach for presenting, calculating, and visualizing differences between three models should be performed first.

Acknowledgements

This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

7. REFERENCES

[1] dot. `www.graphviz.org/` (Viewed June 2010).

[2] Eclipse. `www.eclipse.org/` (Viewed June 2010).

[3] EMF compare. `wiki.eclipse.org/index.php/EMF_Compare` (Viewed June 2010).

[4] Eugenia. `www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/` (Viewed June 2010).

[5] GMF. `www.eclipse.org/modeling/gmf/` (Viewed June 2010).

[6] Metamodel-assisted model comparison tool. `www.win.tue.nl/~zprotic/mctool.html` (Viewed June 2010).

[7] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, pages 165–185, 2007.

[8] P. Domokosa and D. Varró. An open visualization framework for metamodel-based modeling languages. *Electronic Notes in Theoretical Computer Science*, pages 69–78, 2002.

[9] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. *ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, 2009.

[10] M. Lanza and S. Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.

[11] A. Schipper, H. Fuhrmann, and R. v. Hanxleden. Visual comparison of graphical models. In *ICECCS '09: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 335–340, Washington, DC, USA, 2009. IEEE Computer Society.

[12] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

[13] E. Suvanaphen and J. C. Roberts. Textual difference visualization of multiple search results utilizing detail in context. In *TPCG '04: Proceedings of the Theory and Practice of Computer Graphics 2004 (TPCG'04)*, pages 2–8, Washington, DC, USA, 2004. IEEE Computer Society.

[14] M. van den Brand, Z. Protić, and T. Verhoeff. Fine-grained metamodel-assisted model comparison. Accepted to IWMCP 2010.

[15] S. Wenzel. Scalable visualization of model differences. In *CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 41–46, New York, NY, USA, 2008. ACM.

APPENDIX

A. MAPPING RULE TYPES AND EXAMPLE MAPPINGS

As already noted, the mapping between an arbitrary metamodel and a dot metamodel is represented by a set of rules. The goal of each rule is to provide a declarative description of the graphical shape which will be used to represent the instance of a specific metamodel element. Thus, each rule is related to one metamodel element, but is used to visually represent all model elements conforming to that metamodel element. There are five rule types. These types are formulated in such a way to enable an extremely wide range of visualization possibilities.

The detailed rationale behind all the rule types is the following: The *MMElements* are the main ingredients of the metamodels, thus they are included in most rule types. Attributes of *MMElements* are not mapped, because they are considered inseparable parts of *MMElements*, and are treated like that. Thus, all types of rules used to transform *MMElements* have an option to include the attributes in one of the predefined ways (for example, attributes could be visualized in a rectangle which is connected to the node representing the mapped object, separated by horizontal lines and sorted by the name of the attribute). References are included in the mappings in three ways: First, while mapping an *MMElement* into an edge, two instances of *MMReference* of that *MMElement* are selected, such that the instances of *MElements* that are referenced by the instances of *MReferences* that are references by the selected *MMReferences* instances are chosen as the initial or target node connected by an edge. The second way of including the references in the mapping is represented by the fourth rule type and is self explanatory. The third way of including the references in the mapping is represented by the mapping of an *MMElement* into a *nexus*. In this mapping a set of references is chosen as the incoming edges in the *nexus*, and the set of references is chosen as the outgoing edges from the *nexus*.

Next we will give a detailed description of each rule type. Each rule expects a Metamodel element ID. Metamodel element ID is needed in order to connect the rule to a specific metamodel element. The model elements conforming to the specified metamodel element will be visualized by using this rule.

A.1 Rule type 1

This type of a rule can be used to transform model elements to *dot* nodes. It can be used to represent classes in a class diagram, or states in a state machine. It has the following attributes:

- Metamodel element ID
- Node shape
- Attribute name that will be used as a Node label
- The format of visualization of metamodel element attributes
- Positioning attributes list

The *Node shape* is one of the possible shapes for *dot* nodes [1]. The attribute name is the name of the attribute that will be used as a label of this node. If the metamodel element has no attributes, or the node does not need to have a label, this attribute can be set to empty string. Positioning attributes is a list containing the identifiers of four attributes of a model element that will be interpreted as a positioning data for that element(top, left, bottom and right coordinates). The format of visualizations of model element attributes can be one of the following:

- NONE
- RECORD
- HIDDEN RECORD
- TREE

If the format is set to NONE, no attributes of the model element will be visualized. It is important to know that if the format is set to NONE, changes to attributes will not be visible in the visualization of the differences. If the format is set to RECORD, all the attributes are shown in a box, with each attribute separated from others by a horizontal or vertical line. The box representing attributes is connected to the node representing the model element. The HIDDEN RECORD format is similar to the RECORD format, the difference is that the edge that connects the attribute box with the node representing the model element is hidden. If the format is set to TREE, the attributes are visualized by using oval-shape nodes, which are connected to the node representing the mapped model element. An example of all four attributes representation formats is given in Figure 8.

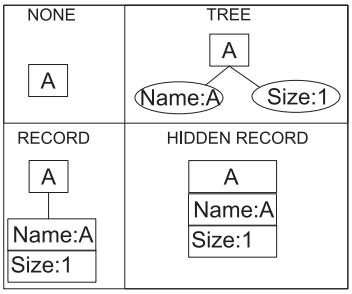

Figure 8: Attributes representation formats

A.2 Rule type 2

This type of rule can be used to transform model elements to *dot* clusters. Clusters can contain other nodes, and are used to represent a containment-type hierarchical structures. This type of rule can be used to represent, for example, a complete class diagram, or a complete state-machine. It can be applied only to metamodel elements that contain other metamodel elements. It has the following attributes:

- Metamodel element ID
- Attribute name that will be used as a Node label
- The list of IDs of sub-metamodel elements that the selected metamodel element contains

- The format of visualization of model element attributes
- Positioning attributes list

Metamodel element ID connects this rule to a specific metamodel element. The attribute name is the name of the attribute that will be used as a label of the resulting cluster. The list of IDs of sub-metamodel elements is used to select which subobjects of the mapped model element will be visualized inside this cluster. All the subobjects (of a model element mapped to a cluster) that conform to the listed metamodel elements will be visualized inside the cluster. The format of visualizations of metamodel element attributes can be one of the following:

- NONE
- RECORD
- HIDDEN RECORD
- TREE
- INSIDE RECORD

The description of all of these formats is the same as in Rule type 1, except of the INSIDE RECORD, which denotes that the attributes should be visualized inside a cluster.

A.3 Rule type 3

This type of rule can be used to transform model elements to *dot* edges. It can be used to represent associations in a class diagram, or transitions in a state machine. It has the following attributes:

- Metamodel element ID
- From reference ID
- To reference ID
- From reference shape
- To reference shape

The rationale behind this element is the following: This type of rule should enable graphical representation of elements that have two or more references of the maximal cardinality 1. Thus, these model elements are actually connections between two or more model elements. For example, the transitions in a state machine or associations in class diagrams should be visualized by a Rule type 3. Thus, this model element will be visualized as a set of edges between all elements that are referenced by its reference having the same ID as the From reference ID, and between all elements that are referenced by its reference having the same ID as the To reference ID. The From reference shape and To reference shape, are shapes of the ends of created edges.

A.4 Rule type 4

This type of rule can be used to transform references to *dot* edges. It can be used to create tree-like structures, instead of cluster-like structures. For example, if one does not want to create a cluster out of a node, he can specify rules that map references of specific model elements to edges, thus creating a tree structure. This type of rule has the following attributes:

- Metamodel element ID
- Reference ID
- From shape
- To shape
- Edge line type

Reference ID is the ID of a reference whose instances will be mapped to edges. From and To shapes are the shapes of the edge begin and end. The *Edge line type* is the type of the line of an edge, and should be one of the recognized *dot* line types.

A.5 Rule type 5

This type of rule can be used to transform model elements to *dot* nexuses. Nexus is a special node that is connected to all instances of all referenced elements by the mapped model element. This node can be used to represent structures like choice, junction or join pseudo states in UML state machines. Specifically, this type of rules is good to represent the structures that connect more than two elements. It has the following attributes:

- Metamodel element ID
- Node shape
- Attribute name that will be used as a Node label
- The format of visualization of metamodel element attributes
- Positioning attributes list

The attributes of this rule type are the same as the ones in rule 1, and have the same meaning.

A.6 Examples

In this section we provide several examples of rules, and their application to the example model.

A.6.1 Example 1

In this example we specify rules that visualize a state machine model (and can be used for other models having the same metamodel) presented in a tree form in the middle of the Figure 4, to its appropriate graphical representation which is presented in the lower part of that Figure.

We will assume that the State metamodel element has an ID S1, that the Transition metamodel element has an ID T1, that the From reference of the Transition element has an ID T1R1 and that the To reference of the Transition element has an ID T1R2.

There are two rules required. The first rule is used to visualize states:

```
RULE:
Type: TYPE1
MetamodelElementID: S1
Shape: Circle
LabelAttribute: Name
AttributesVisualization: HIDDEN
```

This rule is of type 1, and it is thus related to the metamodel element with ID S1. It represents instances of that metamodel element as circles, the label inside the circle is the value of the Name attribute of the element, and the element attributes are not explicitly visualized.

The second rule is used to visualize transitions:

```
RULE:
Type: TYPE3
MetamodelElementID: T1
FromReferenceID: T1R1
ToReferenceID: T1R2
FromReferenceShape: none
ToReferenceShape: normal
```

This rule is of type 3, thus all transitions will be turned into edges.

In the example, there is one transition, for which the From reference (the one with ID T1R1) references object with an ID 1, and the To reference (the one with ID T1R2) references object with an ID 2. Thus, this transition will be visualized as an edge between objects with ID 1, and ID 2, which are states in this example, and are visualized as circles.

Thus, these two rules can be used to visualize the basic state machines.

A.6.2 Example 2

In this example we will extend the metamodel presented in Figure 4 by creating a container for states and transitions (called StateMachine). The new metamodel, tree-view of an example model and appropriate visualization of the example model are presented in Figure 9. In order to visualize the

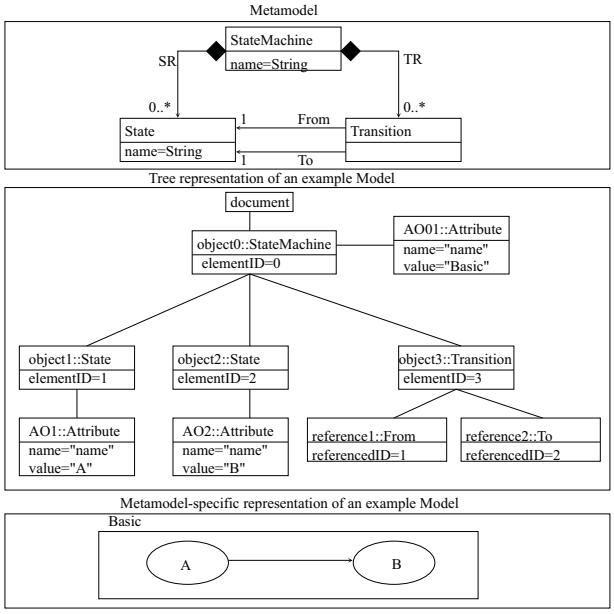

Figure 9: Extended state-machine metamodel and an example model

models conforming to the metamodel presented in Figure 9, two rules defined in example 1 will be used, and one extra rule will be defined. We will assume that the StateMachine metamodel element has an ID SM0, and that it contains metamodel elements State and Transition. The extra rule is as following:

```
RULE:
Type: TYPE2
MetamodelElementID: SM0
SubelementsIDList: S1, S2
LabelAttribute: Name
AttributesVisualization: HIDDEN
```

This rule denotes that all state machines will be visualized as clusters, with their belonging states and transitions visualized inside of them. The attribute Name will be used as a name of the state machine, and no attributes will be visualized.

A.6.3 Example 3

In this example, we will further extend the metamodel presented in the example 2. We will extend this metamodel such that now a start state and an end state can be visualized separately. The extended metamodel, together with a tree-view of an example model, and an appropriate visual representation of an example model are presented in Figure 10. In order to visualize models conforming to this

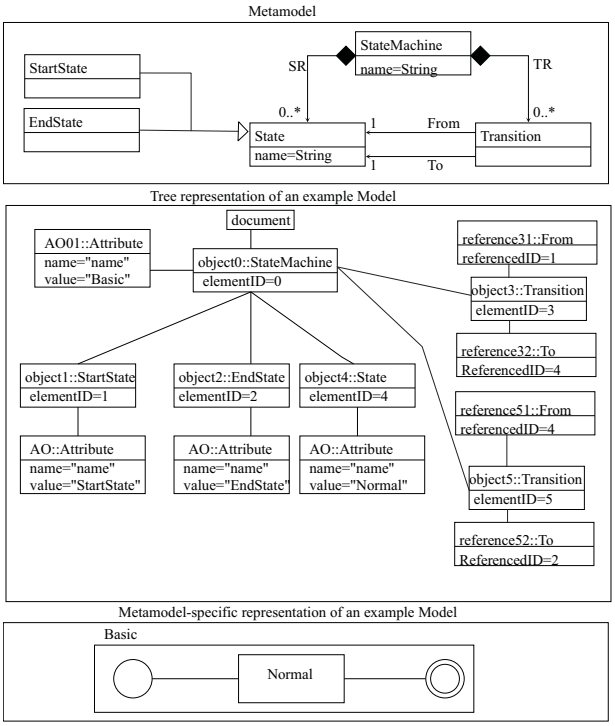

Figure 10: Further extended state-machine metamodel and an example model

metamodel, we will take three already defined rules, change one of those, and we will create two more rules. We will assume that the metamodel element StartState has an ID SS, and the metamodel element EndState has an ID ES. The first changed rule is the one for that is used for visualization of States, now visualizing states as boxes:

```
RULE:
Type: TYPE1
MetamodelElementID: S1
Shape: Box
LabelAttribute: Name
AttributesVisualization: HIDDEN
```

The new rules are as following:

```
RULE:
Type: TYPE1
MetamodelElementID: SS
Shape: Circle
LabelAttribute:
AttributesVisualization: HIDDEN

RULE:
Type: TYPE1
MetamodelElementID: ES
Shape: DoubleCircle
LabelAttribute:
AttributesVisualization: HIDDEN
```

Thus, an instance of a start state will be visualized as a circle, and an instance of an end state will be visualized as a double circle.