Scalable Visualization of Model Differences

Sven Wenzel
Software Engineering Group
University of Siegen, Germany
wenzel@informatik.uni-siegen.de

ABSTRACT

If large models are compared their difference can contain a huge number of local changes. Conventional methods for displaying differences cannot reasonably handle such large differences. This paper proposes a solution to this problem. Our approach is based on the concept of polymetric views and extends it in two ways: firstly, we propose metrics for differences which quantify properties of differences and distinguish relevant from irrelevant changes. Moreover, we propose new graphical features of polymetric views. This combination provides a scalable presentation of differences which makes the changes of large models comprehensible.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*product metrics*; D.2.9 [**Software Engineering**]: Management—*software configuration management*

General Terms

Management, measurement

Keywords

Model-driven development, difference computation, metrics

1. INTRODUCTION

In model-driven engineering developers work mainly or only with models. The term *model* is not restricted to the diagram types of the widely accepted Unified Modeling Language (UML), but it also includes domain specific languages such as Matlab/Simulink diagrams in automotive engineering. According to the trend of growing developer teams working concurrently, the ability to calculate differences between models became very important. Differencing is necessary for comprehension of changes, especially in collaborative work.

Experience shows that models in industrial applications become very large. The size of these models leads to two problems: firstly to compute their difference sufficiently fast – a suitable algorithm has been proposed in [5] – and secondly to present the difference to a developer. Conventional methods of displaying differences fail with large differences: they let the user drown in a plethora of small changes, and they are based on a graphical display of one or both of the compared models, which need too much space. Users need first an overview and must be enabled to capture the relevant changes.

We propose a scalable presentation of model differences based on polymetric views [2]. Polymetric views provide a compact presentation of metrics by combining up to five metrics in one single view. Metrics are a widely used approach for getting an overview of large systems. They are functions that map software characteristics onto numerical values. We introduce *difference metrics* as functions that map changes in specific model elements onto numerical values; these metrics enable us to count, aggregate, or classify changes according to their relevance. Metrics map a difference onto a data set which is much smaller and better comprehensible than the complete difference. The visualization of these data in polymetric views supports developers and maintainers in getting a first impression of the changes between two versions of a large model.

Section 2 discusses background information about difference presentation and polymetric views. In Section 3 we define requirements for difference metrics. Section 4 describes the computation of metrics, while Section 5 focuses on their visualization. Finally, we present a prototype and give examples in Sections 6 and 7.

2. BACKGROUND

2.1 Presentation of Differences

Model differences can be presented to the user in various ways. We can differentiate existing methods of presentation:

The **textual presentation** lists the differences in form of plain text or in some structured format, e.g. XML. The output can be very precise and can be arbitrary large. This method of presentation is applicable to all types of differences. However, it is very difficult, or even impossible, to be read by human readers.

Differences can also be displayed in an **interactive list or tree view of local changes**. The view shows all model elements in a list or in a tree if they have a hierarchical structure. Small icons or labels indicate for each element whether it was inserted, deleted, changed. Presentation of moves is difficult. Again this presentation differs from the visual language of the compared models. Furthermore one gets lost in a long list if the inspected system is large. Optionally, the view can be filtered to show changed elements only, however, then one can hardly comprehend the context of changes.

Multiple windows and an interactive list of changes enhance the previous presentation method. Diagrams of each model are shown in separate windows of a modeling tool beside the list of changes. By selecting changes in the list, the corresponding elements are highlighted. Hence, one can better inspect the context of changes and the user's perception of modeling language is kept. However, the list of changes is still very long and does not give a good overview.

In an **integrated parallel display** two diagrams are shown beside each other and corresponding elements can either be identified by their position or by connecting lines. In the former case insertions and deletions lead to spacing and moves cannot be displayed. The presentation method is limited by display size as diagrams often require at lot of space. Synchronized scrolling of both diagrams can help, but hinders overviewing the difference.

The **unified diagram view** displays the computed differences as a unified (i.e. merged without resolving conflicts) diagram in usual shape. Elements which exist only in one of the models are colored differently, e.g. red and green. Updates elements are marked with a third color; unchanged elements remain black. Moves cannot be displayed sufficiently. If one knows the compared models in detail, one can overview all changes easily. However, the approach does not scale up to large models, due to display limitations.

2.2 Polymetric Views

Metrics map software properties onto numerical values. However, the presentation of pure numbers, e.g. in a table, is not sufficient to get an overview. In order to query different aspects, to find outliers, or to inspect the context of properties, the table would have to be sorted in many different ways. Therefore, Lanza has introduced the concept of polymetric views [2]. In this concept, all entities or a selection of entities of a software system such as classes or operations are represented by rectangles. Up to five metrics can be mapped onto the rectangles' height, width, color, and horizontal / vertical position. Hence, software metrics can be visualized in a way that large software systems become accessible; outliers, e.g. classes with an abnormal number of methods, can be located at a glance. Lanza defines various polymetric views for different issues of software analysis, e.g. the system hotspots view to identify extraordinarily large classes. Furthermore, edges can be used in polymetric views to show relations such as inheritance between entities, e.g. in the system complexity view showing a class hierarchy. Different layouters can be chosen to arrange the nodes on a screen.

2.3 Change Metrics

Demeyer et al. [1] have introduced the term *change metrics* in context of detecting software refactorings. However, they compute object-oriented metrics of parts of two versions of a software system and inspect the differences between the produced values subsequently.

This combination of metrics and software changes is significantly different from our work, as they measure changes of software metrics resulting from software changes. We do not compute the difference between metrics, but we compute metrics on differences.

3. REQUIREMENTS

3.1 Graph Representation of Models

For the purpose of computing difference metrics we use a very simple data model. We translate the models to be compared into attributed, directed, typed graphs. Types are assigned to each node and each edge. Edge types can furthermore be divided into references and part-of-relationships that span a tree. A tree edge points from a parent node to its child. In addition, nodes can have attributes. Each attribute consists of a name-value pair.

3.2 Difference Computation

Our approach does not compute differences itself; it rather delegates that task to a separate differencing library. We require the library to provide us with a table of corresponding nodes of both graphs. We furthermore assume the table to contain information whether and how the nodes have been changed. Corresponding nodes that differ in their attribute values are called **updates**. The table entry contains both the old and the new value. Corresponding nodes, whose references are different in the two versions, have a **reference change** with references to both targets. Nodes that have changed their parent nodes get a **move** difference with a reference to the old parent node. Nodes that have no entry in the correspondence table are considered to be **structurally different**, i.e. they are inserted if they exist only in the newer version or deleted if they exist only in the older version. Nodes that show no changes and exist in both versions are tagged as **equal**. Optionally, the correspondence table may contain information about similarities between each corresponding pair of nodes. However, from our point of view the similarity is just a numerical value between 0 and 1; its computation is concern of the differencing library.

4. DIFFERENCE METRICS

Polymetric views display metrics assigned to entities. So it is advisable to compute metrics assigned to the nodes of the model's graph representations. Therefore, we work with a virtual unified graph. It contains one node for each pair of corresponding nodes and additionally the nodes of each graph that are not involved in correspondences. We compute metrics for each node of the unified graph.

In terms of computation of metrics, we can differentiate two groups of metrics. The first group can be seen as non-domain-specific. All metrics of that group can be derived directly from the list of types in the graph, i.e. the metamodel. The second group, so-called domain-specific metrics, take model semantics into account. Hence, they require additional information that exceeds standard metamodel information based on domain-knowledge.

4.1 Non-domain-specific Metrics

4.1.1 Metrics that count changed nodes

Obviously, the different types of changes can be counted for each node. We call the resulting metrics *trivial*. Trivial metrics can be computed for each node and for each change type. An example in class diagrams is the number of updates of a node of type operation; an example in an activity diagram is the number of reference changes of a node of type transition, or the number of structural changes of a node of type activity. These metrics usually return rather small values. Frequently, the value can only be 0 or 1 depending on whether the node was changed or not. Due to the small values, it is advisable to accumulate trivial metrics in the corresponding parent nodes, e.g. to retrieve the number of updated operations for each class. On this basis, we define four kinds of *accumulated metrics*:

Changed children. This metric counts all child nodes that belong to either of the difference types *Update* or *Reference change*. For example, the number of parameters changed in an operation.

Inserted children. This metric counts all child nodes that belong to the change type *Structural change* and exist only in the second graph, as well as all child nodes of change type *Move*, whose final parent is the current node. For example, the number of parameters added to an operation.

Removed children. In contrast to added children, this metric counts all child nodes that belong to the change type *Structural change* and exist only in the first graph, as well as all child nodes of change type *Move*, whose original parent was the current node. For example, the number of parameters removed from an operation.

Unchanged children. The number of child nodes that are marked with *Equal*. For example, the number of parameters of an operation that have remained unchanged.

The values computed on children can be summed up in the parent node and any ancestor node in the part-of hierarchy. For example, one can calculate the number of inserted operations in each package.

4.1.2 Differentiation of updates & reference changes

In order to better differentiate changed elements, we provide more fine-grained metrics counting attribute and reference changes. The metrics are defined by the set of attribute types and reference types existing in the graphs. For each kind we can easily count the number of changes, and the number of unchanged instances respectively.

A node contains only one attribute of each type, but it may contain several references of the same type. However, their number stays rather low. Hence, the metrics counting attribute and reference changes are often either 1 or 0, which indicates whether there is a change or not. Again we accumulate the computed values in ancestor nodes in the part-of hierarchy.

For example, in class models we can count the number of changes of classes' visibilities in a package. In state charts we can count the number of transitions with changed target[1].

4.2 Domain-specific Metrics

The metrics presented so far are non-domain-specific in the sense that they are solely based on the graph structure of the models, their underlying metamodel (most notably part-of relationships), and the structure of a difference as explained in Section 3.2. Metrics defined on this basis can be called "syntactical" because they do not consider the importance of changes, which depends on the semantics of a model type. For example, the following changes are both simple attribute updates: a) the change of the name of a parameter of an operation in a class, b) the change of the visibility of an operation. Obviously, the latter change can have much more significant consequences and is more dangerous. A designer who wants to get an overview of how a model as changed from one version to the next, is mainly interested in the significant changes.

[1]In the UML metamodel the type transition has associations (i.e. references) pointing to source state and target state.

```
<Nodetype name="operation">
  <Attribute name="name" update="medium"/>
  <Attribute name="isAbstract" update="critical"/>
  <Attribute name="ownerScope" update="medium"/>
  <Attribute name="visibility" ordered="true"
      values="private,package,protected,public"
      increase="trivial" decrease="critical"/>
  <Reference name="returns" change="critical"/>
  <NestedType name="parameter" insert="medium"
      remove="medium"/>
</Nodetype>
```

Figure 1: Part of the domain-specific specification of the significance of attribute updates

The significance of a change depends on the semantics of a model type and cannot be deduced from metamodels such as those used in the UML specification – these metamodels define only the syntactical structure of models. Information about the significance of changes must therefore be specified separately; this information can be stored in arbitrary ways, for example as an extension of data which represent the metamodel of a model type.

In addition to the metamodel, the significance of a change also depends on the purpose of difference analysis. Hence, we cannot propose a classification which holds for all purposes. For instance, changes of owner scope might be irrelevant in analysis, but critical in design.

4.2.1 Specification of the Significance of Changes

We propose to classify changes according to the following categories: *critical*, *medium*, and *noncritical*. An example is given in Figure 1; we have used an XML representation of the specification here. We support individual classification of updates for each attribute type of each model element; the same applies to reference changes.

Many metaattributes have ordered domains, for example visibility has the ordered domain (private, protected, package, public). In such cases, the significance of a change can depend on the direction of the change, i.e. whether the value is increased or decreased. We propose to define the significance of attribute updates separately for the increase and decrease. An example is given in Figure 1.

Similarly one can classify insertion, deletion, or move of subelements differentiated by different element types. We support to define different classifications for each element type with respect to its parent element. For example, the insertion and removal of operations in classes can be weighted different; in interfaces in turn these changes can be weighted different again.

4.2.2 Counting Changes of Different Significance

Given a classification we can easily declare a change to be critical, medium, or noncritical. For each model element we can sum up the number of changes of a certain class. Furthermore, we can differentiate between the types of changes.

Thereby, we produce metrics such as number of critical updates or number of noncritical insertions. If we also divide by the types of model elements, we can produce a large number of metrics. Once again the values can be accumulated in parent nodes and other ancestors. For example, we can calculate for each class in a package how many of its operations have been changed critically.

4.3 Similarity Metric

We assume that the differencing algorithm provides us with the similarity $sim(n_{v1}, n_{v2})$ of each pair of corresponding nodes. A similarity value of 0 expresses that the nodes are not similar at all, the value 1 indicates identical properties. Since high values in all other metrics express dissimilarity, we transform this value into the *degree of change* as follows:

$$DoC(n) = 1 - sim(n_{v1}, n_{v2}), \tag{1}$$

4.4 Aggregation of Metrics

In addition to accumulation of values by summation, it is possible to define more specific metrics in grandparent nodes and their ancestors using other aggregation functions. In particular, one can compute for each node the maximum, minimum and average number of changed, added, removed, and unchanged grandchildren. Obviously, children can again be filtered by their type and other metrics can be aggregated similarly.

An example is given by the maximal number of changed parameters in the operations of one class. Another example is the average number of removed attributes in all classes of one package. These aggregated metrics enable a better assessment of differences.

Aggregation of the degree of change is especially interesting. The average degree of change and the maximal degree of change of direct children or grandchildren can give useful hints at the character of changes. E.g. similar average and maximum values indicate uniform changes, while high maximum and low average values indicate changes to specific subelements. Furthermore, the similarity of two elements computed during difference computation can be defined arbitrarily and may aggregate the similarities of all their subelements. For example, the similarity of two classes could be defined by their local properties, the similarities of their attributes, and similarities of their operations. In contrast to that, the aggregated degree of change can focus on one particular element type, e.g. average DoC of operations of a class.

5. POLYMETRIC VIEWS FOR DIFFERENCE VISUALIZATION

In order to assess differences of two model versions, we can define four key aspects for visualization:

1. to clearly point out the location of changes,
2. to measure the amount of changes,
3. to show the significance of changes, and
4. to distinguish the relevance of changes.

The first two aspects lead to a distinction between model elements with changes and elements without changes. If all elements of a certain type are displayed on one screen, one must be able to identify the elements with most changes immediately. Depending on aspect of analysis one large modification does not consequently weigh more than many small changes. Hence, the significance of a change must be recognizable. However, the relevance of a change is often a matter of opinion.

According to these aspects we can define different polymetric views focusing on the changes of different model elements by assigning the values computed by difference metrics to rectangles' properties and choosing appropriate layouters. In contrast to Lanza's definition we also assign a color to the rectangles' frames expressing the difference type of the visualized node itself: Yellow indicates *updates*, green stands for *insertions*, red for *deletions*, blue shows that the node has been *moved*, and magenta expresses *reference changes*. Nodes that belong to multiple difference types, e.g. nodes that have been moved and updated, are framed in cyan. Unchanged nodes, i.e. nodes belonging to difference type *equal*, are left black.

Experience has shown that difference metrics for themselves do not always provide enough information about the relevance of changes. Often developers have to observe the modifications in the context of the model itself, e.g. the number of changes of a class with regard to the usage of this class. Hence, conventional software metrics should always be included in difference metrics analysis. These metrics can be encoded alongside difference metrics in one polymetric view.

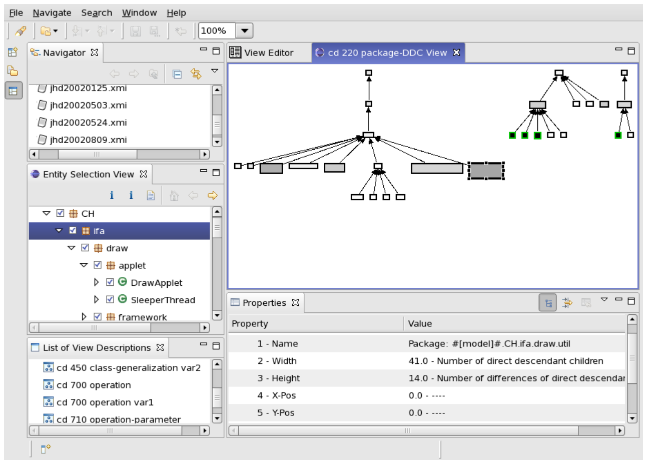
As most polymetric views are applicable to virtually all node types of our graph representation of models, it is impossible to define a fixed set of views. Each software project has individual characteristics that have to be considered when analyzing changes of their models. Hence, we encourage a free definition of user specific views. Some example views are presented in Section 7.

An interesting observation we made is the fact that polymetric views of difference metrics should not be treated as independent from each other, but should be combined in an interactive process instead. The navigation from one view to another based on information gathered from looking at the first view has proven to be beneficial. In particular, the navigation from views representing superordinate model entities such as packages or classes to views representing subordinate entities such as operations or attributes is very useful as we can identify larger entities to be examined in more detail in a subsequent step.

6. TOOL IMPLEMENTATION

We have implemented our approach in a prototype based on Eclipse. A screenshot is shown in Figure 2. For difference computation it uses the SiDiff framework [5], which provides a highly configurable, similarity-based differencing algorithm for graph-based documents. Our tool is structured as follows: In the upper left one can select the model documents to be compared. The documents are loaded and transformed into the graph representation (cf. Section 3.1). The SiDiff framework is called to compare both graphs. Finally, metrics are computed based on the resulting difference.

In the lower left the user can choose between various polymetric views. The visualization itself is presented in the main window. Tool tips and the properties view on the lower right provide additional information such as exact values of metrics, entity names, and the actual changes.



Property	Value
1 - Name	Package: #[model]#.CH.ifa.draw.util
2 - Width	41.0 - Number of direct descendant children
3 - Height	14.0 - Number of differences of direct descendar
4 - X-Pos	0.0 - ----
5 - Y-Pos	0.0 - ----

Figure 2: Difference metrics visualization tool

To focus the analysis on a certain part of the compared models, one can select multiple entities in order to filter the data to be represented. The selection can either be made in a selection tree on the left or by clicking into the view.

The definition of new polymetric views and the manipulation of existing view definitions are handled by a separate view editor. The tool supports integration of standard software metrics into the views, which allows analysis of differences in their context.

7. EXAMPLES

In the following we will present some polymetric views for differences. As explained above one cannot define the ultimate set of views. Nonetheless, we evaluated the difference metrics and their visualization in daily practice by taking the following real scenarios as examples. However, this is not an empirical case study in the strict sense, as we do not measure user perceptions or psychological recognition.

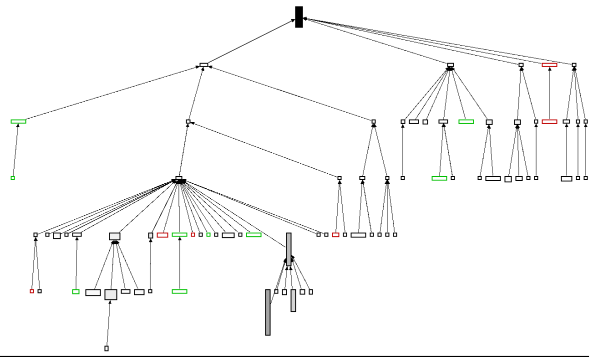
7.1 Differences at a glance

Figure 3 gives an example of a visualization that shows all changes in one view. The view definition is given below the figure.

Here, we compare two class diagrams that describe the implementation of the UML metamodel in the modeling tool Fujaba, namely releases 4.0.1 and 4.1.0. Although, the figure is very small – i.e. the same as having a larger figure with more model elements being displayed – we can clearly identify packages containing differences. The three larger rectangles in the lower center represent the packages `de.uni_paderborn.fujaba.uml` and its subpackages `action` and `unparse`.

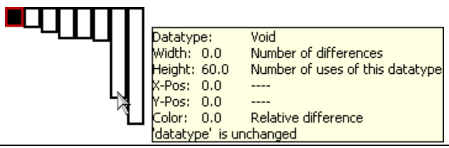
7.2 Usage of datatypes

Figure 4 shows a polymetric view to overview the usage of datatypes. It shows the different datatypes and their amount of usage in two versions of a design model of the data model of a history analysis component of the SiDiff framework. The black colored datatype on the left hand side (ByteArray) has been removed from one version to the next as indicated by its red (gray) frame. However, this change is not critical, since the datatype was no longer used as shown by the rectangle's height.



Nodes:	Package
Edges:	Package → sub-package
Metrics:	· No. of direct children (width) · No. of differences of direct children (height) · Average DoC of direct children (color)
Sorting:	Name

Figure 3: Differences at a glance



Datatype:		Void
Width:	0.0	Number of differences
Height:	60.0	Number of uses of this datatype
X-Pos:	0.0	----
Y-Pos:	0.0	----
Color:	0.0	Relative difference

'datatype' is unchanged

Nodes:	Datatype
Metrics:	· Number of differences (width) · Number of uses of this datatype (height) · Degree of change (color)
Sorting:	Degree of change

Figure 4: Changes of used datatypes

7.3 Navigation between views

As already mentioned, the views for difference metrics support navigation from an overview of all changes downwards to fine-grained difference analysis.

In the following examples we analyzed two versions of the drawing framework JHotDraw, namely from 2001/07/03 and 2001/10/25[2]. In order to get very large and fine-grained models, we enriched the class diagram metamodel by a new element type *statement*, which is subelement of operation. Each statement represents one source code statement and may have a call reference to an operation. The models have been generated by parsing the Java-AST.

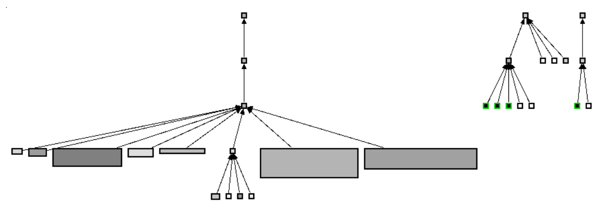
Step 1: Distribution of changes

The view defined in Figure 5 aggregates changes in an average metric (color) and provides an overview of changes per package. The package structure is shown as a tree. The changes of operations and attributes in the packages' classes are encoded in the heights and widths of the package nodes. Here, the classes of the package `CH.ifa.draw.standard`, i.e. the second from the right on the lowest level, show the most changes of attributes and operations. Hence, one should navigate into this package for a more detailed inspection.

Step 2: Relevant changes of classes

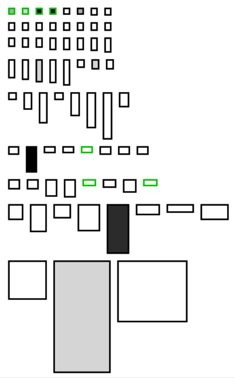
Figure 6 shows the ratio between changed and unchanged statements for each class of the previously selected package `CH.ifa.draw.standard`. Particularly notable are classes with many changes on statements, i.e. the three classes on bottom of the representation (from left to right: `ConnectionTool`, `StandardDrawingView`, and `CompositeFigure`). The number of uses of a class is an indicator for its relevance

[2]There have been no CVS commits between these dates.



Nodes:	Package
Edges:	Package → sub-package
Metrics:	· No. of changes of class operations (width)
	· No. of changes of class attributes (height)
	· Average difference of all children (color)
Sorting:	Name

Figure 5: Packages that contain changes



Nodes:	Class
Metrics:	· Number of unchanged statements (width)
	· Number of changed statements (height)
	· Number of uses of this class (color)
Sorting:	Number of unchanged statements

Figure 6: Statements per class

while the breakdown of statements by classes is required for change localization.

Step 3: Relevant changes of attributes

In the previous example we identified three classes that might contain interesting changes. In Figure 7 we take a deeper look at their attributes by selecting those classes for exclusive inspection. Here, several attributes have been inserted, mainly into the class `StandardDrawingView`; their border is colored green (black). In `ConnectionTool` five local attributes (i.e. the yellow (light gray) marked) have been renamed from `fXXX` to `myXXX`.

7.4 Detection of critical differences

A recurring question in difference analysis is whether critical changes have been made between the versions.

In Figure 8 we are once again inspecting the package `CH.ifa.draw.standard`; this time focusing on attributes. The view for *Critical attribute changes* depicts all attributes inside the selected package. Intuitively, one would examine the attributes on a class-by-class basis; however, our experience has shown that results can be obtained faster using an aggregated view. The six critical changes found (marked by black color) are decreased visibilities; problems might occur if these attributes are used from outside the classes.

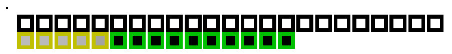
8. CONCLUSION AND FUTURE WORK

Comprehension of differences between two versions of a large model is a challenging task; it is difficult to get a general overview over all changes and to capture the relevant changes in particular.

This paper presents an approach that uses polymetric views as scalable difference visualization. Data basis for visualization are difference metrics that map model differences onto numerical values. Furthermore, the relevance of differences becomes measurable, if the views also contain conventional metrics.

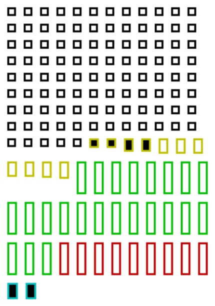
The approach can be applied to virtually all graph-based model documents. A prototype has been implemented based on the difference tool, SiDiff. In excess of this, we can incorporate arbitrary metrics and views. The concept can be realized with other difference algorithms as well, however, the DoC metric requires computation of similarities.

The examples taken from real scenarios already provide an promising insight into the practical effect of metrics-based difference analysis in daily practice. However, research of difference metrics is still in early stage. Empirical case studies which aim at revealing standard sets of metrics and views for popular model types are an open issue. Integration of difference metrics with fine-grained software history analysis [6] is another part of ongoing work.



Nodes:	Attribute
Metrics:	· Number of differences (y-pos)
	· Degree of change (color)
Sorting:	Degree of change

Figure 7: Relevant attribute changes



Nodes:	Attribute
Metrics:	· Number of changes (width)
	· Degree of change (height)
	· Critical Changes (color)
Sorting:	Number of changes

Figure 8: Critical attribute changes

9. REFERENCES

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00*, 2000.

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

[3] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall, 1994.

[4] P. Selonen. A Review of UML Model Comparison Techniques. In Proc. *Nordic Workshop Model Driven Engineering*, pp. 37–57, Sweden, 2007.

[5] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In Proc. *ESEC/FSE 2007*, pp. 295–304, Croatia, 2007.

[6] S. Wenzel, H. Hutter, and U. Kelter. Tracing model elements. In Proc. *ISCM'07*, pp. 104–113, France, 2007.