A Visualization Framework for Task-Oriented Modeling using UML

Christian F.J. Lange*, Martijn A.M. Wijns*, Michel R.V. Chaudron*

* Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, {C.F.J.Lange, M.R.V.Chaudron}@tue.nl, MartijnWijns@gmail.com

Abstract—The UML is a collection of 13 diagram notations to describe different views of a software system. The existing diagram types display model elements and their relations. Software engineering is becoming more and more model-centric, such that software engineers start using UML models for more tasks than just describing the system. Tasks such as analysis or prediction of system properties require additional information such as metrics of the UML model or from external sources, e.g. a version control system. In this paper we identify tasks of model-centric software engineering and information that is required to fulfill these tasks. We propose views to visualize the information to support fulfilling the tasks. This paper reports on industrial case studies and a light-weight user experiment to validate the usefulness of the proposed views that are implemented in our MetricView Evolution tool.

I. INTRODUCTION

The Unified Modeling Language (UML) is the de facto standard for modeling object-oriented systems. The building blocks of UML models are model elements, which are specified in the UML meta model [1]. Classes, methods, objects and messages are examples of UML model elements. These elements represent entities of software programs or relations between them. The UML has 13 diagram types to visualize UML models. Each diagram type describes a projection of a UML model from a certain perspective. Besides the model itself, today considerable amounts of related data are often available such as metrics [2][3], evolution data, documentation and problem reports.

The UML was originally intended to support designing software. However, as today’s software development is becoming more and more model-centric, the UML experiences a much broader use. It is also used to understand systems during maintenance, as a starting point for testing activities, for predictions and other uses. We argue that in model-centric software engineering, views on the available data must be aligned with the tasks in which the views are used (similar to [4]). Most diagram types of the UML were adopted from other software modeling languages and are not aligned with all common tasks in model-centric software development. The purpose of this paper is to propose a framework which is the basis for developing views that support task-oriented modeling. We will identify typical tasks, available (and necessary) data, and we will propose new views and visualization techniques, to improve the use of UML models for practitioners with respect to fulfilling their tasks. Furthermore, we report on industrial case studies and a light-weight user experiment to validate the usefulness of the proposed views that are implemented in our MetricView Evolution tool.

In section II the underlying concepts of the proposed Task-Oriented Modeling framework are described. We propose and discuss new views in Section III. In Section IV we present the validation results from case studies and an experiment and we discuss scalability issues of the views. Section V presents related work and discusses how our work relates to it. Section VI reports conclusions and future work.

II. CONCEPTS OF THE FRAMEWORK

In this section the three underlying concepts of our proposed framework for task-oriented modeling and their relations are described. The three underlying concepts are: Properties, Views and Tasks. The relations between these concepts are illustrated in Fig. 1. Properties are characteristics of UML model elements (e.g. classes, use cases, or sequence diagrams). We will describe an initial overview of properties in this section. Then we define the concept of views which are the basis for visualizing the identified properties. A software developer has to fulfill tasks that have to be performed on model properties and that are supported by views. In the remainder of this section we explain the concepts in more detail and provide examples for each concept. However, we do not claim that the given examples are exhaustive.

Fig. 1. The three underlying concepts and their relations

A. Tasks

We define a task in the context of this work as a unit of work accomplished performed by a software engineer on a software artefact to fulfill a purpose.

As software development is becoming more and more model-centric, the artefact of our interest is the (UML) model. Our proposed framework is a basis for developing model-centric views that support fulfilling the tasks. Hence, the tasks
are the starting point for developing views. Note that our definition of a task does not define a task’s granularity. In this section we give examples of task categories, that combine tasks with a common goal:

**Program understanding.** The need for program understanding (or in this specific context model understanding) can have different reasons. One reason is that a new developer joins an existing team, and needs to understand the system before he is able to make useful contributions. Examples of activities related to this task category are: identifying key classes, identifying which classes implement which functionality, identifying relations between classes and identifying complex interactions.

**Model development.** Creation of models is often an incremental and iterative process including many changes. Either parts of the system are modeled in each step or the system as a whole is modeled from a high abstraction level down to a more detailed level. Examples of activities employed in this type of task are: adding, changing or removing elements.

**Testing.** Testing is used to detect defects in software. A testing task common in model-centric software development is the (automatic) generation of test cases from sequence diagrams.

**Model maintenance.** The process of changing a software system due to corrections, improvements or adjustments is called maintenance. The tasks related to changes in the model belong to this task category. Some activities in which UML models are involved include: extension of a system, bug fixing, handling change requests and performing impact analysis before making a change.

**Quality Evaluation.** As the correction of quality problems is much cheaper at an early stage, i.e. at the modeling stage, than at implementation stage, it is important to evaluate a system’s quality before implementing it. The evaluation of the quality of a model can be performed at several abstraction levels, e.g. separate elements, diagrams or the system as a whole. Besides evaluating a single version of a model one can also investigate multiple versions at once, to detect trends.

**Completeness / Maturity Evaluation.** Related to quality evaluation is the evaluation of the completeness or the maturity of a model. This task category includes analyzing whether a model reflects all requirements and whether the diagrams of the model describe the system completely.

### B. Properties

We define a property in the context of this work as a *directly or indirectly measurable characteristic of a model element.* The set of its properties (or a subset of it) uniquely identifies a model element. Properties belong to the information needed by software engineers to perform tasks. Model elements are the building blocks of UML models. The model element types are defined in the UML meta model. For each model element type, such as class, association, classifier instantiation, use case etc. a number of properties are defined. We identify three different types of properties for model elements:

**Direct Internal.** Those properties of an element that are solely and directly based on information that is present in the model. General examples of this kind of property are the name of an element or the owner. Example properties for a class are its operations, its attributes and its relations to other classes.

**Indirect Internal.** Besides the information that is directly present within the model, we identify properties associated with model elements that can be derived from the model. The information contained in indirect internal properties is derived from the model element itself or related model elements that may also be in different diagrams. For example multi-view metrics as proposed in [5] such as ‘number of use cases per class’ are indirect internal properties that combine information from different diagrams. General examples of this kind of property are metrics and history data. Other examples of indirect internal properties of the model element ‘class’ are the number of methods, the number of instantiations of the class or the complexity of the class (for example based on an associated state diagram).

**External.** A third type of property is based on information from outside the model. This type of properties is ignored by the UML specification and to the best of our knowledge there exist no commercial tools that take external properties of model elements into account. Sources of external properties are other artefacts, such as source code, requirements documents or test documents. Additionally data that is recorded during software development which is traceable to model elements is an external property. For example bug reports can be seen as external properties of classes or can even be traced to use cases. Configuration management systems allow to collect evolution data about a software artefact. For example the number of changes or the number of different developers working on a model element are external properties. As UML is used nowadays not only in design, but also in other phases, such as maintenance, external property data is available for UML models.

### C. Views

We define a view in the context of this work as a *visible projection of a subset of properties of a model’s elements to fulfill a task.* Typically, UML models are visually represented in diagrams. The UML specification [1] defines a variety of diagram types as views on a model from a certain perspective. This specification is only concerned with internal properties and not even all of these properties are viewable in UML diagrams. The relations between model elements in different diagram types are often not intuitively presented by UML. It is, for example, often difficult and tedious to find out on which places a class is instantiated in the model, because this relation is not explicitly present in the diagrams. We argue that in the design of the UML the choice of which properties can be viewed in UML diagrams and the visualization techniques used to represent them are not optimal for common tasks in software engineering.

Views offer visual representations of a model. In a view model element properties are visualized by creating a mapping to visual properties. Examples of these visual properties are: Position (Layout), Size (Width, Height, Depth), Color (Hue, Saturation, Luminance), Shape and Orientation.
III. PROPOSED VIEWS

In this section we propose views that we developed based on the task oriented modeling framework described above. Some of these visualizations are based on existing UML diagrams, others are totally independent. The initial set of ideas was stated in [6]. Most of the proposed views are implemented in the MetricView Evolution tool [7].

A. Context View

The context of a model element consists of all model elements it relates to. The elements of a model are typically scattered over several diagrams. UML diagrams are projections of the entire model, they typically do not contain all model elements. Accordingly, it often occurs that only a limited context of model elements is viewed in one diagram. To fully understand a model element it might be necessary to know its entire context. Therefore we propose the Context View. The model element whose context is viewed in a context view is centered in the diagram. All model elements that are directly related to the particular model element are viewed as a circle around the model element. As the context of a model element is potentially very large and only a specific subset of the context might be necessary for a task it is desirable to filter the context. Two straightforward filtering criteria would be the model element type or the relation type. Figure 2 shows the context view for a class. The context is filtered such that only model elements of the type ‘class’ that are related to the class via a ‘inherits from’ relationship are viewed. The class diagram in the right side of Figure 2 shows only four classes that inherit from the particular class.

Tasks. The Context View is designed to support program understanding tasks. It can for example be used to explore why a class has a specific value. The example in the figure is a class where the metric ‘number of children’ is 28 (for explanations of common object oriented system metrics see [8]). It would be tedious to analyze this outlier by browsing through all diagrams where inheritance relations of the class are viewed. The Context View can also be used for impact analysis (e.g. which classes depend on a particular class?)

Properties. Explicit and implicit relations can be viewed using the Context View (direct and indirect internal).

B. Quality Tree View

Quality models such as [9][10] provide a structure to define what quality means in a given context. The most common approach to create such models is used in so called decompositional quality models. Quality is decomposed in subconcepts such as maintainability and understandability. Each of these subconcepts can in turn be decomposed again. In this way a tree-like structure is constructed which builds a quality model. At the leaves of the tree are metrics. These metrics can be applied to a UML model and the metric values are used to calculate values for each of the nodes in the model.

In the Quality Tree view, the nodes represent the concepts of the quality model and the edges represent the relations between the concepts. Our contribution to the well known concept of quality models is that in the Quality Tree view, color or graphs represent the value of each node. Additionally our proposed Quality Tree View is implemented, such that it interacts with the UML model and supports traceability between nodes in the quality model and elements in the UML model.

As quality models will differ based on the context in which they are used, the Quality Tree View is tailor able to represent several quality models, such as the quality model for UML [11]. This tailoring is possible by changing the structure of the tree, as well as by changing the functions attached to the relations in the tree. Additionally it is possible to change the metrics that produce the input for the quality model.

Tasks. The Quality Tree View is designed to support quality evaluation tasks. This evaluation can start at a high abstraction level by investigating the value of the root node of the tree. From this root node the exploration can continue to lower abstraction levels until the level of metrics is reached.

Properties. Metrics can be indirect or direct internal properties, as well as external properties.

C. MetaView

Figure 3 shows a MetaView in which inter-diagram relations are visualized. The different UML diagrams that are created during the software engineering process offer different levels of abstraction. Typically, starting at a high abstraction level we find the use cases. These use cases are described in more detail by sequence diagrams. In sequence diagrams occur objects representing classes. Classes and their relations to each other are described in class diagrams. The internal behavior of these classes can be described by state machines.

A problem that exists in regular UML is that each of the diagrams is shown separately, this results in the hiding of the relation between different diagrams and model elements. Our proposed solution to these problems is the MetaView. It gives an overview of the diagrams that describe the model and makes it possible to show the relations between (elements on) different diagrams. This last feature allows tracing through the different abstraction levels that the different types of diagrams offer.

Figure 3 shows the four types of elements that take part in this example: A use case, an object that occurs in the sequence diagram describing the use case, the object’s class, and the state diagram describing the class.

Tasks. The MetaView can be applied in program understanding, maintenance, and completeness tasks. Browsing through a model for instance is a program understanding task that is actively supported by this view. Another example is impact prediction for which the visualization of inter-diagram relations can be useful.

Properties. Explicit and implicit relations can be viewed using the Context View (direct and indirect internal).

D. Clustering View

As stated in its description, the MetaView can visualize inter-diagram relations. These relations are mostly visualized at diagram element level. The idea of clustering is to group
them into relations at diagram level, assign a weight to them based on this grouping and use the result to layout the MetaView. This layouting is done by simulating of a mechanical system consisting of springs and rings, so called force-directed placement [12].

Layouting in this way causes diagrams that are strongly related to be positioned close to each other while diagrams that are weakly related or not related at all are positioned further away from each other. Clusters emerge when groups of diagrams are closely related within the group but no relation or only a weak relation is defined with respect to diagrams outside of the group.

Tasks. As Clustering view is a special instance of the MetaView, it can be applied in program understanding, main-
tenance, and completeness tasks.

Properties. Explicit and implicit relations can be viewed using the Context View (direct and indirect internal)

E. MetricView

Figure 4 shows an example of the proposed MetricView, in which three different metrics are visualized on top of a regular class diagram. The idea of MetricView is to combine the existing layout of UML class diagrams with the visualization of metrics and UML models using a set of techniques adopted from geographical information systems (GIS) [6]. Applying metrics to a UML model can result in an overwhelming amount of data. This data is usually presented in tables, such that the software engineer has to make the mapping between metrics values in the table and classes in the UML diagrams manually or in his mind. MetricView supports this problem by integrating the model and metric visualization. MetricView represents the metrics using color, size and/or shape to visualize values.

The tasks supported by this view are: program understanding, quality evaluation and maturity/completeness evaluation. It can for instance help identifying key classes by emphasizing them using metric visualization. It supports the evaluation of metrics on the level of model elements for a single version of a model. These metrics can be both indicators of the quality and of the maturity or completeness of a model.

The metrics for which the values are visualized are direct or indirect internal, or external properties.

F. UML-City View

Figure 5 shows an example UML-City. This view combines the concepts of the MetaView and MetricView. As metric visualization the ‘3D-heightbar’ is used, this visualization shows a box on top of the model element where the height and the color of the box indicate the value of the metric. Low metric values are depicted by flat green boxes while high values are depicted by tall red boxes. Applying this to the MetaView results in a view for which city is a metaphor.

Tasks and Properties. As the UML-City view is a combination of MetaView and MetricView, it also combines their tasks and properties. It gives an overview over the elements with extreme values for a metric.

G. Search and Highlight

In any UML model, especially in large ones, it can be hard to find a specific piece of information. This problem has two main causes. First, the amount of information can be very large and secondly this information is often spread over multiple diagrams. Our solution for tackling these causes uses the MetaView. As stated before, the MetaView gives an overview of the diagrams that describe the model. By adding string search functionality and highlighting the results in the MetaView the user can quickly identify the diagrams in which relevant information is present. Combined with the navigation capabilities that are present in the MetaView this allows for fast searching. The results of search actions are visualized by highlighting the relevant diagrams and diagram elements with a distinct color (e.g. yellow). Additionally relations between elements in the set of result and other elements in the model are drawn. The Search and Highlight function has led to some unintended usage. When highlighting all elements that are related to a specific keyword the tool also shows implicit (not modeled) relations between these elements themselves. Search and Highlight functionality is common in case tools and therefore not proposed by us. It complements the proposed views in a useful manner.

Tasks. Program understanding, model maintenance, and completeness evaluation.

Properties. Names, relations, i.e. direct internal properties.

H. Evolution View

Figure 6 shows the evolution view, in which the two concepts graph and calendar are combined to identify trends. The reason for using a graph is that it is an effective way to visualize the evolution of metric data. The purpose of the evolution view is to enable users to spot trends in the values of quality attributes and/or metrics at multiple abstraction levels. At system level such a graph can be used to give an overview of changes in aggregated data. By combining it with the concept of a calendar, i.e. mapping time on the horizontal axis and values of the vertical axis, and adding color to indicate whether a given value is considered good or bad it becomes a compact and intuitive way to enable the evaluation of the evolution of quality data. The same technique can be applied at diagram and element level to allow for different analysis granularity.

Tasks. Quality evaluation, prediction, monitoring.

Properties. Metrics, i.e. direct and indirect internal and external properties.

IV. VALIDATION AND DISCUSSION

A. Case Studies

To evaluate the proposed views we conducted industrial case studies. We applied our MetricView Evolution tool, which implements the views to industrial UML models and discussed the findings with the developers of the models.

1) Characteristics: We analyzed five UML models from different application domains during the case studies: (A) is a high level architecture of an embedded controller, (B) and (C) stem from the consumer electronics domain, and (D) and (E) are models of a business information system. Figure 7 shows the size of the models in terms of model element types (left) and diagram types (right).

2) Approach and Results: The typical approach for case studies consists of the following activities: First, we analyze the model using our MetricView Evolution tool and other tools to collect metrics data. MetricView Evolution is then used to demonstrate a visualization of a selection of the results of the analysis. The results are presented to the ‘owners’ of the models. During and after the demonstration there is discussion about the findings in the model and the visualization.
3) Findings: The views received positive feedback during the discussions with the model owners. The participants acknowledged in particular that the MetaView and the Context View contributed to increasing the understanding of the overview of the entire model. MetricView, UML-City-View and Quality Tree View were acknowledged to aid in finding quality problems and incompletenesses. Due to the fact that none of the case studies contained more than one version of
the model, the evolution view could not be evaluated in the case studies. The views were regarded as a promising direction for future research.

The largest models (D and E) used to validate MetricView Evolution revealed some scalability issues of the MetricView and the MetaView. For models with a large number of diagrams there is little space available for each individual diagram. Space efficient layouting helps to some extent to reduce the effects of this problem. We discuss the scalability issues of the proposed views in more detail in Section IV-C. It also turned out to be hard to find information about a specific model element if only (part of) the name was known but not the diagrams it occurs in. To assist with this task the search-and-highlight functionality proved to be very helpful. Additionally the search-and-highlight functionality was useful to identify implicit relations, e.g. to find that a state chart that was not related to any class implemented the behavior of a particular class (similarity in names was detected).

B. User Experiment

Additionally we conducted a light-weight user experiment to validate our proposed views. The experiment and its results are described in more detail below.

1) Experimental Design: During the experiment 13 subjects have evaluated the tool. To make the participants familiar with the tool, we first demonstrated its views and intended usage. After this demonstration the participants could experience the tool by performing predefined tasks. The tasks consisted of a number of questions about UML models, that the participants had to answer by using the tool. An example of such a question is “Which class plays the major role in the implementation of the ‘Initialization’ Use Case?” The questions had to be answered about a medium sized model (Case C) and three versions a small model (model A from the case studies and two changed versions). The idea behind these questions is to let the participants explore the various views without forcing them to follow a specified path. This should allow for some creativity and therefore interesting observations. During the evaluation the so called ‘speak-aloud-protocol’ was followed, meaning that the subjects were encouraged to ask questions and make remarks about what they are trying to do and what they were thinking. We collected information about the subjects’ background in a questionnaire to be able to remove subjects without enough relevant experience or skill. A debriefing questionnaire was used to collect the subjects’ evaluation of the views for doing different categories of tasks.

2) Results: The participants were researchers, PhD students and MSc students from our university with a background in software architecture or software visualization. Besides participating in the experiment, none of the participants was involved in this study. The participants’ knowledge of the relevant UML diagram types is good, most have applied it in either academic context or projects with external clients. The most well-known diagram type is the class diagram, closely followed by the sequence diagram. The participants’ knowledge of software development, software visualization and software metrics is sufficient for the experiment, with some noticeable positive exceptions.

The tasks performed during the experiment resulted mostly in correct answers. The effort data identifies two questions, which consumed the largest amount of time. The first question addressed quality evaluation. The reason for the large amount of effort needed is that this task required a large degree of expertise in quality evaluations, which most participants did not have. The second time consuming question addressed finding differences between different versions of a model. Our observations and the analysis of the speak-aloud-protocol show, that this is caused, because most subjects did not use the intended view (the evolution view). We expect that subjects, who are more familiar with the views would perform better for this task.

Several participants suggested a new view, which is a unified class diagram of several versions of the MetaView. Another interesting suggestion we received is related to the Evolution View. One participant made the remark that this view shows great potential as navigational aid and should be the starting
point from which to explore the quality of a model. For this to work better it should be possible to show trends of multiple metrics or quality attributes together in different colors. Also showing the actual values in the Evolution View together with the graph is a suggestion we got from multiple users.

The participants’ ratings of the views for particular tasks are summarized in Figure 8 (correctness) and Figure 9 (efficiency). These histograms show for each task type what percentage of the users found a particular feature useful. The highest rated views with respect to usefulness for correctly performing tasks are: MetaView, Search and Highlight and Context View. With respect to usefulness for fast and efficiently performing tasks the highest rated views are: Search and Highlight, MetaView and Design Smells.

The evaluation of the understandability of the views shows that the Quality Tree View was the easiest to understand. Observations made during the experiment revealed a usability problem of the tool related to activating the Context view. This might be the reason for the fact, that the the Context View received the lowest score for understandability. The observations also showed that once the Context View was activated participants had little problems using it.

With respect to usefulness the participants were asked to mark the feature they expected to be useful for a given task. Understandability and Intuitivity were rated on a Likert scale from 1 to 5 (very useful). For the views Search and Highlight and Context View, figure 10 shows two histograms of the ratings. It can clearly be seen that search functionality was much easier to use for most participants than the Context View. Unlike the context view, searching is a common feature in many tools, which is a likely explanation for this result.

C. Scalability Discussion

Scalability is a common problem in visual formalisms. In this section we discuss the scalability issues of the views that we proposed in this paper and we discuss directions to tackle these issues. The case studies and also some of the figures in this paper show that in particular the MetricView, the MetaView and the UML-City-View (which is based on the previous two views) suffer from scalability issues. A large number of classes causes space problems in the MetricView. We use the existing class diagrams as foundation for the MetricView, hence the MetricView suffers from the same scalability issues than UML class diagrams typically do. As it is the intention of our work to enrich class diagrams with additional information, we do not particularly tackle the scalability problem for UML class diagrams. In the literature exists work that particularly addresses layout optimization of UML class diagrams [13][14]. However, in the implementation of the MetricView, we added zooming functionality to enable both, a high-level perspective and a detailed look at the diagram. Instead of the number of classes, it is the number of diagrams, which can lead to scalability issues for the MetaView and the UML-City-View. We also implemented zooming functionality for these views. The user can identify interesting parts of the diagrams from a high perspective by looking at the relations in the MetaView and at the metrics visualizations in the UML-City-View. After having identified interesting parts, the user can zoom in to have a closer look at the details of the model. Neither in our case studies, nor in the experiment, we have encountered scalability issues for the Context View. The Context View contains by definition only the context of a class, which is usually of a limited size and does not cause scalability issues.

The other views proposed in this paper have no scalability problems. These views rather contribute to tackle the aforementioned scalability issues. For example the Evolution View and the Quality Tree View present metrics information about the model at an aggregate level. By assisting the user in identifying model elements, having outlier values for a metric, these views enable the user to navigate to the identified model elements.

V. Related Work

An approach related to ours is proposed by Kersten and Murphy [15]. Their tool MYLAR presents parts of a source code base which are relevant for a given task. The tool highlights program elements which have a high ‘degree-of-interest’ (DOI) and filters program elements with a low DOI. Their tool is adaptive in the sense, that the DOI-data is collected by the tool during usage. Hence, the interface adapts to a task, which is a difference to our approach, where we provide the user with pre-defined views for specific tasks. So far, their tool only takes internal properties into account. The main difference is, that their tool works for source code and we focus on UML models. Their tool’s views are in fact adaptive interfaces, which change according to the usage pattern

The SDMetrics tool [16] calculates metrics for UML models and presents the data in tables and graphs. The user still has to relate the metrics data to the model elements which are on his mental map. As this mental map often coincides with the layout in the class diagrams, we propose the MetricView to combine metrics representation and the already existing layout of the class diagram. Similar visualizations to represent metrics are proposed by Langelier and Sahraoui [17]. Their visualizations mainly aim at visualizing metrics of source code, therefore they have to create a new layout, whereas we can use the already familiar layout of the UML class diagrams. In [18] several mappings from properties to visual properties, called polymetric views, are explored. The main difference to our work is that polymetric views are general software visualizations aiming at reverse-engineering tasks, while our work consists of UML-based visualizations targeted at various model-centric software engineering tasks. As most related work visualizes additional information such as metrics for source code, the work of Schauer et al. [19] also uses class diagrams as basis. They use color to indicate groups of classes belonging to the same design pattern [20]. Hansen [21] gives an application of the MetricView visualization: he visualized project data mapped to UML classes and packages and reports positive feedback from a case study within the company ABB.
Another direction of related work deals with layout algorithms. For example the work of Eiglsperger [14] and Eichelberger [13] aims at creating layouts for class diagrams, which comply with rules from cognitive psychology. These layout algorithms are in particular useful for reverse engineered models, where no layout exists. An integral idea of MetricView is to keep the existing layout (which could have been generated by a layout algorithm). However the idea of the Context View
is to create a new class diagram only containing a class and its context. Kollmann and Gogolla [22] present a similar idea. They use metrics such as coupling, fan-in and fan-out to determine the set of classes which should be contained in their version of the Context View. In contrast to using metrics, we use the existing structure of relations in the model to determine the Context View.

In our literature study we found several studies about visualizing the evolution of source code, such as Voinea et al. [23] and Langelier and Sahroui [24]. However only little work addresses the evolution of UML models. Xing and Stroufia [25] present the UMLDiff algorithm for automatic detection of structural changes between two class diagrams. The output of the UMLDiff algorithm is a detailed report of structural differences between two versions of UML class diagrams. Our Evolution View presents the trend of metrics of the entire model over several versions. Different from UMLDiff our Evolution View also takes external properties into account. However, opposed to UMLDiff, which is fully automated, our Evolution View is an aid for humans to analyze models.

The main purpose of our MetaView is to give an overview of and navigate through an entire UML model. All UML case tools known to us provide a tree for navigating through the model. Besides this, our literature study did not yield any work related to navigational aids for UML models.

VI. CONCLUSIONS

In this paper we state the problem that existing representations of UML models are not sufficient for common model-centric software engineering tasks. We propose a framework consisting of UML model elements, their properties, and software engineering tasks, that form a basis to develop new views of UML models and related information. Based on this framework we propose eight views to support different tasks. The views are implemented in our MetricView Evolution tool. In industrial case studies and a light-weight user experiment the proposed views were evaluated and we received positive feedback from the users. The users need a learning period to get used to the proposed views. After that period the users understand which views to use for which task and apply the views efficiently. Instead of performing fully-automated analysis of UML models, the presented tool supports the user by presenting information required for a task. Future work is needed to analyze tasks in model-centric software engineering in more detail. We expect that this will lead to a refinement of the task-oriented modeling framework and to the development to more specific views. We currently prepare a controlled experiment to compare our proposed views with the existing UML diagrams.

Acknowledgements. We thank our industrial partners for their collaboration in the case studies and we thank the participants of the experiment for their contribution. We express our thanks to the anonymous reviewers for their very useful comments.