Christopher Werner, Hendrik Schön, Thomas Kühn, Sebastian Götz, Uwe Assmann

Role-based Runtime Model Synchronization

Erstveröffentlichung in / First published in:

Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 29. – 31.08.2018. IEEE Xplore, S. 306 – 313. ISBN 978-1-5386-7383-6.

DOI: https://doi.org/10.1109/SEAA.2018.00057

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

Role-based Runtime Model Synchronization

Christopher Werner*, Hendrik Schön†, Thomas Kühn*, Sebastian Götz*, and Uwe Aßmann*
*Software Technology Group, Technische Universität Dresden, Dresden, Germany
{christopher.werner, thomas.kuehn3, sebastian.goetz1, uwe.assmann}@tu-dresden.de
†Chair of Business Informatics, esp. IS in Trade and Industry, Technische Universität Dresden, Dresden, Germany
hendrik.schoen@tu-dresden.de

***Abstract*—Model-driven Software Development (MDSD) promotes the use of multiple related models to realize a software system systematically. These models usually contain redundant information but are independently edited. This easily leads to inconsistencies among them. To ensure consistency among multiple models, model synchronizations have to be employed, e.g., by means of model transformations, trace links, or triple graph grammars. Model synchronization poses three main problems for MDSD. First, classical model synchronization approaches have to be manually triggered to perform the synchronization. However, to support the consistent evolution of multiple models, it is necessary to immediately and continuously update all of them. Second, synchronization rules are specified at design time and, in classic approaches, cannot be extended at runtime, which is necessary if metamodels evolve at runtime. Finally, most classical synchronization approaches focus on bilateral model synchronization, i.e., the synchronization between two models. Consequently, for more than two models, they require the definition of pairwise model synchronizations leading to a combinatorial explosion of synchronization rules. To remedy these issues, we propose a role-based approach for runtime model synchronization. In particular, we propose role-based synchronization rules that enable the immediate and continuous propagation of changes to multiple interrelated models (and back again). Additionally, our approach permits adding new and customized synchronization rules at runtime. We illustrate the benefits of role-based runtime model synchronization using the Families to Persons case study from the Transformation Tool Contest 2017.**

***Index Terms*—Model-driven engineering, model synchronisation, role-oriented programming**

I. Introduction

Model-driven Software Development (MDSD) focuses on models as the primary development artifact to systematically realize a software system. Therefore, multiple models naturally coexist to express different concerns of interrelated concepts. These interrelated concepts can lead to an increasing number of related models that usually contain redundant information. Yet, independently editing one of these models easily leads to inconsistencies among them, making it imperative to ensure the consistency between them. Typically, model consistency is ensured by defining model synchronizations between those interrelated models, e.g., by means of model transformations, trace links, or Triple Graph Grammars (TGGs). As such, model synchronization is a crucial aspect of MDSD.

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

The literature distinguishes between unidirectional and bidirectional model synchronizations. Standard unidirectional synchronization approaches utilize transformation languages like SyncATL [1] and GRoundTram [2]. These provide well-behaved forward and backward transformations, i.e., they enforce consistency between multiple models and satisfy the roundtrip property [15]. Bidirectional model synchronizations are written as declarative consistency relations between two metamodels. The Query/View/Transformation (QVT) standard [3] of the Object Management Group (OMG), especially the QVT Relations (QVT-R) language, and TGGs [4] are famous instances of bidirectional transformation languages. While these already support model synchronization, they do not address the following three issues:

(1) Classical model synchronization approaches have to be triggered manually to perform the synchronization process between two models. In practice, systems and underlying models evolve at runtime. This evolution of multiple models requires immediate and continuous updates at runtime. Such behavior makes updates of the whole model impossible, which leads to the use of incremental (delta) updates for the synchronization of two models. The lens approach by Pierce *et al.* [5] is an often used concept for runtime model synchronization. Another issue stems from synchronization rules themselves, i.e., (2) the fact that they are specified at design time and, in classical approaches, cannot be extended at runtime. However, when metamodels or synchronization strategies change at runtime, these changes must be reflected by extending transformation rules. Moreover, the lifecycle of tools and models will be shorter in the future demanding a more flexible and extensible approach to model synchronization. Consequently, model synchronization tools will need to support adding and removing synchronization rules to fulfill these requirements. Finally, (3) most classical synchronization approaches focus on bilateral model synchronization, i.e., bidirectional synchronizing two models. Yet, if the number of related models increases, the number of model synchronizations grows quadratically to the number of involved models (i.e., n(n-1)/2 where n is the number of tools). To eliminate this combinatorial explosion of synchronization rules, a combination of synchronization rules over multiple models is essential.

All three issues have been addressed individually by recent approaches. Reactive ATL [6] and VIATRA3 [7] introduce reactive programming principles to model synchronization, i.e., synchronization rules do not need to be manually executed,

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

but are triggered by changes in the source models. Moreover, reactive ATL applies the same principle to synchronization rules themselves, which allows changing the rules at runtime. Finally, in [8], a TGG-based approach for multi-model synchronization rules is presented and in [9], the VIATRA3 approach is extended with the same feature.

This paper addresses all three issues with one approach leveraging on the expressiveness of role-oriented development, a successor of object-oriented development. We propose a role-based approach for runtime model synchronization to investigate the following research questions:

- **RQ1:** How can the role concept increase the quality of runtime model synchronization between multiple models?
- **RQ2:** How can the role concept create a flexible and extensible model synchronization?

To answer these questions, we utilize the role concept to establish runtime model synchronization between multiple interrelated models. In particular, we establish role-based model synchronization rules utilizing the Compartment Role Object Model (CROM) [10] and the SCala ROLes Language (SCROLL) [11] to model and respectively implement runtime model synchronization. As a result, our approach enables the immediate and continuous propagation of changes to arbitrarily many interrelated models (and back again). Additionally, role-based model synchronization permits adding new and customized synchronization rules at runtime by dynamically adding and removing synchronization rules and binding and unbinding synchronization roles to model elements. To illustrate the benefits of the presented role-based runtime model synchronization, we illustrate the approach employing the Families to Persons case study from the *Transformation Tool Contest* 2017 (TTC'17).[1] First, we describe role-based runtime synchronization for the families and persons models incorporating a third simplified EMF-based model. Finally, we outline the addition of custom synchronization rules to account for model evolution. In conclusion, this works applies the role concept to improve runtime model synchronization, to permit the evolution of synchronization rules at runtime, and to support multilateral model synchronization.[2]

The remainder of this paper is structured as follows. Sect. II briefly describes model synchronization and the notion of roles. Afterwards, Sect. III presents the running example that leads through the whole paper. Sect. IV introduces our role-based approach to model synchronization. Subsequently, Sect V elaborates on our prototypical implementation and illustrates its use. We demarcate our approach from related work in Sect. VI. Finally, Sect. VII concludes the paper and highlights possible lines of future work.

II. Background

A. Model Synchronization

Model synchronization is the process of keeping two or more related models consistent. For this, three main approaches can be distinguished.

First, transformation rules can describe how elements of a source model translate into elements of a target model (e.g., as in ATL [12]). Second, transformation rules can be composed of model queries to determine source elements, model change operations to describe how the target model shall be changed and execution schemata, which combine both to represent the transformation rule. An example of this approach is the VIATRA framework [13]. Finally, triple graph grammars can be used to describe transformation rules using three graphs per rule: the first describing a pattern in the source model, the second a pattern in the target model, and the third an intermediate model. We refer the interested reader to Czarnecki *et al.* [14], who introduce a feature-oriented classification of model transformation approaches. Although there are some bidirectional and few multilateral synchronization approaches, most approaches only support one-way batch-oriented model synchronization. However, for MDSD with round-trip engineering [15] bidirectional synchronizations or backward transformations are required, because software is developed in cycles of analysis, design, implementation, test, and deployment. When synchronizing two related models, Hettel *et al.* [16] distinguish between relevant and non-relevant parts for synchronization. The relevant part of a model describes the elements that trigger changes in the other model. In case of changes in the non-relevant part, however, the related model does not need to be changed. Changes in a model are produced at runtime. Thus, they must be propagated and integrated into the related models at runtime. In other words, synchronization rules need to be reactive instead of imperative.

B. Roles in a Nutshell

Roles are not a new concept. Yet, there is still no common understanding of roles in the literature [17], [18]. On the contrary, [17] and [18] identified 26 features attributed to roles that we group into the following three natures. The **behavioral nature** establishes that unrelated objects can play roles and roles adapt the behavior of playing objects [17], [18]. Additionally, objects can play roles of a different type multiple times. This nature is usually captured by the *fills*-relation between classes and role types denoting those classes whose objects can *play* roles of the given type. In contrast, the **relational nature** states that roles denote the binding ends of relationships. This nature is present in most modeling languages, e.g. ER and UML. Still, these languages do not foster the dynamism and flexibility of roles, as roles degenerate to named placeholders. Hence, researchers introduced roles tied to relationships as first-class citizens permitting them to be played by unrelated objects and having relationship specific properties. However, these relational languages assume that relationships are context-independent and cannot play roles themselves [10]. To resolve this, recent role-oriented languages incorporated the **context-dependent** nature of roles, that characterizes roles and relationships as context-dependent. Both are encapsulated in a *context* as a definitional boundary. Yet, different approaches use different terms for this conceptual

[1] www.transformation-tool-contest.eu/2017
[2] https://github.com/st-tu-dresden/RoleSync

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

entity. Consequently, *compartments* were introduced in [18] to generalize the different terms. Compartments can have properties and behavior, as well as play roles themselves. In contrast to context-dependent roles, only a few approaches also include context-dependent relationships [10].

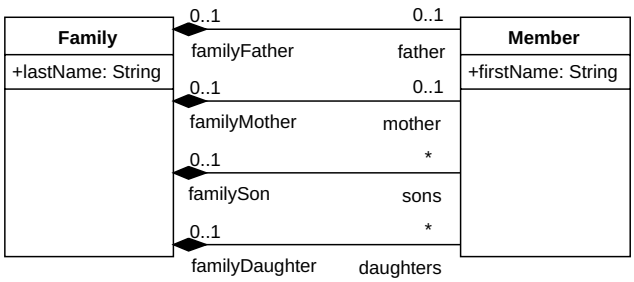
Including these natures into one role-oriented language already leads to an expressive framework. The Compartment Role Object Model (CROM) [10] is one such modeling language and the SCala ROLes Language (SCROLL) [11] a corresponding programming language. Henceforth, we utilize CROM to specify role-based model synchronizations and SCROLL to implement them in our case study. Our implementation utilizes two main features of SCROLL. The `play` operator binds a role to its player, i.e., another natural, compartment or role. The unary operator + before a method call performs a dynamic dispatch to a suitable role played by the receiver. These features enable runtime model synchronization.

III. Running Example

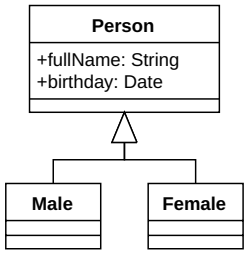
To illustrate the functionality of our role-based model synchronization approach, we use a preexisting scenario as running example, i.e., the Families to Persons use case for incremental model synchronization. Before describing our prototype, the full scenario is described including three models and corresponding synchronization issues. Fig. 1a and Fig. 1b depict the two models from the case study. As our approach supports multilateral model synchronization, Fig. 1c denotes an additional model for persons. For brevity, we denote these three models `A`, `B` and `C`. In detail, we synchronize instances of `Member` from `A`, `Male` and `Female` from `B` and `SimplePerson` from `C`.

The three models have the reference of persons and their respective relations as a common theme. In `A`, the `Family` exists as a core element and the `Members` are classified as father, mother, daughters, and sons. Moreover, the `Members` cannot exist on their own, only with an associated `Family`. In this way, any necessary information, e.g., family relationships, can be derived from `A`. `B` relies on the simple structure of male and female persons. Except for the full name and the birthday nothing is given as information. By inheritance, it is possible to identify the gender of a person at runtime. However, it is not possible to identify the relationship between the individual persons. In `C`, all persons are `SimplePersons`, regardless of their gender or family state. As well as in the other models, we have unique instances for each person. We have a single `SimplePerson` instance per person at runtime, carrying the actual values of name, gender, and an arbitrary (non-synchronization related) value named address. Synchronizing these three models entails certain problem cases. To illustrate them, we will shortly discuss individual bilateral transformations between the three models.

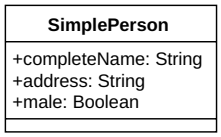
The first transformation direction from `A` to `B` is trivial. If we know the `Member` instance with its attributes (e.g., `familyDaughter`), we are able to decide which `Male` or `Female` instance is affected in `B`. Moreover, the associated `Family` instance gives access to the last name



(a) The Family-Member model.



(b) The Person-Male-Female model.



(c) The SimplePerson model.

Figure 1: The example scenario of family-person models.

Listing 1: Example of an ATL transformation rule.

```
1 rule Member2Female {
2   from s: Families!Member ( s.isFemale() )
3   to   t: Persons!Female  (
4     fullName <- s.firstName+'␣'+s.familyName() )
5 }
```

from the family member. For example, if we know that the `A` creates an instance `Eva:Member` that has set the attribute `familyDaughter` to the related family instance `Smith:Family`, then we can infer that in `B` the transformation rule would create the instance of `Eva-Smith:Female`. This case works similarly for the `father`, `mother`, and `son` association. Consider, for instance, the ATL model transformation from `A` to `B` shown in Listing 1. This rule transforms each female member of a family `Families!Member` into a single female person `Persons!Female`. The name is then composed by the single attributes. Notably, this implementation relies on two more complex helper functions, i.e., `isFemale()` and `familyName()`.

By contrast, the transformation from `B` to `A` is not trivial. Consider, an instance of `B` (e.g., `Eva-Smith:Female`) we want to transform into `A`. Then we simply lack the information, whether Eva is a mother or a daughter, and whether her family already exists or not. A solution would be to ask the user or always creating a new family and set females as `daughters`

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

and males as `sons`.

The transformation from `A` to `C` is straightforward. All information about a person is carried within the `SimplePerson`'s instance. Thus, if we create, modify, or delete a family member `Eva:Member`, then we simply create, modify, or delete the corresponding `Eva:SimplePerson` instance in `C`, as all needed attributes are already available in `A`. In contrast, the transformation from `C` to `A` is more complicated, as we lack necessary information for deciding whether `Eva:SimplePerson` is a mother or a daughter in `A`. In fact, this is the same issue encountered before, such that the same resolution strategy is applicable, i.e., assuming that male `SimplePersons` are always `sons` and females `daughters`. Besides that, a more advanced solution might be to check the family (via the person's complete name) for already existent family members first.

Finally, the transformation from `B` to `C` and back again is very simple, since all necessary information is present in both models. If an instance `Eva:Female` is created, deleted, or modified, then we simply have to add, delete, or modify the corresponding instance of `SimplePerson` and vice versa. Thus, both models can be fully synchronized.

Depending on the application, various model elements can be efficiently synchronized. Granted, it might be impossible to fully synchronize every detail of every new, deleted, or modified instance to all models in all directions. For instance, the `address` and `birthday` fields are unique attributes in `C` and `B`, respectively. Accordingly, these cannot be set when instantiating the corresponding classes. As these values are "optional values", this is no issue, because such fields can be set to `null` instead. In case of "mandatory" fields, however, we are unable to synchronize in this direction without someone providing the missing value to set. Henceforth, we employ this example and entailed synchronization issues to motivate and illustrate our approach.

IV. Role-based Model Synchronization

The advantage of role-oriented programming is its ability to dynamically adapt object's behavior by binding and dropping roles at runtime. For model synchronization, this entails that, when synchronizing preexisting models, roles permit introducing "synchronization management" for background, black-boxed, automated, non-invasive, and runtime model synchronization. Our approach realizes this by representing synchronization rules as compartments, which define synchronization roles played by the concrete model elements. Fig. 2 exemplifies our conceptual model as CROM model, whereas Fig. 3 and Fig. 4 showcase two typical runtime processes by means of CROM instance models illustrating the construction and synchronization of elements.

In general, we aim to synchronize multiple related models with a different structure. For instance, if two classes should be synchronized between two models, the synchronization needs to cover the creation, deletion, and modification of instances of such classes and has to propagate all changes to all related models. Our approach can cover those operations with a minimal effort and maximal expressiveness. However, the user of our approach can define synchronization of arbitrary fields and specify dedicated rules for synchronization compartments. In particular, each change of a model element's state is immediately propagated at runtime and only alters attributes of related objects.

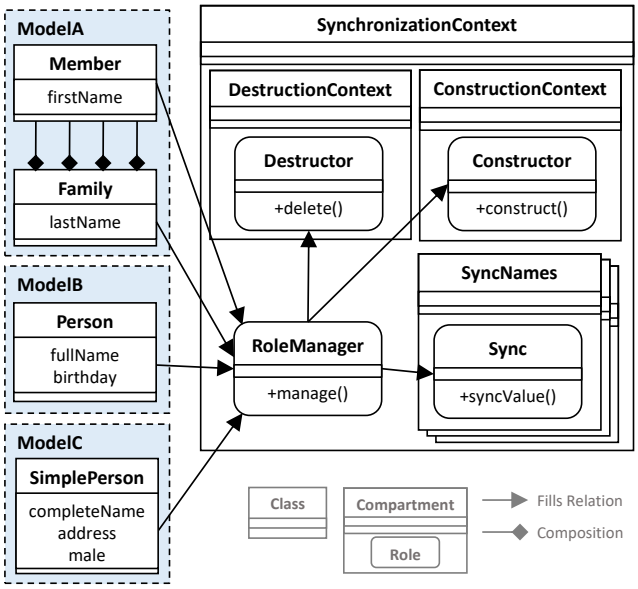


Figure 2: Role-based model synchronization.

Our approach supports two modes of model synchronization: *(a)* establishing the synchronization between models (initially or with additional objects at runtime), as well as *(b)* ensuring consistency between multiple models at runtime. To establish a model synchronization, the user defines all related classes within the different models with the related synchronization rules. To ensure consistency between multiple models, the actual synchronization is performed behind the scenes without the knowledge of the model user, acting as a black-box. In our approach, each model element is bound to a `RoleManager` within a `SynchronizationContext`, to introduce synchronization management of its player. Moreover, each `RoleManager` plays multiple synchronization roles in inner compartments that manage the construction, destruction, and content synchronization of model elements, as presented in Fig. 2. For each `SynchronizationContext`, there exists only one `ConstructionContext` and one `DestructionContext`. Yet, there are as many synchronization compartments, e.g., `SyncNames` compartments, as synchronized model elements. This design allows for dynamically adding or replacing synchronization rules without affecting the model elements, as new synchronization roles are only bound to the corresponding `RoleManagers`. Fig. 3 and Fig. 4 illustrate the synchronization of concrete instances of the three models including each object to the `SynchronizationContext`, which covers the creation, deletion, and modification of the corresponding objects.

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

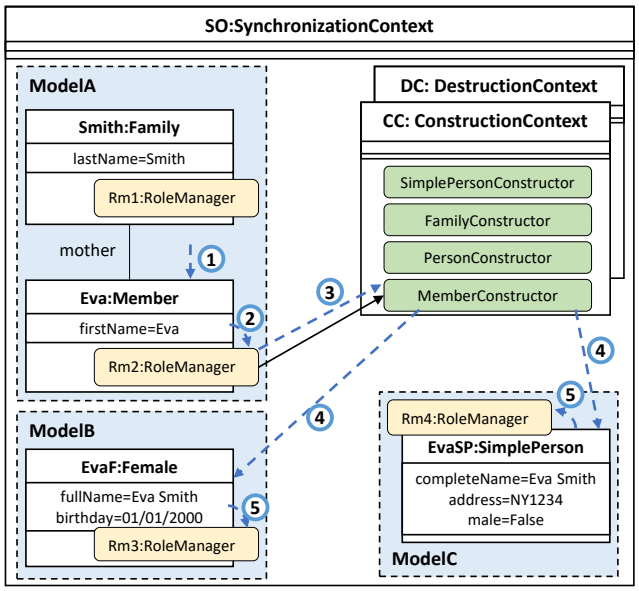


Figure 3: Example Model with construction rule process.

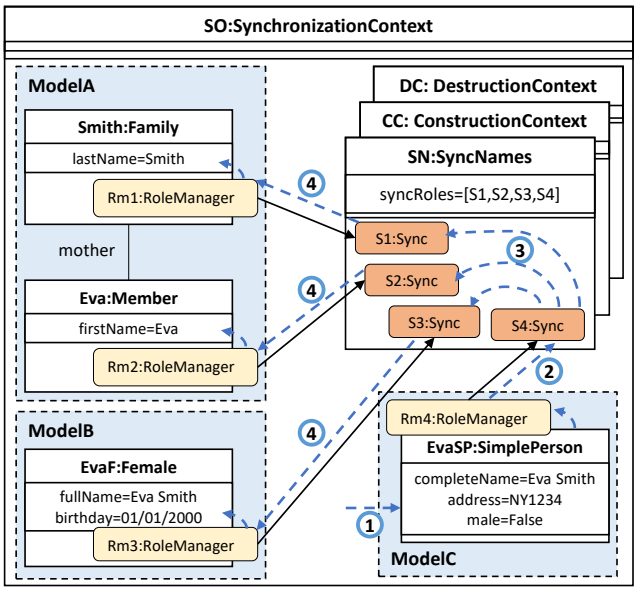


Figure 4: Example Model with synchronization rule process.

This is done at runtime, as we simply let instances of `Member`, `Family`, `Person`, and `SimplePerson` play multiple roles in instances of `SynchronizationContext` that realizes the synchronization.

Moreover, Fig. 3 depicts how model synchronization is established. First, a new `Member` named "Eva" is instantiated, who is the mother of family "Smith". Afterwards, a new instance of the `RoleManager` is bound to "Eva". In (3), the corresponding construction role from the `ConstructionContext` is chosen and bound to the `RoleManager` of "Eva". As specified in the `MemberConstruction` role, (4) creates new objects in both other models and binds new `RoleManagers` to these objects. Finally, each new `RoleManager` is bound to the corresponding roles of synchronization compartments, as depicted in Fig. 4. For clarity, this step is not visualized in Fig. 3. After establishing the synchronization the `RoleManager` dropped the construction role, to avoid redundant object creation. The deletion is handled similarly, whereby the `RoleManagers` delegates this process.

To ensure consistency upon changes to one of the three model instances, the `RoleManager` employs specialized compartments for each synchronization rules, e.g., `SyncNames`. The synchronization compartment propagates changes to all participating roles updating the respective attributes of their players. Fig. 4 presents the synchronization of the names of persons. Assuming the `completeName` of the `SimplePerson` "EvaSP" is changed (1), the `Sync` role is notified (2) and propagates the changes to the other `Sync` roles in the compartment (3). Finally, they apply the corresponding changes to the players of the corresponding `RoleManagers` (4). This process automatically synchronizes all elements participating in the synchronization compartment.

Besides that, binding and dropping roles at runtime allows for dynamically updating and extending synchronization rules by adding or replacing synchronization compartments within a `SynchronizationContext` including the construction and destruction compartment. We argue that introducing roles to model synchronization has the following benefits:

1) Reducing the effort to establish synchronizations between models or objects, due to the adaptive nature of roles and direct definition of synchronization rules. Additionally, it is possible to dynamically decide, which objects to synchronize.
2) The content to synchronize is exactly defined and encapsulated within synchronization compartments. Finally, roles allow for specifying synchronization on the same level as traditional object-oriented mechanisms, e.g., employing the adapter pattern [19].
3) The applicability of model synchronization is improved because any preexisting object can participate in a synchronization compartment with minimal additional implementation effort.
4) The approach supports automation, as all necessary synchronization compartments and roles can be generated from transformation rules (e.g., ATL rules).

Although our prototypical implementation demonstrates the first three benefits, the last benefit is part of future work.

V. Implementation with SCROLL

For our case study, we use SCROLL to prototypically implement the role-based model synchronization approach of the three introduced models. SCROLL is an open source Scala library and implements most role features [11]. It is flexible, lightweight, and easily extensible.

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

Listing 2: Class extension example.

```
1 case class SimplePerson(completeName:String,
2  male:Boolean) extends Player {
3    bindSynchronization()
4    def getMale(): Boolean = male
5    def getCompleteName(): String = completeName
6    def setCompleteName(n: String): Unit = {
7      completeName = n
8      +this changeCompleteName()
9    }
10 }
```

Listing 3: Construction role example.

```
1 @Role class PersonConstructor() {
2   def construct(playerManager:RoleManager) {
3     var player: PlayerSync = this.player
4     var fullName: String = +this getFullName()
5     var simple: SimplePerson = new SimplePerson
6         (fullName,player.isInstanceOf[Male)
7     /* Create Family & Member */
8     var simpleManager = new RoleManager()
9     simple play simpleManager
10    simpleManager play new SimplePersonDelete()
11    playerManager play new PersonDelete()
12    /* Create roles for Family & Member */
13    playerManager.addRelated(simpleManager)
14    simpleManager.addRelated(playerManager)
15    /* Add RoleManagers from Family & Member */
16    new SyncNames() {
17      playerManager play new Sync
18      simpleManager play new Sync
19      memberManager play new Sync
20      familyManager play new Sync
21    }
22  }
23 }
```

In SCROLL, each compartment object contains a role graph and handles calling the role's methods. Moreover, SCROLL permits executing more than one role method, 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 roles. This function is crucial for instances of the `Family`, because changes to the last name must be propagated to all related `Persons` and `SimplePersons` that are members of this family. To find role methods across different compartments, role graphs of the two compartments are merged. These merge commands are not presented in the listings but are imperative for the correct implementation of the synchronization.

To apply the synchronization approach, currently, the model code has to be slightly refined as outlined in Listing 2. First, each model class must extend the compartment type `Player` to add a role graph and permit calling the role functionality. Moreover, after setting attributes in the constructor, each class has to call the *bindSynchronization* method (Line 3) defined in the `Player` to bind the `RoleManager` and other synchronization roles. To propagate changes of attributes to roles, each setter or state modification includes a call to a role method (Line 8). If a corresponding method is found in a played role, it is immediately forwarded to it. Granted these modifications limit our approach, they are easy to overcome with aspect weaving or modified code generation.

After instantiating a model class, a new `RoleManager` is bound and its *manage* method is called. In this function, the type of the `Player` chooses the correct functionality and binds the right construction role and calls the *construct* method, as depicted in Fig. 3. Therein, all specified synchronization roles are bound and all related, synchronized instances in other models are created. Listing 3 exemplifies the construction role for `Person` that creates a related `SimplePerson`. Moreover, each construction method implements five steps:

1) **Line 3-4:** Get the player and other values for the construction of instances from other models.
2) **Line 5-7:** Create related instances in the other models (e.g., create a `SimplePerson` instance with the *fullName* from the person and define whether it is male or female).
3) **Line 8-12:** Add deletion roles to all instances of this construction process. All roles are bound to the corresponding `RoleManagers` to generate a single point for changes of roles. The related instances from step two do not get any construction role, because their construction is handled in this method.
4) **Line 13-14:** To manage related instances of the other models, each `RoleManager` saves its related `RoleManagers`.
5) **Line 16-21:** Add specific compartments for the runtime synchronization of different attributes. In this case, a new `SyncNames` compartment is created and each `RoleManager` binds a new `Sync` role.

The destruction roles are implemented similar to the construction roles, but have fewer steps. In the delete method, the role iterates over all played roles and drops them. Moreover, it takes all related `RoleManagers`, as defined in the construction process, and calls their delete methods. In summary, this method recursively drops every synchronization role and removes the related instances from all models.

Finally, the runtime synchronization of names among the family-person models is handled in the synchronization compartment implemented in Listing 4. The compartment can contain different synchronization roles, e.g., the `Sync` role, that supports synchronization behavior for name attributes. Furthermore, the compartment permits each role to get access to all other roles in this compartment (Line 2). If the `completeName` is changed in a `SimplePerson` instance the function *changeCompleteName* of the `Sync` role is called. In this method (Line 5-18) the complete name of the player of this role is extracted, separated into the first and last name, and propagated to all roles in this compartment, as full, complete, first, or last name. The `isSyncing` variable ensures one-time execution avoiding infinite loops. In sum, these listings present the `PersonConstruction` and `SyncNames` compartment for the running example.
Additionally, the `SyncNames` compartment can be exchanged at runtime, for instance, to account for a model evo-

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

Listing 4: Synchronization of names.

```
1 class SyncNames() extends Compartment{
2   var syncRoles = List[Sync]()
3   var isSyncing = false
4   @Role class Sync() {
5     def changeCompleteName(): Unit = {
6       if (!isSyncing) {
7         isSyncing = true
8         var comN: String = +this getCompleteName()
9         var r: Array[String] = comN.split("␣")
10        syncRoles.foreach{a =>
11          +a setFullName(comN)
12          +a setCompleteName(comN)
13          +a setFirstName(r.head)
14          +a setLastName(r.last)
15        }
16        isSyncing = false
17      }
18    }
19    def changeFirstName():Unit = {...}
20    def changeLastName():Unit = {...}
21    def changeFullName():Unit = {...}
22  }
23 }
```

lution. In essence, to exchange the `SyncNames`, all roles in the `SynchronizationContext` are queried for `RoleManagers` playing roles in a `SyncNames` compartment. All these roles are removed from their `RoleManagers` and replaced with new roles of the extended `SyncNames` compartment. In sum, this process only drops and binds roles without requiring changes to model elements.

Although the case study shows that the current implementation has limitations, e.g., modifying the model code, these can be removed with aspect weaving or code generation. Moreover, SCROLL does not allow for dynamic loading of roles into compartments, which requires loading and exchanging complete compartments at runtime. Despite that, domain-specific languages, such as ATL (Listing 1), can be utilized to generate the synchronization compartments and roles. Finally, the complete synchronization process can be changed at runtime and new models can be integrated at runtime.

VI. Related Work

As a core discipline of model-driven engineering, the field of model transformations has enjoyed tremendous popularity. Most notably, QVT is a language for model transformation standardized by the OMG [20]. ATL builds on top of the Eclipse Modeling Framework [21] and provides a language as well as an execution environment [22]. A special kind of model transformation is model refactoring [23], which has been studied intensely and is integrated into programming tools like Eclipse for the definition of transformations that leaves the functionality unchanged. Roundtrip engineering [15] is the combination of forward and backward transformations and is one of the major challenges of model-driven engineering.
If manual changes in code cannot be propagated back to the model from which the code (skeleton) was originally generated, then models and code quickly get out of sync.

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

	Reactive ATL	VIATRA3	TGGs	RbMS
Change propagation	■	■	■	■
Runtime rule changes	■	□	□	■
Multilateral rules	□	■	■	■

Roundtrip engineering presents the beginning of the model synchronization process. Finally, graph transformation languages as, e.g., in FUJABA [24] and GReAT [25], denote important previous work w.r.t. current model synchronization approaches as described in the following.

In [6], *Reactive ATL* is introduced as the successor of *Incremental ATL* [26] and *Lazy ATL* [27]. The basic principle of reactive ATL is two-fold. First, the meta-modeling framework was extended to support change propagation for individual properties of the source model of model-2-model transformations (i.e., incremental ATL). Second, target model elements are only computed when they are explicitly requested (lazy ATL). By this, the one-shot approach to model transformations is changed to follow the reactive programming paradigm [28], i.e., the handling of changes and requests to models is fully decoupled. A comparable approach w.r.t. incremental ATL has also been proposed by Xiong *et al.* [1].

The VIATRA framework [7] provides an alternative model synchronization approach, which since its third edition [29] follows the reactive programming paradigm, too. In contrast to Reactive ATL, VIATRA aims at high scalability, i.e., is suitable for very large models and, since its first version, VIATRA is based on graph transformations [13]. In 2010, incremental evaluation of model queries was introduced (EMF-IncQuery) [30]. In [9], this approach has been broadened to support distributed model queries, too (IncQuery-D). In comparison to Reactive ATL, VIATRA3 does not use transformation specifications, which explicitly refer to source and target model elements, but use model queries as preconditions for transformations, model manipulation actions and execution schemata as composition programs over both. Another class of alternative approaches to model synchronization is based on the formalism of TGGs as, for instance, shown by Trollmann and Albayrak [8] and Giese *et al.* [31].

As shown in Table I, in comparison to the approaches described above, our role-based runtime model synchronization approach offers a solution to all three issues of model synchronization (reactive change propagation, runtime changes to synchronization rules and multilateral synchronization rules). Our approach allows to directly propagate changes from the source to the target model and back again. Moreover, it is possible to describe unidirectional, bidirectional, and multilateral transformations. Furthermore, the role concept enables the runtime integration of models for synchronization and the evolution of synchronization rules at runtime.

VII. Conclusion and Future Work

In this paper, we have described the potential advantages of using the notion of roles as the foundation to realize a

Final edited form was published in "Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague, 2018",
S. 306 - 313. ISBN: 978-1-5386-7383-6
https://doi.org/10.1109/SEAA.2018.00057

runtime model synchronization approach. In general, role-oriented languages extend the object-oriented paradigm featuring dynamic, context-dependent behavioral adaptation. These features are beneficial for model synchronization, as they allow for creating fine granular transformation rules, where each role is responsible for its own part of the synchronization process, e.g., creation and deletion of model elements or propagation of specific changes. Moreover, roles provide a natural way to hide information between players and roles and introduce a new encapsulation layer [32] for the synchronization process, wherein the players do not need any information about the synchronization. Our prototypical implementation showcased the feasibility of the role-based model synchronization approach by synchronizing three related models in the family-person context. We employed the role-based modeling language CROM to model and the role-oriented programming language SCROLL to implement our case study. Moreover, we described the extensibility of our approach and the ability to apply changes to synchronization rules at runtime.

In the future, we will extend the implementation from Sect. V to create a framework that connects uni- and bidirectional synchronization languages with the role-based synchronization approach. Therefore, we will create an abstract version of this approach that is adaptable with new custom synchronization rules for different kinds of models. Finally, the approach should be utilized to integrate and synchronize a multitude of both new and legacy models.

REFERENCES

[1] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 164–173.

[2] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "A compositional approach to bidirectional model transformation," in *Software Engineering - Companion Volume*, May 2009, pp. 235–238.

[3] O. Mof, "2.0 query/view/transformation specification," *Specification Version*, vol. 1, 2007.

[4] A. Schürr, "Specification of graph translators with triple graph grammars," in *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1994, pp. 151–163.

[5] M. Hofmann, B. Pierce, and D. Wagner, "Symmetric lenses," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 371–384.

[6] S. Martinez, M. Tisi, and R. Douence, "Reactive model transformation with atl," *Science of Computer Programming*, vol. 136, pp. 1–16, 2017.

[7] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the viatra framework," *Software & Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016.

[8] F. Trollmann and S. Albayrak, "Extending model synchronization results from triple graph grammars to multiple models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2016, pp. 91–106.

[9] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró, "Incquery-d: A distributed incremental model query framework in the cloud," in *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 653–669.

[10] T. Kühn, S. Böhme, S. Götz, and U. Aßmann, "A combined formal model for relational context-dependent roles," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015. New York, NY, USA: ACM, 2015, pp. 113–124.

[11] M. Leuthäuser and U. Aßmann, "Enabling view-based programming with scroll: Using roles and dynamic dispatch for establishing view-based programming," in *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, ser. MORSE/VAO '15. New York, NY, USA: ACM, 2015, pp. 25–33.

[12] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005*. Springer, 2006, pp. 128–138.

[13] D. Varró, G. Varró, and A. Pataricza, "Designing the automatic transformation of visual languages," *Science of Computer Programming*, vol. 44, no. 2, pp. 205 – 227, 2002, special Issue on Applications of Graph Transformations (GRATRA 2000).

[14] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.

[15] U. Aßmann, "Automatic roundtrip engineering," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 5, pp. 33–41, 2003.

[16] T. Hettel, M. Lawley, and K. Raymond, *Model Synchronisation: Definitions for Round-Trip Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 31–45.

[17] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," *Data & Knowledge Engineering*, vol. 35, no. 1, pp. 83–106, 2000.

[18] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann, *A Metamodel Family for Role-Based Modeling and Programming Languages*. Cham: Springer International Publishing, 2014, pp. 141–160.

[19] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[20] I. Kurtev, *State of the Art of QVT: A Model Transformation Language Standard*. Springer Berlin Heidelberg, 2008, pp. 377–393.

[21] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[22] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1, pp. 31 – 39, 2008, special Issue on Second issue of experimental software and toolkits (EST).

[23] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2 2004.

[24] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, "Story diagrams: A new graph rewrite language based on the unified modeling language and java," in *Theory and Application of Graph Transformations*. Springer, 2000, pp. 296–309.

[25] D. Balasubramanian, A. Narayana, C. van Buskirk, and G. Karsai, "The graph rewriting and transformation language: GReAT," *Electronic Communications of the EASST*, vol. 1, 2006.

[26] F. Jouault and M. Tisi, "Towards incremental execution of atl transformations," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2010, pp. 123–137.

[27] M. Tisi, S. Martínez, F. Jouault, and J. Cabot, "Lazy execution of model-to-model transformations," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 32–46.

[28] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys*, vol. 45, no. 4, pp. 52:1–52:34, Aug. 2013.

[29] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró, "Viatra 3: A reactive model transformation platform," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2015, pp. 101–110.

[30] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, "Incremental evaluation of model queries over emf models," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 76–90.

[31] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 543–557.

[32] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.