Model Synchronization with the Role-oriented Single Underlying Model

Christopher Werner
Technische Universität Dresden
christopher.werner@tu-dresden.de

Uwe Aßmann
Technische Universität Dresden
uwe.assmann@tu-dresden.de

ABSTRACT

Models@runtime (M@RT) suffer from information fragmentation across heterogeneous runtime models conforming to several metamodels that are filled with information from a source model. This creates an amount of related models with replicated information. In contrast, a recent approach to software engineering utilizes Single Underlying Model (SUM) to generate views of a software system on demand. The views represent fragments specifying required information for model instances. This results in an increasingly complex source model which is consistent from scratch and views with relations between each other that require complex processes to maintain consistency. Thus, to hold the views and models consistent at runtime, an intuitive approach for the creation of an adaptable SUM should be introduced that permits runtime synchronization and adaptation of views. We propose utilizing the concept of roles in the domain of SUM-based software engineering and M@RT. Based on existing work in the area of role modeling, we present a Role-oriented Single Underlying Model approach that provides a natural way to create views from a running model. We show how the role concept simplifies the creation of runtime models as views from a SUM, provide an incremental view update approach, and introduce a flexible adaptation mechanism. Finally, we illustrate our approach with two example views to explain the benefits of a role-oriented SUM.

CCS CONCEPTS

• **Software and its engineering** → **Software development methods**; *Development frameworks and environments*; *Software development techniques*; Software design engineering;

KEYWORDS

Model-driven engineering, single underlying model, role-oriented programming.

Reference Format:
Christopher Werner and Uwe Aßmann. 2018. Model Synchronization with the Role-oriented Single Underlying Model. In *Proceedings of MODELS conference workshops (MRT'18).* 10 pages.

MRT'18, October 2018, Copenhagen, Denmark

1 INTRODUCTION

Since the 1980s, the software engineering community has realized that there is no silver bullet to solve the complexities of software processes and model-dependent tasks. Ensuring the consistency of information fragments, while the system is running, is an essential aspect in the area of Models@runtime (M@RT) and Model Driven Engineering (MDE). Consistency between two models is defined as follows: Two models are consistent if their combination does not generate any contradictions and is realizable with implementation effort. Consistency makes maintaining artifacts a grand challenge. Without much work on maintainability of software elements, the quality inevitably degrades over time and with it the quality of the final product.

In the area of M@RT, it is necessary to create new runtime models as architecture, performance, and failure models from a running system on the fly [27]. These models may, but do not have to, be predefined in the software and must be hold synchronized with all other runtime models and the running system. As a solution, we need a runtime adaptation and synchronization mechanism for new runtime models from the running system and for the running system itself. It is possible to create bidirectional synchronization relations between all related models creating a collaborative consistent model environment. This method increases the number of synchronization relations quadratically to the number of involved models, i.e., n(n-1)/2 where n is the number of models. These pairwise consistency relationships must be added and adapted at runtime. In addition, they must be maintained over time to overcome modification problems like: (1) data inconsistency, (2) evolution of models, and (3) integration of new models.

The Single Underlying Model (SUM) approach by Atkinson et al. [3] is the paradigm we will concentrate on, because it overcomes these modification problems. All known information of a software system is stored in the SUM, whereby views represent user- and concern-specific fragments of the information. This reduces the number of correspondences to the number of views. The direct implementation of the SUM approach is the Orthographic Software Modeling (OSM) [3] approach with minimal overlapping views, which reduces synchronization problems. However, the current implementations of this approach are only design time approaches with a number of predefined views and a completely constructed SUM from scratch. However, the minimization of correspondences to the number of views and the synchronization of information in the SUM make this approach a good foundation to extend it by runtime adaptation mechanisms.

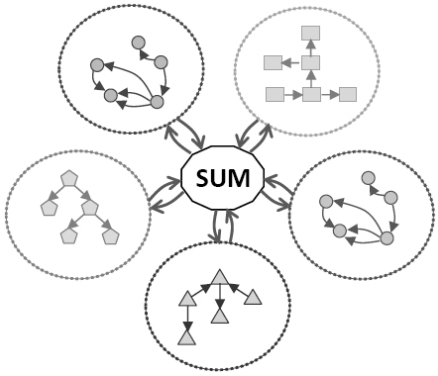

Figure 1: SUM / view centric environment, from [2].

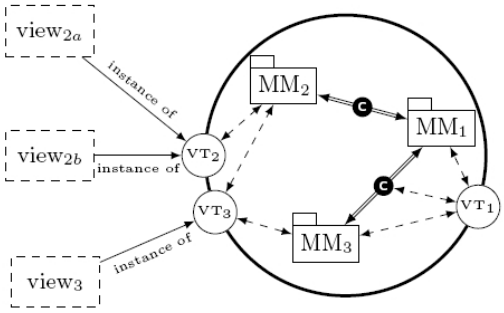

Figure 2: Vitruvius architecture [7].

This paper transforms the SUM approach from design time to runtime to make it usable in the area of M@RT for the synchronization of runtime models as views of the underlying source. We use the Compartment Role Object Model (CROM) by Kühn et al. [23] as SUM modeling concept. The role concept allows the modeling of views as compartments/ contexts, and contains a natural adaptation mechanism with roles that can be bound to objects at runtime and adapt their behavior. In addition, arbitrarily many roles can be bound to each object, which is a natural unlimited extension and adaptation mechanism provided by roles. In this paper, we address the following research questions:

- **RQ1:** How can the role concept simplify the creation of runtime views from a SUM and adaptation of a SUM?
- **RQ2:** How can the role concept be used to implement an incremental view update mechanism at runtime without redundant information?

The contribution of this paper is the Role-oriented Single Underlying Model (RSUM) approach using the role concept to create a SUM which is an integrated synchronization mechanism for views. The concept improves creation of views, introduces runtime adaptation, runtime synchronization between SUM and views, and reduces redundant information. Furthermore, the RSUM is prototypically implemented in the SCala ROLes Language (SCROLL) [25] with an illustrative example showing the benefits.[1]

The remainder of this paper is structured as follows. The next section summarizes background knowledge about closely related topics. Section 3 provides an in-depth discussion of the concepts with an illustrative example and its implementation details. We demarcate our approach from related work in Section 4. Finally, in Section 5, we conclude the paper and discuss lines of future work.

2 BACKGROUND

This section introduces the single underlying model approach, the view-based software development, and the role concept, which we combine in the next sections to our Role-oriented SUM approach.

2.1 Single Underlying Model

As mentioned before, SUM [3] is the central comprehensive model of a system and contains all information for stakeholders (e.g., functional, non-functional, and visual information). If the SUM is engineered, it is consistent from scratch. However, Atkinson et al. describe that the SUM gets complex for small models. This complexity does not allow a single user to understand the whole SUM. To distribute the complexity to different stakeholders, subsets of a SUM are presented in views that support the specific tasks of a user, as shown in Figure 1. If the views are consistent with the SUM, they are also consistent with each other. This is comparable to views in database management systems. Still, dependencies between the views cause problems during parallel modification on same elements in different views. In addition, the SUM and views are defined at design time and no runtime adaptation or extension is possible.

2.2 View-based Software Development

In the view-based software engineering area, views manage access to information from the models, as described in the section before. The views handle the complexity of the overall system and allow the separation of concerns in views. The IEEE 1471/ISO 42010 standard [17] gives the first definition of an architecture view and an architecture view type. In this work, we concentrate on the derived view and view type definition of Goldschmidt et al. [12] "A *View* is the actual set of objects and their relations displayed using a certain representation and layout." [12, p 63] and "A *View Type* defines rules according to which views of the respective type are created." [12, p 64]. Hence, views are instances of view types with elements of a SUM. In view-based modeling, two fundamental approaches are distinguished: the synthetic and projective approach. For the synthetic approach, a special architect creates and integrates all views in a system (e.g., SUM [3]). In projective approaches, the views are generated automatically with a special procedure and do not need any help from a software architect, e.g., Vitruvius [21] with the

[1] https://github.com/chrissi007/RoleSUM

query language ModelJoin [8] to create view types. Figure 2 presents Vitruvius, where users only see a SUM and not the connected metamodels which allows integration and evolution of new and existing metamodels without constructing the SUM from scratch, but creates redundant information in different connected models and needs complex synchronization mechanisms between all metamodels. The creation and synchronization of views from and with the software system is a complex task, which needs a special synchronization mechanism. Furthermore, the extension and adaptation at runtime are not considered in the view-based software engineering area.

2.3 Role Concept

The role concept is an extension of object-oriented design considering objects and classes as naturals (instances of natural types) that can play roles (instances of role types) in a specific compartment (instance of compartment type). Roles represent the behavior of naturals and can be acquired or abandoned multiple times. Moreover, roles interact with each other within compartments. Compartments represent containers or contexts and separate concerns. An advantage is that two naturals of the same type can have completely different behavior at runtime because of differently played roles. The role concept we consider is based on CROM [23] with the graphical notation in [22], which we use for our illustrations. The behavioral, relational, and contextual properties of roles are defined in 26 features which are the foundations of the role concept. Henceforth, we utilize CROM to specify the RSUM and the role-based programming language SCROLL [25] to implement the approach for our running example. SCROLL is an open source Scala library and implements most role features [25]. It is flexible, lightweight, and easily extensible. In our implementation, we use two main features of SCROLL: (1) the `play` operator binds a role to its player, and (2) the unary `+` operator before a method call performs a dynamic dispatch to a suitable role played by the object. The ability of roles to be attached and detached at runtime creates a natural mechanism of extensibility and adaptability, which is used for our requirements to adapt the described view-based approaches at runtime. In addition, Scala allows the extension of the source model, the synchronization, and the views by loading classes at runtime. These features enable the runtime model consistency, as deeply explained in more detail in the next section.

3 ROLE-ORIENTED SINGLE UNDERLYING MODEL (RSUM)

In this section, we describe the concept of an RSUM and the advantages that we see in such a view-oriented approach. For simplification, we only use the word view in the concept, which can be seen as a runtime model from an underlying source. Before that, we first present a running example not related to M@RT, but one that showcases our approach and the important features related to different small views.

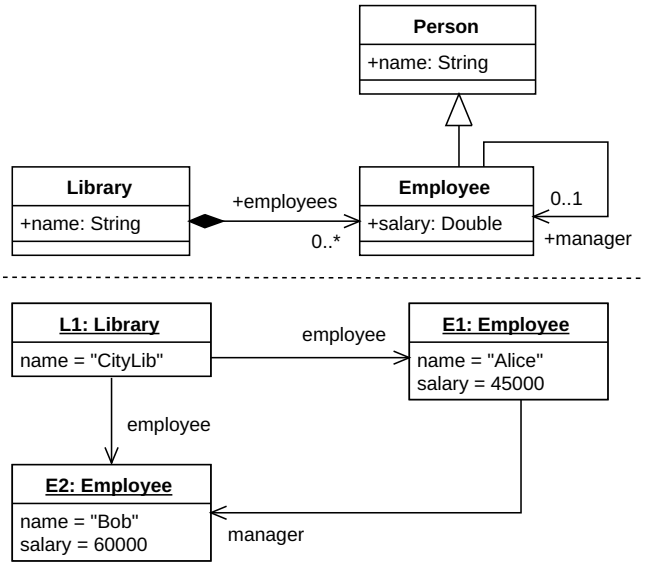

Figure 3: Running example with meta- and instance model.

3.1 Running Example

For the running example, we use a simplified library model, where a `Library` has different `Employees` that inherit from an abstract `Person` class. The running example is presented in Figure 3 with meta and instance model. In the instance model, a library `L1` with the name `CityLib` exists that has two employees called `Alice` and `Bob` where `Bob` is the manager of `Alice`. This example can be split into different views. One view can contain the complete information from the instance model while another view only includes all employees with their manager and a third view only provides the library with its employees without any manager information. We will use this example to showcase our approach.

3.2 Concept

The concept of RSUM is demonstrated for our Library-Example in Figure 4. It is divided into a core part that is marked in a blue box in the lower part and into different views that are visualized in the upper part of the figure. The core contains central information and data structures which are modeled as natural types and relational compartments that represent the relational behavior of the RSUM. In the RSUM approach all relations are transformed to relational compartments, i.e., the `manager` and `employees` associations are transformed to the `ManagerOfEmployee` and `LibraryHasEmployee` compartments, because all relations between naturals except inheritance are not allowed in the role concept. These relations must be moved into relation compartments. The concept of relation compartments allows:

(1) **Loading relations at runtime** with class loader functionality in programming languages like Scala and Java.
(2) **Extending naturals with new relations at runtime** with the playing of roles in relational compartments.

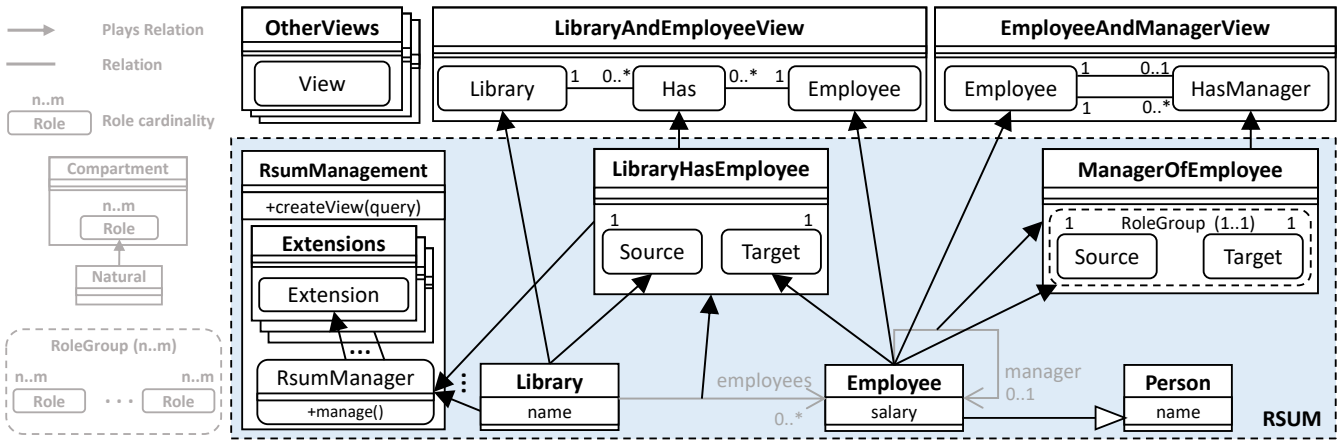

Figure 4: Concept of the role-oriented single underlying model (RSUM).

(3) **Add behavior and states to relations** as methods and attributes in relational compartments.
(4) **n-ary relations** with more role types in the relational compartment.

With these features, the relation compartments allow easy runtime adaptation of the RSUM as underlying source. This modification allows the construction of an RSUM from an object-oriented model with converting relations to relation compartments and the creation of views with elements that can play roles in the views. Moreover, the RSUM consists of an `RsumManagement` compartment, wherein each natural and relational compartment plays an instance of the `RsumManager` role type. In addition, it manages the instantiation of new views because it holds references on all instances from the RSUM and it adapts views at runtime. For clarity, only the plays relations from `Library` and `LibraryHasEmployee` to the `RsumManager` are drawn. These roles manage changes between views by reacting on insert, delete, and modification operations of naturals and relational compartments in the core. Furthermore, they propagate changes to all active views. The management compartment may also contain several `Extensions` to be added depending on the natural types and relation compartment types. These `Extensions` can implement histories, versioning, logging, and many more to observe user behavior and adaptation changes.

Two example views are represented in the upper part of Figure 4. The views are modeled as compartments with roles as the representations of the internal structure (e.g., the `EmployeeAndManagerView` presents only the `manager` relation). In `EmployeeAndManagerView`, each `Employee` role is played by a natural `Employee` and can have one `HasManager` role outlining his or her manager and many `HasManager` roles representing the subordinated employees. The `HasManager` role is played by the `ManagerOfEmployee` compartment. All relational compartments only contain a tuple of `Source` and `Target` role visualized with cardinality 1, i.e., the number of relational compartments is equivalent to the number of instances that form a relationship and the roles can only be

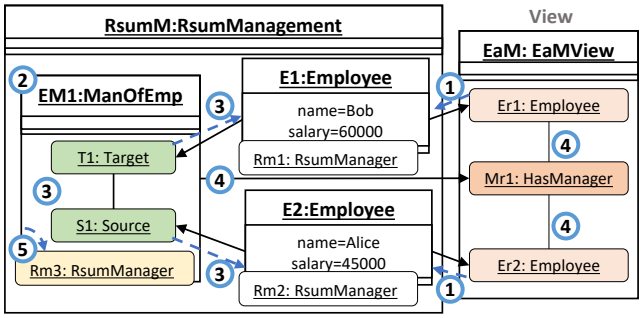

Figure 5: Instance view of RSUM from running example.

played by naturals of the RSUM. All roles that do not have a specific cardinality can be unlimited. The role group constraint with the cardinality (1..1) in the `ManagerOfEmployee` compartment describes that an employee cannot be their own manager, because he or she can not play both roles at the same time. In addition, each role in a view also requires an associated object in the RSUM core. This can be a natural or a relational compartment. The roles in the views also do not contain any information, because data can be retrieved and changed directly from the core using the internal mapping of the plays relation. However, the concept forces every view with write access to know which naturals or relation compartments have to be created when creating roles in it.

In Figure 5, the instance model of the RSUM approach and the creation process of the `HasManager` role (`Mr1`) are visualized, whereby names from the concept are abbreviated for space reasons. The representation shows the core on the left and the `EmployeeAndManagerView` view on the right side.

Now, we show how to create an RSUM in six steps. For creating `Mr1`, the `Employee` roles `Er1` and `Er2` are required: (1) the players of these roles `E1` and `E2` are determined. The view checks whether these players are really instances in the RSUM. If this is the case, (2) a new `ManagerOfEmployee`

compartment is created, otherwise the process is stopped. Within this, (3) new `Source` and `Target` roles are instantiated and bound to the two naturals from (1). If it is determined that the role group property is not fulfilled, the process is terminated. Otherwise, (4) establishes the play relation to the role in the view and sets the relationships to the `Employee` roles. In (5), a new instance of the `RsumManager` role is bound, and if `Extension` roles exist for this type, these are bound directly to the `RsumManager` role, but this process is not shown in Figure 5. The last step (6) iterates over all active views and creates new roles in other views when necessary. However, there is only one view that presents instances of this relation compartment, i.e., this step does not create more roles and bindings in this case and is not shown in Figure 5. Yet, this process shows the incremental runtime updates process underlining RQ2, because it only creates elements that are connected to the specific action and do not change other elements.

As presented in the RSUM concept and explanatory example, the role approach offers a lot of features and benefits. A natural has the ability to play arbitrarily many roles. This ensures that naturals can play roles in all existing views, which do not limit the number of views. The role concept also allows roles and compartments to play roles which is used for the `Extensions`. The creation of deep views in view-based approaches has not yet been considered in this paper, as it is not relevant for model synchronization and is not shown in Figure 4. However, the property that roles themselves can play roles, makes it easy to model and implement deep views. The resulting layered architecture with deep roles makes no sense on the small running example.

For implementing the concept, the runtime class loading options of programming languages are used to extend and adapt the approach in multiple directions, e.g., new naturals, relation compartments, and views can be added to the RSUM at runtime. These elements only need the mechanism to acquire and abandon roles to adapt the RSUM and to load and create new views at runtime (RQ1). In addition, it is possible to create read-only views, which only allow access to information with get functionality without the opportunity to change the internal state with set functionality. Removing views also just means removing roles and does not affect the RSUM core. In contrast, the deletion of elements in the views is propagated into the core and removes their core natural and all bound roles. The insertion process does the opposite: It creates and binds new management, extension, and view roles in the core and the active views to the new elements on the fly. This mechanism is another necessary part for incremental updates of the RSUM core at runtime because it connects the views to the RSUM and thus also the views among each other (basis for RQ2).

The roles within the views have no copied data from the RSUM and access the information of the naturals via the plays relation. This ensures that no redundant information is kept at runtime and that the status of the RSUM is consistent (underpins RQ2). If an attribute is changed simultaneously in two views, the last propagated change remains. A history

Listing 1: Interface of relation compartment.

```
1 trait IRelationCompartment extends Compartment {
2   var source: ISource = null
3   var target: ITarget = null
4
5   def delete() : Unit = {
6     if (source != null) {
7       plays.removePlayer(source)
8       source = null
9     }
10    if (target != null) {
11      plays.removePlayer(target)
12      target = null
13    }
14    +this deleteEverything()
15  }
16
17  trait ISource extends IRelationRole {
18    def getTarget(): ITarget = IRelationCompartment.
          this.target
19    def deleteRelation(): Unit =
          IRelationCompartment.this.delete()
20  }
21
22  trait ITarget extends IRelationRole {
23    def getSource(): ISource = IRelationCompartment.
          this.source
24    def deleteRelation(): Unit =
          IRelationCompartment.this.delete()
25  }
26 }
```

mechanism as extension compartment allows to monitor such changes. Our concept is based on materialized view types as own models of compartments to instantiate views at runtime and provide views with a predefined structure. Furthermore, in the role concept, each natural with all bound roles forms a compound object, which ensures that all roles reflect the same object and no object schizophrenia occurs. This fundamental mechanism allows to directly work on parts of the underlying instance without the usage of a communication layer to propagate changes through multiple layers or objects. The whole concept presents the synchronization of views from a single underlying source. However, if we create a combined model from preexisting models, where redundant information is removed, we can depict preexisting models as views from the RSUM. This results in a synchronization approach, which is able to generate new views/models and integrate new or existing legacy models.

However, the concept presented in this section also has some shortcomings. For each pair of elements involved in a relation, a new relation compartment is required which leads to an immense amount of compartments. It must also be ensured that the players of the `Source` and `Target` roles in the relation compartments match the players of the bound role in a view, which must be guaranteed at runtime.

3.3 SCROLL Implementation

For our case study, we use SCROLL [25] to prototypically implement the RSUM approach of the running example introduced in Section 3.1.

In SCROLL, each compartment object contains a role graph called `plays` that handles the calling of the role's

Listing 2: Example of relation compartment.

```
1 class EmployeeHasManager(emp:Employee, man:Employee)
      extends IRelationCompartment {
2   this.source = new Source()
3   this.target = new Target()
4   emp play this.source
5   man play this.target
6   def behavior() : Unit = { ... }
7   class Source() extends ISource { ... }
8   class Target() extends ITarget { ... }
9 }
```

Listing 3: Class example of RSUM.

```
1 class Library (_name: String) {
2   private var name: String = _name
3   def getName(): String = name
4   def setName(n: String): Unit = {
5     name = n
6   }
7 }
```

Listing 4: Example role in view.

```
1 class LibraryRole(name: String) extends IViewRole {
2   /* Role specific functionality. */
3   def getNameView(): String = {
4     return +this getName()
5   }
6   def setNameView(name: String): Unit = {
7     +this setName(name)
8     +this changeTrigger()
9   }
10 }
```

methods and allows insertion and removal of nodes and edges. This option makes function calls compartment-specific, i.e., compartments do not allow other compartments to call internal role behavior. For our approach, we combine the role graphs to one that enables us to call all available role methods with the unary `+` operator in front of a method name. These `combine` commands are not presented in the listings but are imperative for the correct implementation of the RSUM. Moreover, SCROLL permits the execution of more than one role method per API call, i.e., when an instance plays more than one role of one type, a call of a role's method is successively executed for all bound roles. This function is crucial for instances of the `RsumManager`, because changes in the RSUM core must be propagated to all extensions and views.

To apply the RSUM approach to an object-oriented model, the object-oriented model must first be transferred to a core model as shown in Figure 4. Therefore, all relations must be expressed as relation compartments.

Listing 1 shows the interface of a relational compartment that has to be implemented. The interface defines the `Source` and `Target` roles and a function to delete the compartments. In the `delete` method (Lines 5 to 15), the roles are deleted from the role graph (Lines 7 and 11) and all bound roles of the compartment will be deleted in the `deleteEverything` method. This function uses the `+` operator to call a function defined in another connected role or in the player object. In this case, the `deleteEverything` method is implemented in the `RsumManager` role, which must be played by every instance in the core and is explained later in this section. These internal roles only manage access to related roles and forward delete functions to the compartment. This step is necessary because once a natural is deleted in the RSUM, all relationships of that element must also be deleted.

In Listing 2, the `EmployeeHasManager` relation compartment is shown that must inherit from the interface of Listing 1. It gets the associated naturals during creation and assigns them to new roles (Line 2 to 5). This ensures that the structures at instance level are always bound correctly. The `behavior` method should show that each relation compartment can have its own behavior and state to extend relations with new information.

To transfer the normal classes to the RSUM, hardly any adjustments have to be made except for deleting the relationship attributes. Listing 3 shows the `Library` with getters and setters to access the name. These methods are used by all played roles to change the state of the object.

After the RSUM has been created, views must be defined to reproduce the filtered content. Each view type is given a unique name to distinguish it from other types. Further, all interfaces, no matter whether they are views or extensions, have the `getRole` method that returns a corresponding role for a class. With this method each view knows what instances it represents and what roles these instances get in the view. If some elements are not presented in the view, this method does not return any role instance. In addition, this method must be unique, which means that it can not be possible for a class to receive several types of roles, or for one role type to be returned for several classes. This mechanism allows creating only one associated object in the core, when instantiating roles in a view. Furthermore, each view compartment contains a list of `roles` that includes all roles played in this compartment instance. This list is used to prevent duplicate instantiations.

Listing 4 shows an example role from a view that can be played by the natural `Library` in Listing 3. Each view role inherits from the `IViewRole` interface for correct view management. In addition, the listing shows how data of players is accessed (Line 4) and changed (Line 6 to 9). Both functions require the `+` operator to call a function defined in another connected role or in the player object (e.g., `setName` and `getName` are passed to the `Library`). The `changeTrigger` method is implemented in the `RsumManager` role and activates all `Extension` roles bound to this object once there. If the views should react to changes of instances, this mechanism can be used to update visualizations in views. Furthermore, view roles still have delete functions which are not specified in the listing. These functions trigger a cascade of changes in the core model and delete all connected roles and relational compartments if necessary. If a view is created as read-only view, all delete and insert operations will not be generated in the later view generation process. Without these functions, it is not possible to manipulate elements in the RSUM.

Listing 5: Excerpt from the management compartment.

```
1 object RsumManagement extends MultiCompartment {
2   protected var extensions = ListBuffer[..]()
3   protected var activeViews = ListBuffer[..]()
4   protected var allViews = ListBuffer[..]()
5   protected var allRelations = ListBuffer[..]()
6   protected var allNaturals = ListBuffer[..]()
7   /* Insertion, creation, and deletion of views */
8
9   class RsumManager() {
10    def manageRsum(input: Object): Unit = {
11      if (input.isInstanceOf[IRelationCompartment])
            {
12        allRelations = allRelations :+ input
13        /* Combine compartments here. */
14      } else {
15        allNaturals = allNaturals :+ input
16      }
17      extensions.foreach { e =>
18        var role: IExtensionRole = e.
              getExtensionRole(input)
19        if (role != null)
20          input play role
21      }
22      activeViews.foreach { v =>
23        var role: IViewRole = v.getViewRole(input)
24        if (role != null)
25          input play role
26      }
27    }
28
29    def deleteEverything(): Unit = {
30      var player = this.getPlayer()
31      var roles = plays.getRoles(player)
32      /* Remove player from list */
33      roles.foreach { r =>
34        plays.removePlayer(r)
35        if (r.isInstanceOf[IViewRole])
36          r.removeRole()
37        else if (r.isInstanceOf[IRelationRole])
38          r.deleteRelation()
39        else
40          r.deleteNatural()
41      }
42    }
43
44    def changeTrigger(): Unit = {
45      +this runExtension()
46    }
47  }
48 }
```

The code example in Listing 5 represents the management object. This inherits, in comparison to all other compartments, from the `MultiCompartment` which allows the consecutive execution of several role behaviors with identical declaration and not only the execution of the first discovered role method. This option is currently only needed by the `changeTrigger` method in the `RsumManager` role (Line 44 to 46). This role sends the information about changes to all extension roles of the object as explained before. The extension roles react to this information in various ways (e.g., saving the change or modify some other elements). Furthermore, the `RsumManagement` object manages active views, integrated views, extensions, naturals, and relations (Line 2 to 6). An integrated view can be activated at any time, which inserts a new instance in the list of active views. The active views describe instances of known view types in which someone is currently working. The `createView` algorithm is not shown in Listing 5 but works as follows: (1) iterating over all relation compartments and binding them to roles returned from the view, and (2) iterating over all naturals to create and binding the last roles to the view. In the views, the links of the players of the `Source` and `Target` roles are created simultaneously, if they do not yet exist. In this process, the main part of the calculations takes place in the views. Furthermore, the management compartment contains all `RsumManager` roles (Line 9), which coordinate the deletion and insertion. The `manageRsum` method in the `RsumManager` role is called as soon as a new core element is created, which is passed to the function as an input and processed as follows:

(1) The new element is inserted into the corresponding list and if it is a relation compartment, the role graphs will be combined (Line 11 to 15).
(2) Matching `Extension` roles from the respective `Extension` compartments are bound (Line 17 to 21).
(3) Suitable roles are created and bound in all active views (Line 22 to 26).

However, as soon as a delete operation is called, the `deleteEverything` method is triggered. First, the player of the role (Line 30) and all associated roles (Line 31) are determined. Then the player is removed from the appropriate list in the management compartment and the function iterates over all roles to delete them from the graph (Line 29 to 42). For different role types, individual delete operators are called, which are only defined within these roles to guarantee the error-free deletion from compartments.

For the example implementation, we also wrote an extension compartment, which saves every change of a natural and thus creates a history. As soon as a change is made, a copy of all values is stored in a list. This extension can be used as a basis for a versioning module to reload older versions of the RSUM. This scenario is of interest for a distributed development scenario, in which changes can be traced and older states can be used.

All these functionalities are necessary to ensure the incremental updating of views at runtime and to avoid redundant data, which underlines our research question (RQ2) and provides a foundation for further developments. In addition, the role concept that SCROLL offers is the foundation for an understandable and easily usable way to create views at design and runtime (RQ1). The limitations of this approach are discussed in the section before. Moreover, the current state is a hand written implementation to show the applicability of the concept, but in the future the code should be generated from a domain specific language for view specification.

4 RELATED WORK

The RSUM approach described in this paper connects the research areas: single underlying model, view-based software development, databases, and role-based modeling. Each topic has its own background and related literature. We only refer to the work not yet mentioned in Section 2.

In the M@RT area there are already some approaches that deal with the synchronization of runtime models with their running model. The MORSE [16] approach is a model-aware service environment consisting of a model repository and model-aware services interacting with the repository. The repository manages model projects and artifacts that are equipped with universal unique identifiers for differentiation. The approach addresses traceability and collaboration problems between models and takes over versioning of artifacts. Attached services interact with the model repository at runtime and retrieve elements. MORSE includes managing models and their versions at runtime with external access. The SM@RT tool [26] is a synchronization tool between the running model and a MOF-compliant. The developer defines how elements are managed and manipulated in the tool. For this step, the target system must provide a management API. Each synchronization must be triggered before and after reading and writing the model. Moreover, the tool consists of a common library and a code generation engine, which together build the synchronization and thus offer the automatic generation of synchronization engines. These map a running system to model-based views, which represent completely separate models. In the work of Vogel et al. [27], runtime models are generated from the running system by triple graph grammars (TTG). These runtime models reflect various properties of the running system and are updated incrementally at runtime. In this approach, the runtime models are completely separated to avoid manipulating results. A notification mechanism reports changes from the source model to the target model. In addition, the approach can be extended by writing new TTG rules. As an example, performance, architecture, and failure models are generated, but in comparison to our approach they separate runtime models from the running model.

Since the beginning of the view-based software development research, different model view approaches have been created and examined regarding their usefulness during the software development process. Bruneliere et al. [6] present a survey about model view approaches with a feature model and sum up the view terminology and thereby also use the terminology from Goldschmidt et al. [12]. They identify 16 approaches that meet their requirements, i.e., software is based on MOF, defines view types, and computes conforming views. In their conclusion, problems of model view approaches are named, such as inconsistent terminology, view update problems, incremental view management, concrete syntax generation, and security aspects. Our approach addresses two of these problems: the view update problem, and incremental view management (with the `RsumManagement` compartment). Furthermore, the role concept provides a natural way to encapsulate information. We consider the approaches with bidirectional, incremental, and immediate updates in Table 1: EMF Profiles [24], mVTGG [1], Epsilon Decoration [20], OSM [3], OpenFlexo [13], and VIATRA viewers [10]. As additional criteria, we analyze the utilization of virtual views, the possibility to create deep views and the occurrence of object schizophrenia. vVTGG [18] is the only approach without incremental updates in Table 1, but it is the only approach that does not have object schizophrenia. OSM [3] was already mentioned in Section 1 and provides all requirements except deep views, which are not mentioned in their publications and they have object schizophrenia. They use the delta-based lens approach by Diskin et al. [11] to create an incremental, intermediate, and bidirectional update mechanism for the SUM. All approaches except vVTGG [18] are object schizophrenic, because the most approaches work on new model instances and not directly on the base model (e.g., EMF Profiles, Epsilon Decoration, and mVTGG). Other approaches create virtual views with traceability information between view and base model but also do not directly work on the base model and must propagate their updates across the traceability links (e.g., OSM, OpenFlexo, and VIATRA viewers). VTGGs [18] [1] are a more dedicated form of Triple Graph Grammars (TGGs), in which it is not possible to invert the source and target model. VTGGs are presented in two publications. In [1], mVTGG are implemented with an object adapter pattern, whereas [18] presents vVTGGs using the class adapter pattern, the use of which permits to remove object schizophrenia. Moreover, TGGs save the traceability information between two models in the correspondence graph and in doing so allow immediate and bidirectional updates between underlying source and view. In addition, the two approaches allow deep views while mVTGGs are based on a formal spezification and vVTGGs are not. Another approach is EMF Profiles [24] which transforms the profile mechanism from UML to EMF. This is done by extending EMF models with profiles to create a reusable mechanism in EMF and benefit from the UML profiles standard. The profiles are persisted as separate materialized models which can be called viewpoint over a single metamodel with the objective to extend it with annotations on existing elements. In addition, they are a dynamic model extension and do not pollute the base model. Their publication does not mention deep views, however we think it is possible to integrate them. The Epsilon Decoration [20] approach has model decoration support to create annotations of models to add new information. The decorations are represented in a generic way and the whole approach is implemented on top of Epsilon. They provide generic model handling functionalities like loading, storing, querying, and updating which are directly inspired by relational databases. The decorated models behave like views on the original model with new manually added information but they save their modification in separated decorator models and not directly in the original model. The authors do not mention deep views in their publication but it should be possible to create them with decorator chains. OpenFlexo [13] supports the federation of data from homogeneous technical spaces into virtual views, wherein no duplicated data is produced. The complete step is done with a DSL creating the views while an underlying model federation framework allows homogeneous handling of data as models. Each view is connected to its base models, but it is also possible to connect the different base models to the view resulting in a synchronization mechanism between the base models across the view. These bindings allow incremental, immediate, and

Table 1: Comparison with State-of-the-Art.

	vVTGG 2006 [18]	**EMF Profiles** 2012 [24]	**mVTGG** 2014 [1]	**Epsilon Decoration** 2010 [20]	**OSM** 2010 [3]	**DOREEN** 2017 [4]	**OpenFlexo** 2016 [13]	**VIATRA viewers** 2014 [10]	**RSUM**
Bidirectional updates	■	■	■	■	■	■	■	■	■
Immediate updates	■	■	■	■	■	■	■	■	■
Incremental updates	□	■	■	■	■	■	■	■	■
Virtual views	■	□	□	□	■	■	■	■	■
Deep views	■	∅	■	∅	□	□	□	■	■
No object schizophrenia	■	□	□	□	□	□	□	□	■

■: yes, □: no, ∅: not applicable

bidirectional updates, but is not suitable for deep views. VIATRA Viewer [10] has emerged from EMF IncQuery and provides an efficient incremental view management technique based on graph transformations. It uses derivation rules that are defined with annotation on query patterns form EMF IncQuery. Moreover, trace models are used between the base models and the views to reason about changes and provide synchronization support. Besides, it offers a way to serialize and reuse views and thereby, produces materialized view types with virtual and deep views as view chains.

The DOREEN approach [4] is a new deep view-point language similar to and with the same properties as OSM that uses the deep modeling technology for views and SUM. They use projective views of the SUM, which are completely defined by their content and its visualization. DOREEN is a component-centric view approach and uses the composite pattern to increase conceptual simplicity and flexibility and therefore has advantages in scalability, focusability, and customizability. For model synchronization tasks between SUM and views, round-trip engineering (RTE) strategies are needed. Hettel et al. [15] introduce definitions for RTE. For the transformation between two models, they distinguish between the relevant and non-relevant part of the two models. If there are changes in the relevant part of model A, then model B must be changed, too. Changes in the non-relevant part have no consequences in other model which is important for incremental updates. Furthermore, model transformations languages like ATL, QVT, and TGG are used to transform models in RTE systems.

In view-based software development, the views from a SUM need to be updated over time. This problem is closely related to the View Update Problem in relational databases. In the year 1978, Dayal [9] defined specific criteria that need to be satisfied by an updatable view to avoid any side effects. An updatable view behaves exactly like a standard table in a standard database. Therefore, Keller et al. [19] proposed an interactive view definition mechanism and introduced criteria to satisfy the update translations in the simplest form, such as no database side effects, minimal changes, and no unnecessary changes. But, after 40 years of research, an easy and powerful solution is still missing.

In Section 2, we briefly described role models, their properties, and SCROLL. For our realization, we use SCROLL as a role-based programming language but there are other options to implement the features of roles. The Role Object Pattern (ROP) [5] allows unanticipated changes at runtime. This flexible design pattern extends a core object to dynamically add and remove functionality in the form of new objects. The use of the ROP creates a lot of design overhead and does not allow the use of contexts. In contrast to ROP, SCROLL [25] and Object Teams/Java (OT) [14] are both role-based programming languages covering most of the 26 role features [23]. OT adds cross-cutting concerns to an existing application, whereby software composition and aspect-orientation play the central role. In OT, teams are compound objects similar to compartment types and contain all participating roles. The players are called base objects and play roles in the teams. Furthermore, roles add methods that can be called from the base class or forward information to the base class.

5 CONCLUSION AND FUTURE WORK

In this paper, we have described the advantages of using the role concept as the foundation to realize a SUM-based software engineering environment called the RSUM approach to synchronize models as views over an underlying source. The approach provides incremental runtime updates in views, and runtime integration of new views. In addition, it introduces a runtime adaptation mechanism in the view and the underlying source level without redundant data, whereby views work on compound objects removing object schizophrenia. Moreover, the role concept provides contextual, behavioral, and relational properties that are used for modeling relational compartments and views to separate concerns in views. The usability of SUM-based software engineering depends on both,

flexibility and stability. Flexibility means to add new types of views and models over time, whereas stability means to have fixed and stable metamodels, to support standards, and to have tool interoperability. The role concept and the related tools like SCROLL are research prototypes. This reduces the overall tool support and stability of the RSUM approach.

In future work, we need strategies to integrate legacy models into the RSUM. To achieve this, we work on a role-based model synchronization and transformation approach to use a uniform modeling and programming language for the whole process. In addition, we want to provide DSLs to generate views and view types without the currently hand-written implementation overhead. This DSLs should allow filter operations over the naturals and relational compartments to present only parts of all instances.

ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

REFERENCES

[1] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. 2014. Efficient Model Synchronization with View Triple Graph Grammars. In *Modelling Foundations and Applications*. Springer International Publishing, Cham, 1–17.

[2] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. 2013. A Multi-level Modeling Environment for SUM-based Software Engineering. In *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO '13)*. ACM, USA, 2:1–2:9. https://doi.org/10.1145/2489861.2489868

[3] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. Orthographic Software Modeling: A Practical Approach to View-Based Development. In *Communications in Computer and Information Science*. Springer Science & Business Media, 206–219.

[4] Colin Atkinson and Christian Tunjic. 2017. A Deep View-Point Language for Projective Modeling. In *21st International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 133–142. https://doi.org/10.1109/EDOC.2017.26

[5] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. 1998. The Role Object Pattern. In *Washington University Dept. of Computer Science*.

[6] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2017. A feature-based survey of model view approaches. *Software & Systems Modeling* (2017), 1–22. https://doi.org/10.1007/s10270-017-0622-9

[7] Erik Burger. 2013. Flexible Views for Rapid Model-driven Development. In *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO '13)*. ACM, USA, 1:1–1:5. https://doi.org/10.1145/2489861.2489863

[8] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based model-driven software development with ModelJoin. *Software & Systems Modeling* 15, 2 (2016), 473–496. https://doi.org/10.1007/s10270-014-0413-5

[9] Umeshwar Dayal and Philip A. Bernstein. 1978. On the updatability of relational views. In *4th international conference on Very Large Data Bases - Volume 4*. VLDB Endowment, 368–377.

[10] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. 2014. Query-driven Incremental Synchronization of View Models. In *2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, USA, 31:31–31:38. https://doi.org/10.1145/2631675.2631677

[11] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. 2011. *From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case*. Springer Berlin Heidelberg, 304–318. https://doi.org/10.1007/978-3-642-24485-8_22

[12] Thomas Goldschmidt, Steffen Becker, and Erik Burger. 2012. Towards a Tool-Oriented Taxonomy of View-Based Modelling. In *Modellierung 2012 (GI-Edition – Lecture Notes in Informatics)*, Vol. P-201. Gesellschaft für Informatik e.V. (GI), Bonn, 59–74.

[13] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. 2016. Addressing Modularity for Heterogeneous Multi-model Systems Using Model Federation. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM, 206–211.

[14] Stephan Herrmann. 2003. *Object Teams: Improving Modularity for Crosscutting Collaborations*. Springer Berlin Heidelberg, 248–264.

[15] Thomas Hettel, Michael Lawley, and Kerry Raymond. 2008. *Model Synchronisation: Definitions for Round-Trip Engineering*. Springer Berlin Heidelberg, 31–45. https://doi.org/10.1007/978-3-540-69927-9_3

[16] T. Holmes, U. Zdun, and S. Dustdar. 2009. MORSE: A Model-Aware service environment. In *2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*. 470–477. https://doi.org/10.1109/APSCC.2009.5394083

[17] May ISO. 2011. *Systems and software engineering–architecture description*. Technical Report. ISO/IEC/IEEE 42010.

[18] Johannes Jakob, Alexander Königs, and Andy Schürr. 2006. Non-materialized Model View Specification with Triple Graph Grammars. In *Graph Transformations*. Springer Berlin Heidelberg, 321–335.

[19] Arthur M. Keller. 1985. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In *4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '85)*. ACM, USA, 154–163. https://doi.org/10.1145/325405.325423

[20] Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos Matragkas, Richard F. Paige, Fiona A. C. Polack, and Kiran J. Fernandes. 2010. Constructing and Navigating Non-invasive Model Decorations. In *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 138–152.

[21] Max E. Kramer, Erik Burger, and Michael Langhammer. 2013. View-centric Engineering with Synchronized Heterogeneous Models. In *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO '13)*. ACM, USA, 5:1–5:6. https://doi.org/10.1145/2489861.2489864

[22] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. 2015. A Combined Formal Model for Relational Context-dependent Roles. In *2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*. ACM, USA, 113–124. https://doi.org/10.1145/2814251.2814255

[23] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. *A Metamodel Family for Role-Based Modeling and Programming Languages*. Springer International Publishing, Cham, 141–160. https://doi.org/10.1007/978-3-319-11245-9_8

[24] Philip Langer, Konrad Wieland, Manuel Wimmer, Jordi Cabot, et al. 2012. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology* 11, 1 (2012), 1–29.

[25] Max Leuthäuser and Uwe Aßmann. 2015. Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming. In *2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. ACM, USA, 25–33.

[26] Hui Song, Yingfei Xiong, Franck Chauvel, Gang Huang, Zhenjiang Hu, and Hong Mei. 2010. Generating Synchronization Engines between Running Systems and Their Model-Based Views. In *Models in Software Engineering*. Springer Berlin Heidelberg, 140–154.

[27] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. 2010. Incremental Model Synchronization for Efficient Run-Time Monitoring. In *Models in Software Engineering*. Springer Berlin Heidelberg, 124–139.