Operator-based Viewpoint Definition

Johannes Meier[1], Ruthbetha Kateule[2] and Andreas Winter[1]
[1]*Software Engineering Group, University of Oldenburg, Oldenburg, Germany*
[2]*Computer Science and Engineering, University of Dar es Salaam, Dar es Salaam, Tanzania*
{*meier, winter*}*@se.uni-oldenburg.de, rkateule@udsm.ac.tz*

Keywords: View, Viewpoint, Viewpoint Definition, View Definition, Operator, View-update.

Abstract: With the increase in size, complexity and heterogeneity of software-intensive systems, it becomes harder for single persons to manage such systems in their entirety. Instead, various parts of systems are managed by different stakeholders based on their specific concerns. While the concerns are realized by viewpoints, the conforming views contain projected parts of the system under development. After analyzing existing techniques to develop new viewpoints and views on top of existing systems, this paper presents an approach that defines new viewpoints and views in a coupled way based on operators. This operator-based approach is used to define new viewpoints and views for an existing Module-based description of architectures.

1 MOTIVATION

The growth of size, complexity and heterogeneity of software-intensive systems results in the growth of size and complexity of systems descriptions during development. Thus it becomes more difficult for one person to manage all the related concepts and information of systems. To manage such increasing information required for the description of the whole system, only a *subset* of the existing information should be handled rather than all information together. Having only a subset of such information allows focusing on the currently relevant aspects of the system only while ignoring all other aspects. The currently relevant aspects depend on the tasks of the current person working on the system, on the development phase and on functional building blocks of the system. All these relevant aspects are referred as *concerns* of the different people like developers, designers, testers, project manager, requirements engineers, and so on. Since different people involved with the system represent different *stakeholders* with different concerns, lots of different views on the system are required.

These ideas motivate the use of multiple *viewpoints* and *views*: Following the ISO Standard for Architecture Description 42010:2011 (IEEE, 2011), views describe parts of the system under development that address specific concerns, and help stakeholders with these concerns to work on the parts of the system, that are matched by their concerns. Viewpoints determine, which information of the system should be contained in conforming views. In this paper, views are realized as models and visualized as UML object diagrams, while viewpoints are realized as metamodels and visualized as UML class diagrams.

To assemble and manage several views showing parts of the same underlying system, (Atkinson et al., 2009) presented the idea of *Single Underlying Model*s (SUM) which contains all information of the whole system under development in an integrated and consistent way (Margaria and Steffen, 2009). The stakeholders do not work directly on the SUM, but use views which are projected from the SUM. Using views, the stakeholders can work "in isolation" on only the currently relevant aspects. After finishing their work, changes made in the views are transmitted into the SUM automatically to keep it up-to-date. Since all views are projected from the SUM, the views remain consistent to each other. The SUM is explicitly realized as model conforming to the explicit *Single Underlying MetaModel* (SUMM).

Therefore, the goal of this paper is to enable new views projected from an existing SUM: Following the SUM idea, this paper presents an operator-based approach to define new viewpoints on top of a SUMM in Section 4, along a running example introduced in Section 2. Section 3 analyzes related work, derives challenges for view definitions and points to the contributions of the new approach. Section 5 presents more applications to demonstrate the applicability of the new approach. The contributions of this paper and its approach are summarized in Section 6.

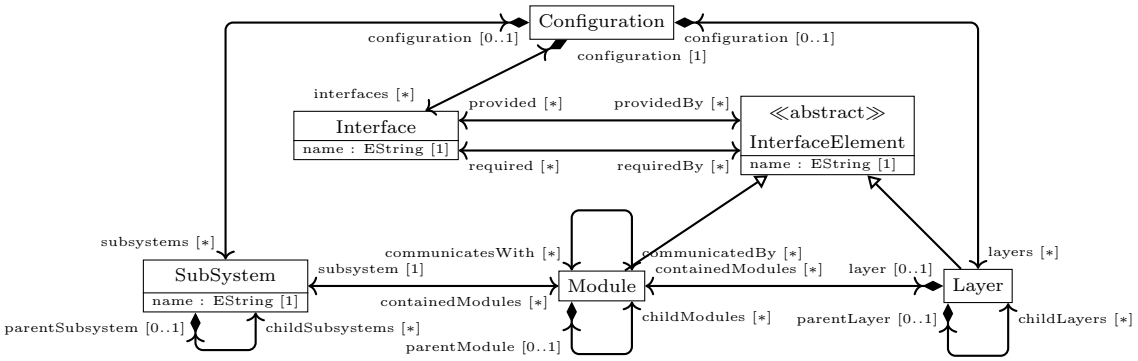

Figure 1: Metamodel describing architectures of SEISs, used as SUMM in the running example to derive new viewpoints.

2 VIEW(POINT)S ON MODULES

As running example for this paper, an existing module-based description for architectures of *Smart Environmental Information Systems* (SEISs) (Kateule and Winter, 2016), adopted from (Hofmeister et al., 2000), is chosen and treated as SUMM. SEISs refer to those systems which incorporate the sensing and actuating techniques to observe, analyze and control certain environmental phenomena such as forest fires,flood,air pollution,landslides.The *Module description* addresses the concerns of system architects, project manager and development experts which are related to how the SEIS is technically realized. It is treated as SUMM in this paper, since it contains different aspects which are usable for new (sub) viewpoints and organized in an integrated way.

The used metamodel for this Module description is shown in Figure 1 which describes the essential elements for architectures of SEIS, mainly software implementation modules, their interfaces and relationships required for the construction of modules. A module represents a set of functionalities of SEISs that can be realized and provided as a service. The modules of SEISs are organized into two orthogonal structures such as decomposition and layering. The decomposition encompasses how the system is decomposed into subsystems and modules, while layering demonstrates how the modules are assigned to layers. Subsystems can contain other subsystems or modules. Modules can interact with each other directly, if they belong to the same layer. The interaction of modules across different layers is facilitated by both required and provided interfaces.

On instance level, a very small example SEIS is used as SUM in this paper, shown in Figure 2 as object diagram: The whole SEIS is represented by one subsystem named "SEIS System" (`sub0`), which is divided into two subsystems for the "Sensor-Subsystem" (`sub2`) and the "ControlCentre" (`sub1`). The sensor-subsystem measures physical environment events, i.e., temperature, humidity etc.. The control center refers to the subsystem, which integrates and analyses information gathered by sensor-subsystem to determine the likelihood of certain environmental phenomena. The system is structured technically into two layers: data management (`l2`) and application logic (`l1`). Inside the application layer, the "Analyzer" module (`m2`) analyses data collected by sensor-subsystem to detect the environmental phenomena. The required data are provided directly by the "Acquisition" module (`m1`), which are stored in the "DataBase" module (`m3`). The acquisition module obtains data from the real world by allowing the collection of data from either digital or analog sensors. The database module is used to store data. Since these two modules `m1` and `m3` are located in different layers, the communication between them uses the "DataAccess" interface (`i1`). The `root` object is required only for technical reasons introduced by the underlying Eclipse Modeling Framework (EMF).

The decomposition of the modules in sub-modules and their grouping by layers and subsystems is the main goal of the Module description and helps mainly *architects* to design the structure of the system under development. Since the subsystem can be used to structure the system and its modules regarding functionality, subsystems are also helpful for the *project manager*: To measure the progress of the development project and to estimate the required development effort for subsystems, the project manager needs the mapping of the subsystems with the layers, but without being overburdened by the more detailed modules. Therefore, the project manager wants to use a *new viewpoint* called "Intersections" (Figure 3) with only the following concepts: (1) subsystems with their decomposition (2) layers with their decomposition (3) information about systems having intersec-

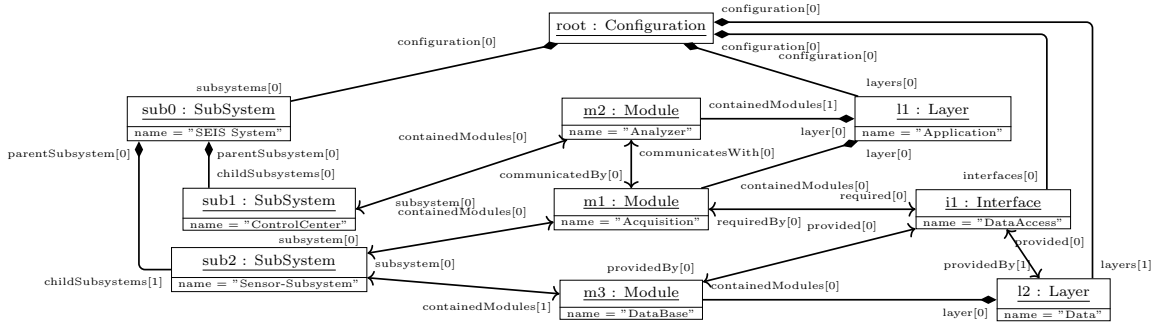

Figure 2: Model describing the architecture of a small SEIS, used as SUM in the running example to derive new views.

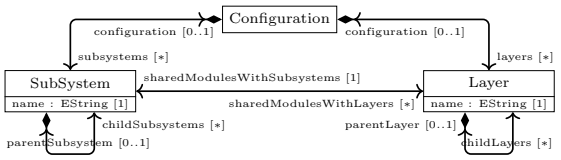

Figure 3: Intersections viewpoint, subset of Figure 1.

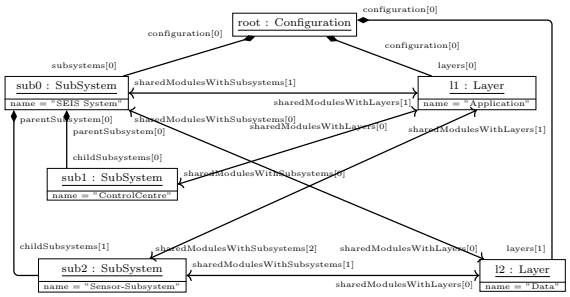

Figure 4: Intersections view, subset of Figure 2.

tions with layers regarding the same modules. Instead of the complex model shown in Figure 2, the project manager wants to work only with the model shown in Figure 4, which contains all relevant data, but none of the superfluous data, thus reducing unnecessary complexity. This Intersections viewpoint is used as the running example in the paper. Some more examples for new viewpoints are given in Section 5.

3 RELATED WORK

To realize the SUM idea as motivated in Section 1, (Meier et al., 2019) discusses and compares some existing SUM approaches, which contribute related work also on how to define new viewpoints on top of an existing SUMM: The "Orthographic Software Modeling" approach (*OSM*) (Atkinson et al., 2009) realizes the SUM idea by storing all information about the system under development inside one explicit model, the SUM, without any redundancies and dependencies, and a conforming SUMM. All views are projected on-demand from the SUM by executing a model transformation. Since OSM supports multi-level modeling, DeepATL (Atkinson et al., 2013) as an extension of ATL with multi-level modeling is used. To propagate changes back into the SUM, trace links between view and SUM are used (Tunjic and Atkinson, 2015).

Another approach for the creation of new views is realized with *ModelJoin* (Burger et al., 2014), which takes several (meta)models as input and provides the resulting output model as view including its metamodel to the stakeholder. ModelJoin is a textual DSL, whose execution results in model, metamodel and model transformations for the new view(point).

The ModelJoin approach was developed in the frame of the *Vitruvius* approach (Kramer et al., 2013) as another SUM approach. Vitruvius realizes the SUM idea by combining already existing metamodels and models internally as modular SUM and by providing views to the stakeholders as in the original SUM idea. These views are combined from the internal models using ModelJoin.

Another SUM approach is the "Role-oriented Single Underlying Model" (*RSUM*) (Werner and Assmann, 2018), which combines existing metamodels and models into one explicit SUM(M) by adding role-based associations between them. New viewpoints are realized by reusing the ModelJoin approach.

Independent from SUM approaches, generic *model transformations* (Mens and Van Gorp, 2006; Jakumeit et al., 2014) are used to create views: The existing SUM and SUMM are used as source model and source metamodel, while the output model is provided for the stakeholder as the view conforming to the output metamodel which is the viewpoint. Since the output metamodel often has to be defined before writing the model transformation, these two artifacts have to be kept consistent to each other: Changes in the target metamodel require changes in the model transformation and vice versa. Since this is maintained often manually, the coupled evolution of target metamodel and model transformation is source for possible inconsistencies. This counts also for OSM.

The technical relationship between SUM and view is often described by one whole, comprehensive transformation. This allows to write them compactly, but hinders the reuse of parts of the written transformation, e.g. for the definition of similar viewpoints. Since such transformations are executed en bloc, debugging the transformation is difficult, since stable intermediate states are missing. This counts also for OSM and ModelJoin.

Another approach to define viewpoints on top of a metamodel is presented in (Cicchetti et al., 2011): For all meta-classes and their attributes and associations in the metamodel can be decided, whether they should be part of the new viewpoint or not. Although this simplifies the definition of the viewpoint and the back-propagation of changes, the stakeholder as a user of the new view is supported with views only in moderately quality and usability. Because the selection of existing parts of the metamodel is possible only without restructurings and adding computed values, the view(point) cannot be tailored specifically to the concerns of the stakeholders, but inherits the quality of the initial (meta)model. This counts also for model pruning approaches like (Sen et al., 2009).

The approach presented in Section 4 targets mainly transformations between SUMM and viewpoint divided into reusable units. This targets to specify new viewpoints easier by reuse, an iterative procedure and stable intermediate steps for debugging. Additional design goals are viewpoints which differ from the initial structure and are kept in sync with the transformations. The work of this paper extends MoConseMI as another SUM approach, showing that is possible not only to integrate several data sources into a new SUM(M) (Meier and Winter, 2018), but also to derive new viewpoints from an existing SUMMs.

4 VIEWPOINT DEFINITION APPROACH

The main idea of the new approach to define viewpoints and views is to split up the transformations into a sequence of small transformations. Each of the small transformations is realized by an *operator* (Section 4.1), which should be reusable to define new viewpoints easier. Another design goal is to keep the model transformations between SUM and view consistent to the final viewpoint.

4.1 Operators

Each operator realizes a small part of the whole transformation between SUM(M) and view(point): Input for each operator is the current metamodel and conforming model. After a small change in the metamodel and conforming changes in the model, these changed metamodel and model are the output of the operator and serve as input for the next operator. This design allows the creation of chains of operator forming arbitrarily long transformations.

The third row of Figure 5 shows the chain of operators configured for the transformation of the SUM(M) into the Intersections view(point): Operators are depicted along edges, while the connected nodes before and after are input or output for this operator. As example, the operator `AddDeleteAssociation` takes the SUM(M) as input and adds a new association to the SUMM, which results in a changed metamodel for ⓰ as output. This operator was chosen to store intersections between subsystems and layers for the project manager. The operator was configured to calculate and store intersections by linking each subsystem with those layers which contains at least one module which is part of the subsystem.

To be reusable, each operator can be configured regarding its changes in the metamodel: As an example, the operator `AddDeleteAssociation` provides metamodel decisions for the source and target class which should be connected by the new association. Complementary, each operator changes the model to keep it conform to the changed metamodel and fulfills the required model-co-evolution. This operator-based coupled evolution of metamodels and models is introduced by (Herrmannsdoerfer et al., 2011) and is extended here: Since the model-co-evolution has some degrees of freedom how to change the model to keep it conform to the changed metamodel, each operator can define model decisions to control these degrees of freedom: As an example, the operator `AddDeleteAssociation` allows adding arbitrary links in the model for the new association in the metamodel. By this design, the main functionality of the operator `AddDeleteAssociation` to create new associations can be realized once and provided as reusable operator. All operators are provided in a *reusable library*. An incomplete version of it can be found in (Meier and Winter, 2018).

In contrast to model transformation chains (Lúcio et al., 2013), instead of whole transformations, only small recurring transformation patterns are reused, while their configuration is project-specific. As other action-based approaches managing models like (Mosser and Blay-Fornarino, 2013), only the model level is addressed, while the metamodels are also required to get the new viewpoints.

The orchestration of operators like in Figure 5 is done only once and manually by selecting opera-

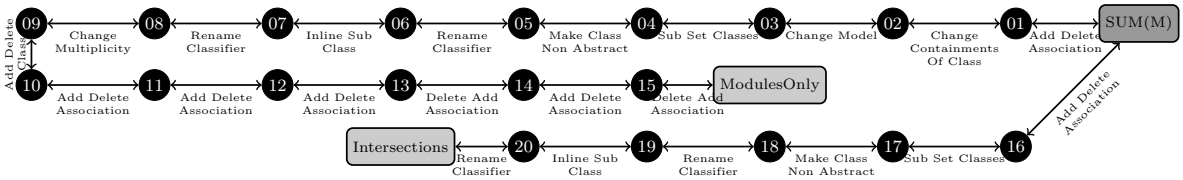

Figure 5: Chains of configured operators to define two new view(point)s Intersections and ModulesOnly on the SUM(M).

tors from the library, configuring them and combining them as chain of operators. Afterwards, this orchestration is stable and can be executed several times automatically: Executing the chain of operators with the SUM(M) as starting point results in the projected viewpoint and the projected view. Since the viewpoint is generated out of the operator chain, it is always *in sync* with the configured transformation. In contrast to some generic model transformation languages, the operators change metamodel and model *in-place* to avoid copying the whole model for each operator, which improves performance.

After calculating and storing the intersections in ⑯ for the running example, the modules and interfaces are removed by using the `SubSetClasses` operators ⑰, which is described with some more details in Section 4.3. Finally, the meta-classes `InterfaceElement` and `Layer` should be combined to ease the new view(point): This is done by the operator `InlineSubClass` ⑲→⑳, while the remaining operators `RenameClassifier` and `MakeClassNonAbstract` support that.

This chain of operators also show, that new viewpoints can be defined in iterative way: The model and metamodel after calculating the intersections can be analyzed at ⑯, before the following operators reduce the existing information at ⑰…⑳. This helps to analyze and debug the single steps of the transformation and allows development in steps over time.

After projecting the Intersections view from the SUM, the project manager can read and change the Intersections view. This leads to the challenge to propagate changes in the view back into the SUM, which is discussed in the next Section 4.2.

4.2 View-update-Problem

The Intersections view allows the project manager not only to analyze possible intersections, but also to change things, e.g. to rename an existing layer. To keep the SUM up-to-date, these changes have to be propagated from the view to the SUM. This leads to the known view-update problem (Bancilhon and Spyratos, 1981). In related approaches like (Cicchetti et al., 2011), which uses a subset of the SUMM without any structural changes or refactorings, changes made in the view can be applied directly to the SUM. Since the approach of this paper supports changes in the new viewpoint (Figure 3) compared to the SUMM (Figure 1), a more complex solution is required here.

The main idea to update the SUM is to execute the configured chain of operators in inverse direction to transform the (changed) view into the (changed) SUM. For that, each operator is combined with an inverse operator: Two operators *A* and *B* are inverse to each other, if the metamodel after applying *A* and *B* is the same as before executing these two operators. In other words, *B* reverts the metamodel changes that are done by *A*. Therefore, the presented operator `AddDeleteAssociation` represents the operator `AddAssociation` combined with its inverse operator `DeleteAssociation`. Each operator is combined with an inverse operator. Therefore, translating the viewpoint back into the SUMM is ensured automatically by the design of the operators.

Looking at the model level, after executing the operator chain in the inverse direction it is also ensured automatically, that the resulting SUM is still conforming to the SUMM, because the model-co-evolution done by the operators is fixed accordingly to the metamodel changes. More interesting are the degrees of freedom provided by the model decisions: Model changes triggered by model decisions of an operator has to be inverted by the inverse operator. For the `AddDeleteAssociation` operator, this is easy, since all added links are deleted by the inverse operator as part of the regular model-co-evolution. More complex is the `Inline/ExtractSubClass` operator, which is also used in the running example ⑲→⑳: After inlining a sub-class (`InlineSubClass`), the inverse operator extracts the sub-class again (`ExtractSubClass`). On the model level, the instances migrated to the super-class have to be migrated back to the recreated sub-class again. The selection of the instances of the super-class to migrate to instances of the sub-class is controlled by a model decision. This model decision has to be carefully decided for the current situation to match only the instances migrated by `InlineSubClass` previously.

Finally, the view-update problem is solved once and manually by configuring the model decisions ap-

propriately to realize the wanted back-propagation of changes. The operator's design ensures the metamodel level and the general model-co-evolution, but the semantic details have to be ensured manually by the coordinated and consistent configuration of the operators. Summarizing, changes in the views are propagated into the SUM by transforming the changed view back into the changed SUM.

4.3 Information Loss

Since stakeholders want to use views only with information tailored to their specific needs, only a subset of the information contained in the SUM is shown in views. Since using a view should be done "in isolation" from all other views and the SUM, the approach of this paper removes information step-wisely during the transformation of the SUM to the view, done by the operator chain. To transfer the changed view back into the SUM, the operator chain is executed in the inverse direction, as explained in Section 4.2. Since the operators work in-place, the operators have to *restore previously removed information* again to *prevent a loss of information in the updated SUM*. Three different solutions are available to realize that.

The *first solution* is to use operators like `Add/DeleteClass`, `Add/DelteAssociation` und `Add/DeleteAttribute` to remove unused information. During the execution of all operators by the supporting framework, all model changes induced by operators are recorded and stored. For the mentioned operators, the removed information is stored in the form of model changes. After executing the inverse operator to recreate the SUM, a complex algorithm currently under development is used to invert and apply the stored model changes again in order to recreate the removed information. Since this mechanism is provided in a generic way for all operators by the framework and can be requested if needed, this solution should be preferred. In the running example, the used operator `SubSetClasses` ⑯→⑰ removes the two meta-classes `Module` and `Interface`. `SubSetClasses` is a composite operator and uses several instances of `DeleteAddClass` internally.

While the first solution is a post-treatment for the execution of operators not visible to them, the *second solution* can be used directly inside the implementation and configuration of operators: For each combination of a configured operator with its configured inverse operator a map to store arbitrary information is available during execution. The map can be used to read and write information into it during all executions of the operator and its inverse operator. As example, this option is used by the regular model-co-evolution of the `MergeClasses` operator to remember which instances were merged during the last execution to merge them now again. Additionally, this "history map" can be used in configured model decisions to remember some important information. This can make sense for example in the `ChangeModel` operator, which allows arbitrary changes in the model, while the metamodel remains unchanged. Together with the history map, this operator gives full flexibility to realize arbitrary views with solving the problems of view-updates and information loss. This second solution is only controlled by the operators and its model decisions, and therefore requires careful use.

The *third solution* is a special case supported by special operators to split a model into two independent models, while one of the split models is provided as output and the other one is stored inside the operator to be used by the inverse operator to add it again. Because of this restricted design, this operator is normally not used for defining new viewpoints.

5 APPLICATION

After presenting the approach for operator-based coupled definition of views and viewpoints, Section 5.1 shows the final result of the running example. Section 5.2 discusses a completely new viewpoint, with a variant in Section 5.3.

5.1 Viewpoint Intersections

As a running example, the Intersections viewpoint help project managers to detect overlaps of subsystems and layers regarding the same modules. This information can be helpful to identify possible conflicts, to plan resources and to bridge functional units (the subsystems) with technical units (the layers) as an overview. After configuring the operators ⑯ until ⑳ in Figure 5 to define the Intersections viewpoint (Figure 3) on top of the SUMM, they help to project Intersection views like in Figure 4 from the SUM and to transfer changes back into the SUM afterwards.

A variant of this viewpoint could add the number of intersecting modules between subsystems and layers, since in the current design links only indicate, that there is at least one intersecting module (Figure 3).

5.2 Viewpoint ModulesOnly

Looking into more detail, modules with their communication are also interesting for *developers* who realize different communications kinds between modules. Since they are not that interested in the structuring of

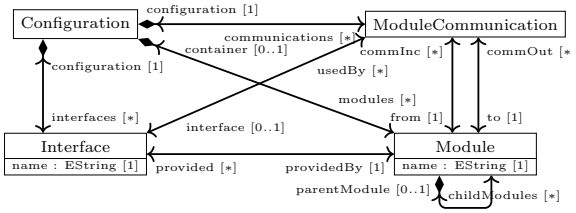

Figure 6: ModulesOnly viewpoint, subset of Figure 1.

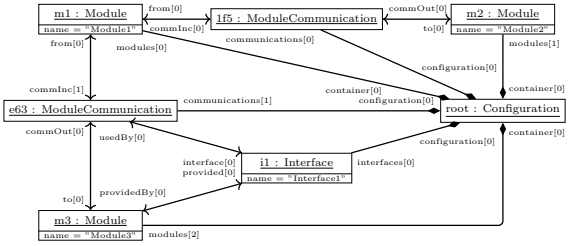

Figure 7: ModulesOnly view, subset of Figure 2.

modules with layers and subsystems on a more conceptual level, they want to use another *new viewpoint* called "ModulesOnly" with only the following concepts (Figure 6): (1) modules with their decomposition, (2) all communications between pairs of modules, (3) interfaces used by communicating modules. Instead of Figure 2, the developers want to work only with the information shown in Figure 7, which contain only the wanted information as view on the SUM.

The chain of configured operators to realize the ModulesOnly viewpoint is shown in the first two rows of Figure 5: The first two operators ⓫ and ⓬ are required as technical precondition and change the EMF containment of the modules from the layers to the configuration, since the layers will be removed later. Now all subsystems and layers are removed by `SubSetClasse` ⓭→⓮, since they are not relevant for the developers. The previous operator `ChangeModel` is required to fulfill the view-update problem: If new modules are created in the ModulesOnly view, even after restoring subsystems and layers as described in Section 4.3, new modules are not related to any subsystem, which is a conflict to the SUMM (Figure 1). To fix this, `ChangeModel` is configured to link each module with one random subsystem, if there is no linked subsystem.

The following four operators ⓯ until ⓲ are used to combine the meta-classes `Module` and `InterfaceElement`, since the other sub-class `Layer` was removed before. Since this is very similar to the combination of `Layer` and `InterfaceElement` for the Intersections viewpoint (⓲ until Intersections, described in Section 4.1), the same operators are taken from the library and reused with slightly different configurations. This shows, that the operators are reusable and that there is a need for reusable parts of the transformations between SUM and new views.

Operator ⓳ ensures that each interface is provided by exactly one module. The operators ⓴ until ㉔ migrate the direct communication of modules via the looping association `communicatesWith` in the SUMM to the more structured representation using the new `ModuleCommunication` meta-class. The operators ⓴ until ㉓ creates the information first, the operator ㉔ removes the old information afterwards. Since this transformation is done without any information loss, there is no need for the strategies to prevent information loss described in Section 4.3. Again, this step-wise procedure with mapping single operators to single purposes shows the support for iterative development and debugging of the operator chain.

The last two operators realize the communication between modules via an interface by the same strategy as for the direct communication: The first operator enables `ModuleCommunication` to use an `Interface` and adds the interface-based communications on model level. The last operator removes the old representation and all links of the `Module.required` association.

Again, the operators ㉑ until ㉕ show, that some functionality like adding a new association is required several times for different purpose. Therefore, operators like `AddAssociations` help to develop transformations between SUM and view faster and easier, because only variation points like source and target class for the new association have to be specified, while the details of the transformation can be skipped and are internal details of the reusable operator. In contrast to pre-defined transformations, operators like `ChangeModel` provide full flexibility to realize specific needs for new view(point)s, as used for ⓭.

Because the Intersections viewpoint is similar to the original SUMM, six operators are enough for the transformation. Since the ModulesOnly viewpoint describes the communication between modules completely different as in the SUMM, 16 operators are required: Growing complexity of viewpoints can be handled and scaled by longer chains of operators.

5.3 Viewpoint LayersOnly

The ModulesOnly viewpoint helps developers to structure the system in detail (Section 5.2). In contrast to that, software architects are more interested in layers as the general structure of systems. This counts in particular for huge systems with lots of layers and an unmanageable amount of modules inside them.

Therefore, similar to the ModulesOnly viewpoint, a LayersOnly viewpoint can be specified to sup-

port the architect in focusing on only the layers of the system under development: Additional to the layers, also their communication with each other is of interest and can be described similarly like for modules in Figure 6. One difference is, that each `LayerCommunication` always uses exactly one `Interface`, since the communication of layers is allowed only using interfaces. Again a chain of operators can be configured to describe the transformation of the SUM(M) into the LayersOnly view(point), similar to the chain of operators for the ModulesOnly viewpoint shown in Figure 5. Most operators used for the ModulesOnly viewpoint are reused for the LayersOnly viewpoint, but with a slightly different configuration. In the end, the ModulesOnly viewpoint and the LayersOnly viewpoint complement each other by focusing on different levels of granularity.

6 CONCLUSION

Summarizing, this paper contributes a new approach to define new viewpoints and views by using operators. Since the operators split the whole transformation between SUM(M) and view(point) into parts, the operators provided as library are designed to be reusable and generic by using metamodel and model decisions. The presented exemplary chains of operators are showing their reusability for different new viewpoints on top of already existing SUMMs.

The chain of configured operators allows developing new viewpoints in an iterative way by adding more and more operators to improve and adapt the new viewpoint step-wisely to the needs of the stakeholders. These steps ease also debugging purposes, since the current (meta)model can be reviewed after each operator. The framework to realize the operators and to enable the definition of viewpoints is under development and provides support to overcome the challenges view-update and of preventing information loss, but leaves room for individual solutions during the manual configuration of operators.

Compared to approaches from the related work in Section 3, this approach keeps the viewpoint and the transformation to project the views in sync, because the viewpoint is generated during the execution of the operator chain. The presented approach does not use explicit traces between SUM and new view and allows specifying viewpoints, which differ highly from the structure of the initial SUMM.

REFERENCES

Atkinson, C., Gerbig, R., and Tunjic, C. V. (2013). Enhancing classic transformation languages to support multi-level modeling. *SoSyM*, 1–22.

Atkinson, C., Stoll, D., and Bostan, P. (2009). Supporting View-Based Development through Orthographic Software Modeling. *ENASE*, 71–86.

Bancilhon, F. and Spyratos, N. (1981). Update semantics of relational views. *TODS*, 6(4):557–575.

Burger, E., Henss, J., Küster, M., Kruse, S., and Happe, L. (2014). View-based model-driven software development with ModelJoin. *Software & Systems Modeling*.

Cicchetti, A., Ciccozzi, F., and Leveque, T. (2011). A hybrid approach for multi-view modeling. *Recent Advances in Multi-paradigm Modeling*, 50.

Herrmannsdoerfer, M., Vermolen, S. D., and Wachsmuth, G. (2011). An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. *Software Language Engineering*, LNCS 6563:163–182.

Hofmeister, C., Nord, R., and Soni, D. (2000). *Applied Software Architecture*. Addison-Wesley, Boston.

IEEE (2011). ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description. 2011(March):1–46.

Jakumeit, E., Buchwald, et al. (2014). A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, 85(PART A):41–99.

Kateule, R. and Winter, A. (2016). Viewpoints for sensor based environmental information systems. In *EnviroInfo*, 211–217.

Kramer, M. E., Burger, E., and Langhammer, M. (2013). View-centric engineering with synchronized heterogeneous models. *1st VAO '13*, 1–6.

Lúcio, L., Mustafiz, S., Denil, J., Vangheluwe, H., and Jukss, M. (2013). FTG+PM: An Integrated Framework for Investigating Model Transformation Chains. In *LNCS*, volume 7916, 182–202.

Margaria, T. and Steffen, B. (2009). The One-Thing-Approach. In *Handbook of Research on Business Process Modeling*, volume 49, 1–26. IGI Global.

Meier, J., Klare, H., Tunjic, C., Atkinson, C., Burger, E., Reussner, R., and Winter, A. (2019). Single Underlying Models for Projectional, Multi-View Environments. *MODELSWARD*, SCITEPRESS.

Meier, J. and Winter, A. (2018). Model Consistency ensured by Metamodel Integration. *6th GEMOC, co-located with MODELS 2018*.

Mens, T. and Van Gorp, P. (2006). A taxonomy of model transformation. *ENTCS*, 152(1-2):125–142.

Mosser, S. and Blay-Fornarino, M. (2013). Adore, a logical meta-model supporting business process evolution. *Science of Computer Programming*, 78(8).

Sen, S., Moha, N., Baudry, B., and Jézéquel, J.-M. (2009). Meta-model Pruning. In *LNCS*, volume 5795, 32–46.

Tunjic, C. and Atkinson, C. (2015). Synchronization of Projective Views on a Single-Underlying-Model. *VAO*.

Werner, C. and Assmann, U. (2018). Model Synchronization with the Role-oriented Single Underlying Model. *MODELS 2018 Workshops*, 2245:62–71.