Identifying Performance Challenges in Consistency Preserving View-Based Environments

Lars König [*], Thomas Weber [†]
lars.koenig@kit.edu, thomas.weber@kit.edu
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Abstract

The development of systems, e.g., software systems or cyber-physical systems, becomes more and more complex. Successful approaches for reducing the complexity of their development are view-based model-driven approaches, where developers see only the relevant part of the system for a specific task. As these views still show the same system, their content may be semantically related and changes on one view might require changes on other views to keep them consistent. Although already used in industry, view-based approaches are still not a mature field of research and especially performance is often not a focus in their development. For industrial application, however, performance is crucial, as tools can otherwise become unusable with the extensive sizes of industrial models. To target this problem, we identify possible performance challenges of consistency preserving view-based environments and provide ideas on how to overcome them.

Index terms— consistency preservation, model-driven engineering, performance, view-based development

1 Introduction

Systems have increased in size and complexity over the last decades and still do. To tackle this complexity, the concept of views, initially defined for software development, has been applied to model-driven system development. Views contain only parts of the information of a system and thus reduce the complexity a developer has to deal with. View-based development is a field of active research [8, 9], but the performance of view-based environments is often not the focus. While view-based development is used in the industry today, performance, especially with industry-scale models, remains an open issue. As one measure towards better performance, view-based environments may use model deltas, which represent a model as a sequence of changes. Applied, these changes construct the model state. This enables working in an incremental way, which improves the performance, as, e.g., only changed parts of a model have to be re-analyzed instead of the whole model. As this is only one opportunity to improve performance, we outline challenges in a consistency preserving view-based environment and provide ideas on how to overcome them.

Scope In our paper, we want to focus on the performance of consistency preserving view-based environments, which includes the processing of models and views, as well as reacting to changes to preserve consistency between them. We are not concerned with improving the creation of such an environment by assembling metamodels and specifying views or consistency specifications for them. We also do not consider performance benefits gained by the choice of a specific programming language or hardware platform for their realization. In addition, improvements to the usability, e.g., by changing user interface designs, are not a focus of this paper.

2 Background

In view-based model-driven development, task-specific views allow developers to focus on the part of the system that is relevant for the task [5]. This is especially relevant for the development of large multi-domain systems where many different models are used. Since the different views show parts of the same system, they need to be kept consistent when changes are made. The process of generating changes on views or models as a reaction to changes on other views or models is called *change propagation*.

There are two main approaches to consistency preservation in view-based environments: *synthetic* and *projective* [5]. With the synthetic approach, changes on a view are directly propagated to all related views. In contrast, with the projective approach, views are generated from underlying models, to which the changes are propagated. While the first approach suffers from a quadratic number of consistency rules between the views, the difficulty of the second approach lies in assembling the underlying metamodel.

While a single, redundancy-free metamodel would reduce consistency preservation to the change propagation between the views and the model [1], it is quite

[*]This work was supported by "Kerninformatik am KIT (KiKIT)" funded by the Helmholtz Association (HGF).

[†]This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

complex to define. As a solution, multiple, partially redundant metamodels can be used, requiring explicit consistency preservation rules between them, which propagate changes between the models [8]. For change propagation, both between the views and the models and between the models, model deltas should be used [4]. Instead of using the propagated deltas to update the states of the models, it is also possible to store the models as a sequence of deltas, from which the model state can be reconstructed [6]. Applying changes to a model is then reduced to appending the respective model deltas to the stored delta sequence of the model.

Although processing the changes incrementally through the entire consistency preserving view-based environment avoids the performance problems of state-based model transformations, it introduces a number of performance-critical mechanisms, which we discuss in this paper. Our work is related to the performance of model transformations in general, which has been investigated, e.g., by Amstel et al. [3], who compared the performance of three major model transformation engines.

3 Challenges

We have identified four performance challenges for consistency preserving view-based environments that utilize model deltas. They are part of the challenges of collaborative environments, which are already discussed elsewhere [2]. We therefore focus on challenges specific to the use of model deltas, consistency specifications, view generation, and change propagation.

3.1 Derivation of Model States

Working with model deltas instead of model states provides several benefits, e.g., the ability to use incremental analyses and a reduction in the amount of data needed to update versions. Instead of providing the complete new version, only the changes between the versions have to be applied. While incremental analyses work with model deltas, humans and analyses working on the whole model cannot. Developers could directly interact with models in the form of deltas, but that representation is not useful, as the used delta language removes the domain-specific constructs. This requires the derivation of complete model states.

In order to improve the performance of the creation of model states, we propose to use a model cache, which applies the model deltas to derive a model state. The model cache can be updated with new deltas to reflect changes on the delta sequence. An additional optimization for the generation of model states is the use of *effective change sequences*, i.e., a change sequence that is admissible [9] and minimal, thus there exists no change sequence that is shorter and still admissible. Such a sequence can be computed by deriving changes between a model state and an updated model state [9]. Since a model state may be used to derive a view, this optimization also affects the view generation process.

3.2 Consistency Specifications

While consistency preservation works with model deltas and not model states, it still needs the model state for requests about existing model elements. The existing model elements may influence the result of executing the consistency specifications, e.g., because a corresponding element already exists and is updated instead of being newly created. An example is a Java class that implements a UML class description. A change of the name of the UML class should lead to a change of the name of the Java class. Because of this, model caches are also useful for consistency preservation.

This challenge can be seen as a special case of the challenge mentioned in Subsection 3.1. While the derivation of model states is used for developers working with the consistency preserving environment, this challenge is inherent to the environment itself, independent of its developers. Besides the model state, optimizations regarding the physical representation of the information may also be useful. One example of an optimization could be to store model elements that were created by a consistency specification with a link to the model element that triggered the consistency specification resulting in the model element creation. This link will most likely be used once one of the elements is modified and the corresponding one has to be modified too to keep them consistent. An additional challenge is the orchestration of consistency specifications, which is discussed in [7].

3.3 View Generation

View generation transformations are used to create views on the underlying models of the system on-demand, i.e., when a user of the view-based environment requests it. As a consequence of changes on a view, the underlying models will receive changes as well. If other views were generated from the same models, there might be the need to update these views as well. Possible reasons for this could be that a user of one of these views wants to see the newest state of the models or that there are conflicting changes on the view, which need to be merged with the changes on the models first. In the first case, it would be enough to re-generate the view from the models, discarding the old state. However, in the second case, this would overwrite the changes already made on the view.

While not a good solution for all required cases, also performance-wise the re-generation of views is not preferable. Even for small changes to the underlying models, the entire state of the view would have to be re-computed with possibly expensive transformations. As an example, a developer could have created a view, showing the internal dependencies of a large software system. In the view, they notice an unnecessary dependency and create another view to remove the dependency. To verify that the dependency was successfully removed and continue inspecting the remaining dependencies, the developer updates the view showing the dependencies. If the view update mechanism would have to re-generate the view, a new dependency analysis on the complete system would be necessary.

Instead of re-generating the views entirely, we therefore believe it would be beneficial to update the views incrementally. In the example, it would then be enough to analyze the dependencies of the changed artifact and remove the dependency from the view. The incremental updates on the views can be represented as view deltas, in the same way as model deltas represent changes on models. Of course, this comes with requirements for the specification language of the view generation transformations, which must support the transformation of model deltas to view deltas, in addition to the generation of view states.

The generation of view states is still required for creating a new view without an existing view state. However, since models can be stored as deltas, as described in Section 2, we believe it is possible to use the same techniques for this. One possibility would be to replay the stored model deltas and execute the transformation of the model deltas to view deltas to generate the view state. To avoid having to replay the complete change history of the models, techniques described in Subsection 3.1 can be used to minimize the delta sequence.

3.4 Change Propagation

The consistency preservation interacts with the models in the view-based environment, e.g., by modifying artifacts in the same way as the developer. Additionally, the consistency preservation needs the models to not be changed while it is applied. The reason for this is that the consistency preservation may end up in an undefined state, if the models are updated while the consistency preservation is running. Because of this challenge, the models have to be locked, while the consistency preservation is applied. An example for an undefined state is the modification of the imports of a class by the consistency specification, concurrently to the modification of class names by a developer. The classes, retrieved by name by the consistency preservation, may have their names swapped by the developer, which leads to the correct application of the consistency preservation but with an incorrect result.

This locking can, e.g., be done with a view that is generated for the consistency preservation. This modified view can be automatically merged with the version of the model the consistency preservation was applied to, but may need user interaction to resolve conflicts with changes that have occurred in the meantime.

To avoid unnecessarily locking a whole model or even multiple models, slicing techniques could be applied. The model slices produced by such techniques can then be used to only lock parts of the models, and thus enable the modification of the non-locked model parts concurrently to the application of the consistency specification. This avoids the generation of specific views for the consistency preservation application and additionally reduces the needed resources, because potentially computationally expensive merges become unnecessary.

4 Conclusion

We outlined performance challenges for consistency preserving view-based environments and sketched our ideas to overcome them. The list of challenges we discussed in this paper is by no means complete. Thus, for future work, further challenges should be investigated, and prototypical implementations of our ideas should be evaluated regarding their actual performance improvements. If our ideas turn out to improve performance, they may enable case studies on an industrial scale, thus paving the way for the industrial application of consistency preserving view-based environments.

References

[1] C. Atkinson, D. Stoll, and P. Bostan. "Orthographic software modeling: a practical approach to view-based development". In: *ENASE*. Springer. 2008, pp. 206–219.

[2] I. Mistrík et al. *Collaborative software engineering: challenges and prospects*. Springer, 2010.

[3] M. van Amstel et al. "Performance in Model Transformations: Experiments with ATL and QVT". In: *Theory and Practice of Model Transformations*. Ed. by J. Cabot and E. Visser. Lecture Notes in Computer Science. Springer, 2011, pp. 198–212.

[4] Z. Diskin, Y. Xiong, and K. Czarnecki. "From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case". In: *JOT* 10 (2011), 6:1.

[5] C. Atkinson, C. Tunjic, and T. Moller. "Fundamental Realization Strategies for Multi-view Specification Environments". In: *EDOC*. 2015, pp. 40–49.

[6] A. Yohannis, D. Kolovos, and F. Polack. "Turning models inside out". In: *CEUR Workshop Proceedings 1403*. 2017, pp. 430–434.

[7] J. Gleitze, H. Klare, and E. Burger. "Finding a universal execution strategy for model transformation networks". In: *FASE*. Springer International Publishing Cham. 2021, pp. 87–107.

[8] H. Klare et al. "Enabling consistency in view-based system development—the vitruvius approach". In: *JSS* 171 (2021), p. 110815.

[9] J. W. Wittler, T. Saglam, and T. Kühn. "Evaluating Model Differencing for the Consistency Preservation of State-based Views". In: *JOT* 22.2 (2023), 2:1–14.