Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations

Manuel Wimmer, Nathalie Moreno, and Antonio Vallecillo

Universidad de Málaga, Spain
{mw,moreno,av}@lcc.uma.es

Abstract. Multi-viewpoint modeling is an effective technique to deal with the ever-growing complexity of large-scale systems. The evolution of multi-viewpoint system specifications is currently accomplished in terms of fine-grained atomic changes. Apart from being a very low-level and cumbersome strategy, it is also quite unnatural to system modelers, who think of model evolution in terms of coarse-grained high-level changes. In order to bridge this gap, we propose an approach to formally express and manipulate viewpoint changes in a high-level fashion, by structuring atomic changes into coarse-grained composite ones. These can also be used to formally define reconciling operations to adapt dependent views, using *coupled* transformations. We introduce a modeling language based on graph transformations and Maude for expressing both, the coarse-grained changes and the coupled transformations that propagate them to reestablish global consistency. We demonstrate the applicability of the approach by its application in the context of RM-ODP.

1 Introduction

Large-scale heterogeneous systems are inherently much more complex to design, develop, and maintain than classical, homogeneous, centralized systems. One way to cope with such complexity is by dividing the design activity according to several areas of concerns, or *viewpoints*, each one focusing on a specific aspect of the system and allowing different stakeholders to observe the system from different perspectives [18].

Although separately specified, developed, and maintained to simplify reasoning about the complete system specifications, viewpoints are not completely independent: elements in each viewpoint need to be related to elements in the other viewpoints to ensure consistency and completeness of the global specifications. Such relationships are normally specified by means of *correspondences*, which are statements that permit some items in each viewpoint to be identified as related to items in the other viewpoints. Prominent examples that advocate such architectural decomposition are the Reference Model of Open Distributed Processing (RM-ODP) [17], the Model-Driven Web Engineering (MDWE) initiative [26] or UML [27], which provide different diagrams to represent different aspects of a system.

In this paper we are concerned with the evolution of multi-viewpoint specifications. As any other software artefact, they evolve over time due to a variety of reasons: changes in the requirements, errors in the design, evolution in the underlying technology, modifications in the system configuration, hardware or network connections to improve performance, etc. In general, dealing with model evolution is not easy, and the situation

C.A. Furia and S. Nanz (Eds.): TOOLS Europe 2012, LNCS 7304, pp. 337–353, 2012.
© Springer-Verlag Berlin Heidelberg 2012

is even worse in case of multi-viewpoint system specifications: this implies not only consistent single-view evolution but also consistent multi-view evolution. A change in a view may imply changes in the rest of the views, or in the set of correspondences, which need to be synchronized to restore consistency.

A large number of approaches address the problem of multi-viewpoint integration and synchronization (e.g., cf. [7,10,32,37]). Due to the low-level of detail at which model changes are identified, represented, and handled by them, in terms of fine-grained atomic changes, most of these approaches become quite unnatural to system modelers, who think of model evolution in terms of coarse-grained high-level changes. Furthermore, tools supporting model evolution neither support detecting changes at this level of abstraction, nor do they permit propagating these kinds of changes through the correspondences. Therefore, everything needs to be done at the level of basic atomic changes, such as adding and removing elements or modifying their values. Thus, the semantic of the coarse-grained changes is lost which again may hamper the reconciliation of models, e.g., information is lost which should be preserved.

In order to bridge this gap, we propose an approach to formally express and manipulate viewpoint changes at a higher level of abstraction, by structuring fine-grained changes into coarse-grained ones that represent the conceptual units by which domain experts think and reason about the changes. They can also be used to formally define reconciling operations to adapt dependent views, using *coupled* transformations. For this purpose, we introduce a modeling language based on graph transformations and Maude for expressing both the coarse-grained changes and the coupled transformations that propagate them between viewpoints. Although our proposal has been designed to be generally applicable to any multi-viewpoint specification framework, in this paper we demonstrate the applicability of the approach by its application in the context of RM-ODP [17], the ISO/IEC and ITU-T standard architectural framework for multi-viewpoint specification of open distributed systems. RM-ODP provides five complementary viewpoints: *enterprise*, *information*, *computational*, *engineering*, and *technology* that allow to observe the environment from different perspectives.

2 Motivating Example

In order to illustrate our proposal we will use here a simple example of a multi-view specification in the context of the RM-ODP (Fig. 1-top). It models a banking application, which manages accounts owned by customers. Users can access banking services through Branches or ATMs. Some operations should be authorized by regional head offices, and several databases store the customers information, account, and the own bank organization. This can be seen as a three-layer architecture, where branches and ATMs provide the interfaces and basic banking operations to users, the headquarters provide the main business logic, and the databases store the system data.

The *Computational Viewpoint* (CV) focuses on the functionality of the system and its software architecture, which is described in terms of components (computational objects) and connectors (that can be either simple primitive bindings or more complex binding objects). The *Engineering Viewpoint* (NV) deals with how the functional components (basic engineering objects, or BEOs) are distributed in nodes (separated

computing places) and connected via channels. Fig. 1 shows these two views, using the UML Profiles defined by the UML4ODP standard notation [16]. The other three viewpoints described by ODP (Information, Enterprise, and Technology) have not been included here for simplicity.

The elements of these two models are related through correspondences, which are expressed here using UML dependencies. Correspondences are shown in Fig. 1 using thicker dashed lines. They relate computational objects and bindings with the corresponding engineering objects and channels in the NV.

Evolution Scenarios in ODP. Let us think of a revised version of the Bank IT system specification, shown in Fig. 1-bottom. It contains three main changes: (1) computational objects Branch and ATM have been merged; (2) the primitive binding between HeadOffice and DBManager computational objects has been substituted by a binding object with more functionality (to add more powerful security mechanisms), and (3) the replica manager Dup in the NV has been moved from node Site4 to Site5.

Describing the changes at this level of detail is the way in which we normally reason about any system specification. Existing model difference tools calculate a very large number of atomic changes that need to be applied to the individual model elements. Understanding and manipulating those changes to, e.g., reason about the evolution of the system specifications or to propagate the (atomic) changes from one view to the rest, becomes quite a complex and brittle task. Quoting the well-known saying: "you can't see the forest for the trees."

In order to address this problem, we need to have a mechanism for representing and manipulating changes in models at a higher level of abstraction. We do that by structuring atomic changes into coarse-grained changes, which are closer to the way in which domain experts think about viewpoint evolution. Of course, the higher the abstraction level, the more *domain-specific* they get. This is because of the semantics they convey. In general, high-level composite operations depend on the specific domain. Each one defines a set of operations which reflect the kinds of changes commonly used in such a domain. For example, when dealing with software architectures two usual changes are to split a component into several and to merge several components into one. They imply a set of many atomic changes due to all the arrangements that need to be done with their ports, their connections with other components, etc. But conceptually they are just two changes.

In the following sections, the evolution of the Bank IT system specification is used to describe how composite changes are (a) represented; (b) identified and constructed from the set of individual atomic changes that existing model difference tools detect; and (c) propagated from one viewpoint to others by using Maude.

3 Formalizing Viewpoints in Maude

Maude [6] is a high-level language, a high-performance interpreter and compiler that supports rewriting logic based specification and programming of systems. Because of its efficient rewriting engine and complete analysis toolkit, Maude turns out to be an

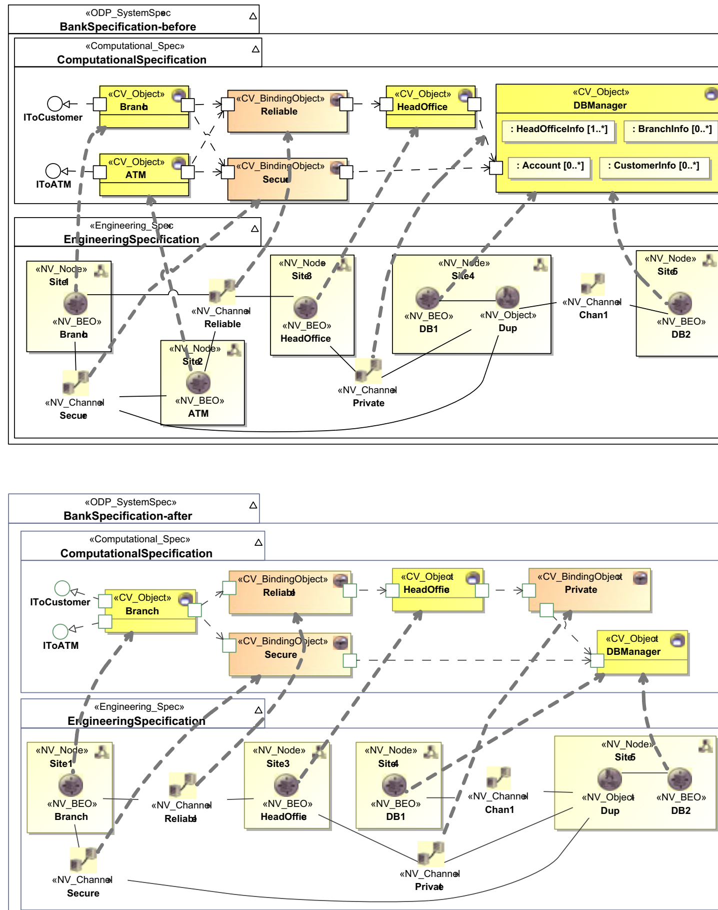


Fig. 1. Bank IT system specification expressed in ODP: Initial (top) and revised (bottom) models

excellent tool to specify and analyze many kinds of systems, at the appropriate level of abstraction. One of the benefits of using Maude is that its specifications are executable, since the rewrite rules that describe the behaviour of the system can be used to simulate it. The syntax for conditional rules is crl [l] : t => t' if $Cond$, with *l* the rule label, *t* the left-hand side (LHS) of the rule, *t'* the right-hand side (RHS), and *Cond* its condition.

Maude supports the specification of concurrent object-oriented systems in terms of object-oriented modules, which specify the system classes and their behaviour. Maude objects are structures of the form $< o : c \mid a_1 : v_1 , \ldots , a_n : v_n >$, where o is the object identifier (of Sort `Oid`), c is the class the object belongs to, a_i are attribute identifiers and v_i their corresponding current values. The current state of the object-oriented system, which is called a *configuration*, has the structure of a multiset made up of objects that evolves as dictated by the rewriting rules. Predefined sort `Configuration` represents configurations of Maude objects, with `none` as the empty configuration.

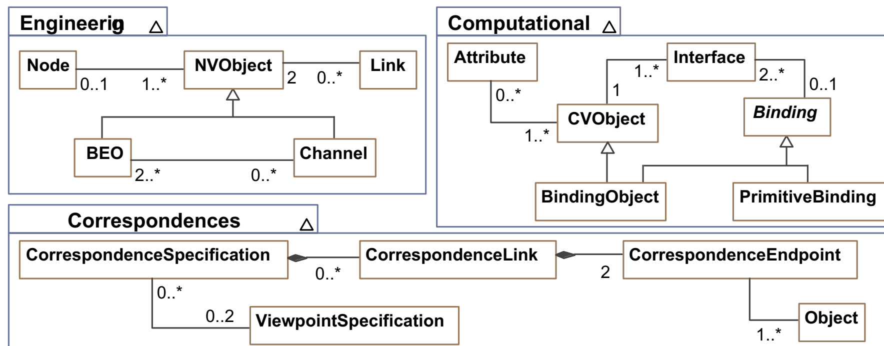


Fig. 2. Basic metamodels for CV, NV and Correspondences

In Maude, metamodels can be seen as object-oriented modules, which contain the specification of the metamodel classes. Thus, attributes can be represented as Maude attributes and references between metaclasses can be also represented as attributes, by means of sets of Maude object identifiers (Oid's). In this regard, depending on the multiplicity, we can use:

- a single identifier if the multiplicity is $[1]$
- a Maybe{Oid} which is either an identifier or a null value, for representing a $[0-1]$ multiplicity or
- a Set{Oid} for multiplicity $[*]$

The following listing describes the CV metamodel shown in Fig. 2 as a Maude object-oriented module:

```
(omod ODP is
protecting QID INT BOOL SET{Oid} CONVERSION .
--- Computational Viewpoint
class CVObject | interface : Set{Oid}, attribute : Set{Oid} .
class Attribute | cVObject : Set{Oid} .
class Interface | binding : Maybe{Oid}, cVObject : Oid .
class Binding | interface : Set{Oid} .
class PrimitiveBinding .
class BindingObject .
subclasses BindingObject PrimitiveBinding < Binding .
subclass BindingObject < CVObject .
endom) .
```

In the same way, a model that conforms to this metamodel can be represented in Maude by a configuration of Maude objects. Since objects may have attribute values and links, they are encoded as values of Maude objects' attributes. The configuration of Maude objects shown below represents an extract of the Bank specification model w.r.t. the CV specification illustrated in Fig. 1. It models two `CVObjects`, Branch and HeadOffice, linked by the `BindingObject` Reliable and two `Bindings`.

```
< 'Branch : CVObject | interface : ( 'IC1 , 'IC2 ) >
< 'HeadOffice : CVObject | interface : ( 'IC3 , 'IC4 ) >
< 'Reliable : BindingObject | interface : ( 'IC5 , 'IC6 ) >
< 'BC1 : Binding | interface : ( 'IC2 , 'IC5 ) >
< 'BC2 : Binding | interface : ( 'IC6 , 'IC3 ) >
```

4 Change Detection: From Fine- to Coarse-Grained Changes

Two kinds of approaches to change detection may be distinguished, namely, *model comparison* and *change tracking*. In a perfect world, we would assume to have a complete change log produced by the model manipulation tools automatically. However, current modeling editors are often not equipped with a change recorder. Furthermore, models can be edited with different tools and on different levels, e.g., within graphical or textual modeling editors, using UML and DSM tools, using the models' XML-based serializations, or by applying automatic model transformations. Model comparison is a generic approach to decouple the change log computation from the actual model manipulation.

In the context of this paper, we employ a two-phase *model comparison* approach. In the first phase, fine-grained changes are computed based on object identifier equivalences. For this phase, we build on our previous work presented in [30]. In the second phase, the fine-grained changes are analyzed to find coarse-grained changes between the two model versions. Furthermore, coarse-grained changes can also be composed into even coarser ones. In the following, we demonstrate both phases with the help of our running example.

4.1 Phase 1: Detecting Fine-Grained Changes

The first phase of the change detection consists of two sequential steps. The first step is to find the corresponding elements in the initial model and revised model based on matching rules. From the match result, differences are derived in the second step based on differencing rules.

Step 1: Matching. In the context of this paper, we use object identifiers to find the corresponding elements. A match is reported for each pair of objects having the same identifier assigned in the initial model and in the revised model. If such a pair is found, a match object is created which links the two objects. Of course, more sophisticated match rules based on name and structure similarities may be applied [30].

The following listing formalizes the previously explained match strategy. First, classes for representing MatchModels and Matches are introduced which are instantiated by the subsequent equation match. This equation is executed as long as objects with same identifier are found in the initial and the revised version of the model. Please note that both models are represented as *configurations* in Maude. Thus, the match operation is defined for two configurations (representing the initial and the revised model) and returns a match model which is again a configuration.

```
(omod Match is
 class MatchModel .
 class Match | initEl : Oid, revEl : Oid .
 subclass MatchModel < Configuration .
 vars INITIAL, REVISED, MATCH : Configuration .

 op match : Configuration Configuration -> MatchModel.
 eq match(< O : C1 | ATTS1 > INITIAL, < O : C2 | ATTS2 > REVISED)
 = < M : Match | initEl : O, revEl : O > match(INITIAL, REVISED) .
 eq match(INITIAL, REVISED) = none [owise] .
```

Example. When the match operation is executed for a subset of our running example considering only the elements involved in the EnrichBinding change, matches are

generated for the CVObjects, but the Binding in the initial model as well as the BindingObject and its Bindings in the revised model remain unmatched.

Step 2: Differencing. Based on the match model, the difference detection is performed. In the following, we introduce fine-grained change types and how instances of them may be detected. The following listing shows the supported fine-grained change types as Maude classes.

```
(omod fDiff is
 class DiffElement .
 class Addition | elem : Oid .
 class Deletion | elem : Oid .
 class Update | elm1 : Oid, elm2 : Oid, feature : String .
 subclass Addition, Deletion, Update < DiffElement .
endom)
```

Diff Calculation. Classes Addition, Deletion, and Update are instantiated by equations. These equations are built based on the following change detection rules explained in natural language: (a) If a model element of the initial model is not matched then it generates a deletion; (b) If a model element of the revised model is not matched then it generates an addition; (c) If a model element of the initial model is matched to an element of the revised model then they are compared for each feature the values of both model elements. Just when their values are different, an update is generated.

Diff Representation. For representing changes in a more convenient way, we rewrite the produced diff elements which are typed by generic change types (cf. classes Addition, Deletion, and Update) to metamodel-specific changes. Although, such differences are specific for a given metamodel, the difference metamodel is automatically derivable from the modeling language metamodel by using a dedicated transformation [4]. Instead of stating in the change model that an object has been added and more information about this change has to be queried by navigating to the objects in the revised and initial models, we aim for presenting more information about a change directly in the difference model (diff model) by having metamodel specific change types.

The design rationale for choosing this change representation is based on the assumption that metamodel-specific change types allow for a more concise formulation of programs analyzing the fine-grained changes—so to speak to provide an intuitive programming interface. Such programs are actually needed for finding coarse-grained changes in a set of fine-grained changes as well as for change propagation. Besides usability, also performance of dependent programs may be enhanced by this kind of representation.

Example. The differencing rules explained above allow to derive the following difference model for EnrichBinding change excerpt of the running example. By starting from the previously calculated matches, we end up with four fine-grained differences: DELBinding, ADDBindingObject, ADDBinding, ADDBinding.

```
Maude> rewrite < 'M1 : Match | iniEl : 'Branch, revEl : 'Branch >
               < 'M2 : Match | iniEl : 'DBManager, revEl : 'DBManager >
 result @Object: < 'D1 : DELBinding | element : 'B1 >
< 'A1 : ADDBindingObject | element : 'Reliable >
< 'A2 : ADDBinding | element : 'B2 >
< 'A3 : ADDBinding | element : 'B3 >
```

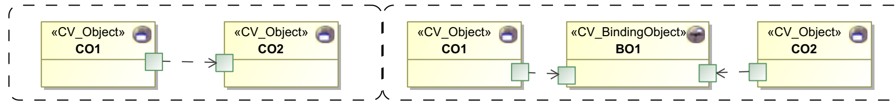


Fig. 3. Evolution pattern for the EnrichBinding change

4.2 Phase 2: Detecting Coarse-Grained Changes

Additional rules have to be formulated to structure fine-grained changes into coarse-grained changes. The development of such rules should be done in the language of the modelers. Because, in contrast to fine-grained changes which have simple and generic contracts, coarse-grained changes may comprise complex contracts. Thus, we sketch in the following subsection coarse-grained changes based on graph transformations patterns stating the situation before a coarse-grained change is applied, i.e., the pre-condition, as well as showing the effect of the coarse-grained change, i.e, the post-condition. These graph transformation patterns act as blueprints for the implementation of the detection rules for coarse-grained changes in Maude.

Sketching coarse-grained Changes. For sketching coarse-grained changes, we use *evolution patterns* which are based on graph transformation patterns using the concrete syntax of modeling languages. The pattern shown in Fig. 3 visualizes the EnrichBinding change in the concrete notation of ODP. The LHS of the pattern represents the situation before the change is executed and the RHS is showing the situation after the change has been applied. Thus, the semantic of the patterns is equivalent to standard graph transformation patterns. If an element resides on the LHS as well as on the RHS (i.e., the same variable name is used on the LHS and on the RHS), then it stays in the model. If an element only resides in the LHS and not in the RHS, it is deleted. Finally, if an element only resides in the RHS and not in the LHS, it is created. However, the operational semantics of such evolution patterns are different to standard graph transformation approaches. The evolution pattern is not executed by finding a match of the LHS in a model to rewrite it as given by the RHS to produce a new model version. Instead the evolution pattern is used to derive a program which detects the application of the described change. The detection is done by analyzing the initial and the revised model as well as the fine-grained changes between them.

Encoding detection rules in Maude. The detection rules for finding the evolution patterns are implemented in Maude based on the Maude operation called evolution, which has as input parameter a triple of models: (a) model before the change, (b) the model after the change, and (c) the difference model describing the fine-grained changes. The output is again a model which covers all coarse-grained changes happened between the initial and the revised model.

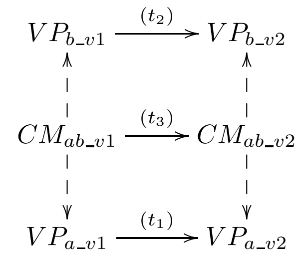
Based on the notion of the evolution operation and sketched evolution patterns, e.g., cf. Fig. 3, a Maude rule may be developed which searches for the application of the change. The main mechanism is to match for the set of fine-grained changes which make up the coarse-grained change. Each change type is represented by its own class which is instantiated by an accompanying rule.

```
(omod cDiff is
op model : Configuration -> Model[ctor] .
op evolution : Model Model Model -> Model .
class EnrichBinding | binding : Oid, bindingObject : Oid .
rl [EnrichBinding] :
   evolution (
    model( < C01 : CVObject | > < C02 : CVObject | >
     < B1 : Binding | source : C01, target : C02 > INITIAL ),
    model( < D1 : DELBinding | element : B1 >
     < A1 : ADDBindingObject | element : B01  >
     < A2 : ADDBinding | element : B2 >
     < A3 : ADDBinding | element : B3 > DIFF ),
    model( < C01 : CVObject | > < C02 : CVObject | >
     < B01 : CVBindingObject | >
     < B2 : CVBinding | source : C01, target : B01 >
     < B3 : CVBinding | source : C02, target : B01 >  REVISED ) )
=> evolution (
    model( < C01 : CVObject | > < C02 : CVObject | >
     < B1 : Binding | source : C01, target : C02 > INITIAL ),
    model( < EB1 : EnrichBinding | binding : B1, bindingObject : B01 > DIFF ),
    model( < C01 : CVObject | > < C02 : CVObject | >
     < B01 : BindingObject | >
     < B2 : Binding | source : C01, target : B01 >
     < B3 : Binding | source : C02, target : B01 >  REVISED ) ) .
endom)
```

Example. For detecting EnrichBinding changes, the Maude rule is shown in the above listing. The LHS of the rule is searching for matches of the evolution pattern of Fig. 3 by matching it on the initial, diff, and revised models. If a match is found, the atomic differences in the diff model are consumed and the coarse grained change EnrichBinding is instantiated instead, linking to the deleted binding in the initial model and to the introduced binding object in the revised model. By using this rule, the atomic differences computed by **Phase 1** can be reduced to just one EnrichBinding change. This result is further processed for change propagation in order to reflect the coarse-grained change of one view in depending views which is explained in the next section.

5 Change Propagation by Coupled Transformations

After coarse-grained changes in one viewpoint have been detected, they have to be propagated to dependent viewpoints. For this purpose, we follow the idea of *coupled transformations*—a term originally coined by Ralf Lämmel [22]. In particular, we aim for asymmetric reconciliation of viewpoints by exploiting explicit correspondence links between viewpoints.



The schema on the RHS illustrates the notion of coupled transformations interpreted in the context of viewpoint synchronization. An *initiator change*, by executing t_1 on the viewpoint VP_{a_v1}, produces a new version VP_{a_v2}. For retaining consistency between the dependent viewpoint VP_{b_v1} and VP_{a_v2}, the reconciling transformation t_2 has to be executed on VP_{b_v1}. Furthermore, to consider the modifications in the two viewpoints, another reconciling transformation t_3 has to be executed on the correspondence model CM_{ab_v1}.

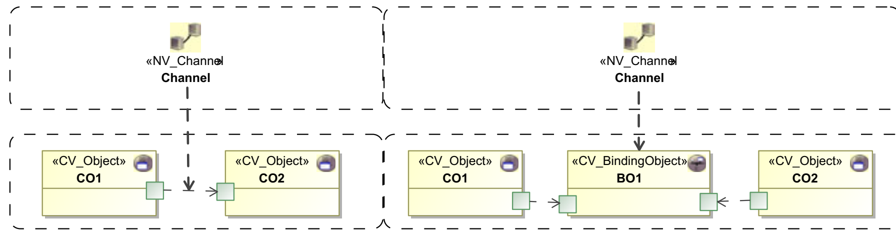


Fig. 4. Co-Evolution pattern for the EnrichBinding change

The execution of the initiator transformation is independent, meaning that it does not dependent on matches of other rules to compute its own matches. In contrast, the matches of reconciling transformations are based on the matches of the initiator transformations to consider the proper set of elements in dependent viewpoints which have to be adapted. To identify this proper set of elements, the correspondences between elements involved in the initiator change and elements of dependent viewpoints are the key information.

In the following, a high-level notation is introduced for coupling transformations which represents coarse-grained changes on different viewpoints. The notation extends the evolution patterns for coupling different evolution patterns to model *co-evolution patterns*. Subsequently, it is shown how co-evolution patterns are implemented in Maude.

5.1 Sketching Coupled Transformations

Co-evolution patterns, e.g., as shown in Fig. 4, comprise the following structure. First the initiator transformation (t_1) has to be specified. This is done by reusing an already evolution pattern which describes the change as discussed in the previous section. Having the initiator transformation as a basis, we may define new or reuse existing evolution patterns for describing reconciliator transformations (t_2 and t_3). For determining the exact matches of the reconciliator transformations, links between elements of the different patterns are used. In case of modeling languages offering explicit correspondence models, these links are expressed by additional correspondence models interlinking elements of two evolution patterns.

Example. For the EnrichBinding change, we may reuse the evolution pattern of Fig. 3 as the initiator transformation for defining the co-evolution pattern. `Bindings` in the CV are linked to `Channels` in the NV via correspondences. So when a `Binding` is deleted—this is actually the case when an EnrichBinding transformation is executed—there remain correspondences linking to missing elements in the CV. Thus, reconciliator changes are necessary to reestablish a link to proper CV elements. In case of EnrichBinding, the correspondences from `Channels` have to be relinked from missing `Bindings` to newly introduced `BindingObjects`. This reconciliation is specified in the co-evolution pattern of Fig. 4 by modeling another transformation for the correspondence model (middle layer). However, for finding the correspondences to adapt,

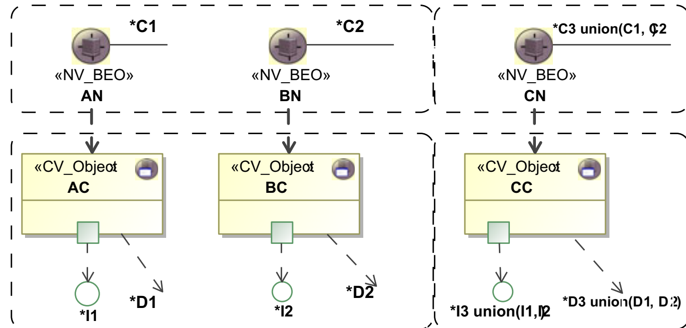


Fig. 5. Co-Evolution pattern for the MergeComponent change

the `Channels` of the NV are needed. Therefore another layer is introduced on top. No change in the NV is necessary, so one `Channel` is shown in the LHS and in the RHS to find the proper set of correspondences to relink.

A more complex example concerning the reconciliation of multi-viewpoints is the MergeComponent change. When it is detected in the CV, not only the correspondences, but also the NV has to be modified. This also involves to have sets of elements in the evolution patterns which are marked in our notation by the star operator. For instance, as sketched in Fig. 5, if two `CVObjects` named AC and BC are merged in a final `CVObject` CC, the union of their interfaces and bindings (cf. `union(I1,I2)` and `union(D1,D2)`) are required for CC. Similarly, when two `BEOs` are merged to reflect the change also in the NV, the union of their links to channels has to be build as well (cf. `union(C1,C2)` in Fig. 5).

5.2 Encoding Coupled Transformations in Maude

We describe the co-evolution of multi-viewpoints in Maude as an operation named multievolution that when applied to a particular configuration of viewpoints, produces a new configuration of them as a result.

```
(omod Reconciliator is ...
op multievolution : Configuration Configuration Configuration -> Configuration .
... endom)
```

Although more than one viewpoint may evolve at the same time, for the sake of simplicity, we assume here that there is only one base view that initializes the change. Thus, for each initiator change type, we use a Maude rule to trigger the evolution of other viewpoints related to the changed viewpoint. Of course, changes in one viewpoint will require the definition of several rules—one rule for each of the viewpoints that might be affected by the change—which describe how the system must continue to evolve in order to reach a reconciliation state between all viewpoints. The following listing sketches the general form of a propagation rule.

```
rl [nameRule] :
 multievolution( --- detection of the initiator change
  evolution ( --- V2: dependent viewpoint
    model( V2—BEFORE ), model( V2—REST ), model( V3—AFTER ) ),
  evolution ( --- correspondences
    model( CORR—BEFORE ), model( CORR—REST ), model( CORR—AFTER ) ),
  evolution ( --- V1: viewpoint originating the change
    model( V1—BEFORE ), model( V1—REST ), model( V1—AFTER ) )
 ) =>
 multievolution( --- execution of the reconciliation change
  evolution ( --- V2: dependent viewpoint evolves
    model( V2—BEFORE ), model( V2'—REST ), model( V3'—AFTER ) ),
  evolution ( --- correspondences evolve
    model( CORR—BEFORE ), model( CORR—REST' ), model( CORR—AFTER' ) ),
  evolution ( --- V1: viewpoint originating the change
    model( V1—BEFORE ), model( V1—REST ), model( V1—AFTER ) )
 ) .
```

The LHS of the rule contains the evolution models for the initiator and the related viewpoint and the correspondences relating them. For the viewpoint initiating the evolution, the composite operation representing the initiator change has to be identified. In this way, the RHS contains the effect of propagating the change to the other viewpoints and to the correspondences, again defined in terms of the high-level composite operations.

Example. Let us consider the Maude rule for propagating the EnrichBinding change from CV to NV. As mentioned before, the NV needs no adaptation, but the correspondences may have to be updated. Thus, the following rule matches for an occurrence of the EnrichBinding change that has not been propagated yet, using the third evolution pattern in the LHS. The first and the second evolution patterns are used to find the elements that are involved in the reconciliation. Thus, the LHS has to find the correspondence which link a channel with the enriched binding. In the RHS, the first and the third evolution patterns are equivalent to the LHS patterns, but the second evolution pattern is not. It takes care of relinking the correspondence to the created `BindingObject`.

```
rl [EnrichBinding2NV] :
multievolution(
 evolution ( --- engineering viewpoint
    model( < Chan : Channel | > ENG—BEFORE ),
    model( ENG—REST ),
    model( < Chan : Channel | > ENG—AFTER ) ),
 evolution ( --- correspondences
    model( < C1 : Correspondence | source : B1, target : Chan > CORR—BEFORE ),
    model( CORR—REST ),
    model( < C1 : Correspondence | source : B1, target : Chan > CORR—AFTER ) ),
 evolution ( --- computational viewpoint originates the change
    model( < C01 : CVObject | > < C02 : CVObject | >
     < B1 : Binding | source : C01, target : C02 > INITIAL—REST ),
    model( < EB1 : EnrichBinding | binding : B1, bindingObject : B01, propagated—
        ↪NV : false > DIFF—REST ),
    model( < C01 : CVObject | > < C02 : CVObject | >
     < B01 : BindingObject | > ... REVISED—REST ) )
) =>
multievolution(
 evolution ( --- engineering viewpoint same as in LHS ),
 evolution ( --- correspondences are updated
    model( < C1 : Correspondence | source : B1, target : Chan > CORR—BEFORE ),
    model( CORR—REST ),
    model( < C1 : Correspondence | source : B01, target : Chan > CORR—AFTER ) ),
 evolution ( --- computational viewpoint same as in LHS
             --- except < EB1 : EnrichBinding | propagated-NV : true >  )
) .
```

Let us now consider the Maude rule[1] for propagating the MergeComponent change. The third evolution pattern of the LHS matches for MergeComponent change in the CV that has not been propagated, yet. If a match is found, the NV and the correspondences start to evolve as the RHS of the rule dictates. Since each computational object that is not a binding object corresponds to a set of one or more basic engineering objects (and any channels which connect them), the coarse-grained MergeComponent operator causes that the Branch and ATM `BEOs` in the NV must also be merged. Finally, in order to preserve the system correspondences, the rule throws a final reconciliator evolution in the correspondences model to reestablish proper links between the new elements generated in the CV and the NV models. The reader should note that, at this point, a similar rule will also be required to define the effects of the MergeBEO composite operator in the entire system specification that we omit here for the sake of simplicity.

```
rl [MergeComponent2NV] :
multievolution(
 evolution ( --- engineering viewpoint
    model( < AN : BEO | > < BN : BEO | > ENG-BEFORE ),
    model( ENG-REST ),
    model( < AN : BEO | > < BN : BEO | > ENG-AFTER ) ),
 evolution ( --- correspondences
    model( < C1 : Correspondence | source : AC, target : AN >
           < C2 : Correspondence | source : BC, target : BN > CORR-BEFORE ),
    model( CORR-REST ),
    model( < C1 : Correspondence | source : AC, target : AN >
           < C2 : Correspondence | source : BC, target : BN > CORR-AFTER ) ),
 evolution ( --- computational viewpoint originating the change
    model( < AC : CVObject | >  < BC : CVObject | > COMP-BEFORE ),
    model( < MC : MergeComponent | target : CC, source1 : AC, source2 : BC,
        ↪propagated-NV : false > COMP-REST ),
    model( < CC : CVObject | > COMP-AFTER ) )
) =>
multievolution(
 evolution ( --- engineering viewpoint
    model( < AN : BEO | > < BN : BEO | > ENG-BEFORE ),
    model( < XN : MergeBEO | target : CN, source1 : AN, source2 : BN, propagated-
        ↪TV : false > ENG-REST ),
    model( < CN : BEO | > ENG-AFTER ) ),
 evolution ( --- correspondences
    model( < C1 : Correspondence | source : AC, target : AN >
           < C2 : Correspondence | source : BC, target : BN > CORR-BEFORE ),
    model( CORR-REST ) , --- replaces both correspondences by a new one
    model( < C3 : Correspondence | source : CC, target : CN > CORR-AFTER ) )
 evolution ( --- computational viewpoint same as in LHS
             --- except < MC : MergeComponent | propagated-NV : true >  )
) .
```

6 Related Work

Multi-Viewpoint Integration and Synchronization. A large number of approaches address the problem of multi-viewpoint integration and synchronization [7]. We have works on synchronizing artifacts in software engineering, mostly influenced by original works on multi-view consistency [11,13] using a generic representation of modifications and relying on users to write code to handle each type of modification in each type of view. This idea influenced later efforts on model synchronization

[1] Building the union of the links (cf. `union(C1,C2)` in Fig. 5) requires an additional rule for filtering reflexive links as well as duplicates which is not shown for sake of simplicity.

frameworks in general [19,20] and in particular bi-directional model transformations [33,37]. Other approaches use so-called correspondence rules for synchronizing models in the contexts of RM-ODP and MDWE [3,10,32]. More theoretical works propose to use different kind of lenses [8,9,12,15].

All these approaches have in common that they consider only atomic changes when reconciling models. Thus, the goal of the reconciliation is to change the models in a way that they satisfy again the given constraints. However, when structuring the changes to composite changes, more appropriate reconciled models may be found. The reason for this is that the semantics of the changes, modeling languages, and modeling domains are considered instead of reasoning with generic atomic changes for generic model elements. For example, when merging two elements into one may be represented by three atomic changes, namely deleting both elements and adding a new element which represents the two merged elements. When considering each atomic change in isolation, depending elements in other views may be deleted and a new element may be added if we have a one-to-one correspondence to fulfill between the views. However, the information of the deleted elements is lost. By using our approach, we are able to specify the rules for the reconciliation without information loss by merging also dependent elements in the other views instead of deleting them. The only work we are aware of allowing to propagate more complex changes is [29], however, in this approach it is required to record the initiator changes during model editing.

Metamodel/Model and Model/Instances Co-evolution. This involves synchronization between models of different abstraction levels [34]. In the general case, semantics-preserving transformations must be developed manually, based on the understanding of the semantic intent of the change. Several dedicated languages for metamodel/model co-evolution have been recently developed for specifying semantic-preserving transformations [5,14,25,31]. Most related to our approach is [36], where the composition of atomic differences to composite differences is discussed for Ecore-based metamodels. Having composite differences between metamodel versions is considered to be the prerequisite for finding the appropriate co-evolution for the model level. However, the propagation of the composite changes to the instances has not been presented. Our approach is generic in the sense that also metamodel/model co-evolution may be supported. In particular, the coupling between the metamodel changes and model changes is similar as the coupling of changes between different views.

Coarse-grained changes for models. Most existing approaches for defining coarse-grained changes focus solely on model refactorings. The work in [35] defined a set of UML refactorings on the conceptual level by expressing pre- and post-conditions in OCL, and [2] presented a refactoring browser for UML supporting the automatic execution of pre-defined UML refactorings. While these two approaches focus on pre-defined refactorings only, other approaches [21,28,38] allow the introduction of user-defined refactorings by using dedicated textual languages. A similar idea is followed in [1,24] but instead of textual languages, graph transformations are used to describe refactorings. However, the proposed approaches cover mostly single-view evolution and focus on the implementation of semi-automatically executable refactorings. Only some

first ideas for tackling consistency between different views in the context of coarse-grained changes have been presented. For instance, [23] proposed to refactor UML class diagrams, also adapting attached OCL constraints.

7 Conclusions and Future Work

We have presented an approach for expressing, executing, and synchronizing viewpoint changes at a high-level of abstraction. We structure atomic changes into coarse-grained changes that represent the conceptual units that domain experts are used to, and are coupled for propagating the semantics of one change in one viewpoint into related viewpoints. A major strength of our approach comes from the use of Maude and its expressive power. Although coarse-grained changes and coupled transformations have been used in previous works, the composition of fine-grained changes into coarse-grained changes for viewpoint synchronization using coupled transformation is novel and represents an alternative to constraint-based model synchronization.

As future work, we want to investigate a hybrid synchronization approach by using in the first phase the presented approach for propagating coarse-grained changes and in the second phase a constraint-based approach for propagating atomic changes which could not be composed into coarse-grained changes. In addition, applying the approach to other modeling domains will provide us extensive feedback. These experiences will be used to establish a model synchronization benchmark based on real-life scenarios coming from different application domains.

Acknowledgements. This work has been partially funded by the Austrian Science Fund (FWF) under grant J 3159-N23, and by Spanish Research Project TIN2011-23795.

References

1. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 425–439. Springer, Heidelberg (2006)
2. Boger, M., Sturm, T., Fragemann, P.: Refactoring Browser for UML. In: Aksit, M., Awasthi, P., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 366–377. Springer, Heidelberg (2003)
3. Cicchetti, A., Ruscio, D.D.: Decoupling Web Application Concerns through Weaving Operations. Science of Computer Programming 70(1), 62–86 (2008)
4. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. Journal of Object Technology 6(9), 165–185 (2007)
5. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Dependent Changes in Coupled Evolution. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)

7. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MoDELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011)
8. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. JOT 10(6), 1–25 (2011)
9. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS 2011. LNCS, vol. 6981, pp. 304–318. Springer, Heidelberg (2011)
10. Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A.: Change management in multi-viewpoint systems using ASP. In: WODPEC 2008. IEEE (2008)
11. Finkelstein, A., Gabbay, D.M., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling in Multi-perspective Specifications. In: Sommerville, I., Paul, M. (eds.) ESEC 1993. LNCS, vol. 717, pp. 84–99. Springer, Heidelberg (1993)
12. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: ICFP 2008, pp. 383–396. ACM (2008)
13. Grundy, J., Hosking, J., Mugridge, W.B.: Inconsistency Management for Multiple-view Software Development Environments. IEEE Trans. Softw. Eng. 24(11), 960–981 (1998)
14. Herrmannsdoerfer, M., Benz, S., Jüergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
15. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: POPL 2011, pp. 371–384. ACM (2011)
16. ISO/IEC: Information technology – Open distributed processing – Use of UML for ODP system specifications (2009), iSO/IEC19793, ITU-T X.906
17. ISO/IEC: RM-ODP. Reference Model for Open Distributed Processing (2010), iSO/IEC 10746-1 to 10746-4, ITU-T Recs. X.901 to X.904
18. ISO/IEC 42010: Systems and software engineering – Architectural description (2008)
19. Ivkovic, I., Kontogiannis, K.: Tracing Evolution Changes of Software Artifacts through Model Synchronization. In: ICSM 2004, pp. 252–261 (2004)
20. Johann, S., Egyed, A.: Instant and Incremental Transformation of Models. In: ASE 2004, pp. 362–365. IEEE (2004)
21. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update Transformations in the Small with the Epsilon Wizard Language. JOT 6(9), 53–69 (2007)
22. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations (2004)
23. Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. SoSym 7(1), 25–47 (2008)
24. Mens, T.: On the Use of Graph Transformations for Model Refactoring. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 219–257. Springer, Heidelberg (2006)
25. Meyers, B., Wimmer, M., Cicchetti, A., Sprinkle, J.: A generic in-place transformation-based approach to structured model co-evolution. In: MPM 2010 (2010)
26. Moreno, N., Romero, J.R., Vallecillo, A.: An Overview of Model-Driven Web Engineering and the MDA. In: Web Engineering: Modelling and Implementing Web Applications, pp. 353–382. Springer (2007)
27. OMG: Unified Modeling Language (UML) 2.3. Object Management Group, Inc. (2010)
28. Porres, I.: Rule-based Update Transformations and their Application to Model Refactorings. SoSym 4(4), 368–385 (2005)
29. Ráth, I., Varró, G., Varró, D.: Change-Driven Model Transformations. In: Schürr, A., Selic, B. (eds.) MoDELS 2009. LNCS, vol. 5795, pp. 342–356. Springer, Heidelberg (2009)

30. Rivera, J.E., Vallecillo, A.: Representing and Operating with Model Differences. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 141–160. Springer, Heidelberg (2008)
31. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010)
32. Ruiz-Gonzalez, D., Koch, N., Kroiss, C., Romero, J.R., Vallecillo, A.: Viewpoint synchronization of UWE models. In: MDWE 2009, pp. 46–60 (2009)
33. Song, H., Huang, G., Chauvel, F., Zhang, W., Sun, Y., Shao, W., Mei, H.: Instant and Incremental QVT Transformation for Runtime Models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS 2011. LNCS, vol. 6981, pp. 273–288. Springer, Heidelberg (2011)
34. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. J. Vis. Lang. Comput. 15(3-4), 291–307 (2004)
35. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.: Refactoring UML Models. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 134–148. Springer, Heidelberg (2001)
36. Vermolen, S., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: SLE 2011. Springer (2012)
37. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic Model Synchronization from Model Transformations. In: ASE 2007, pp. 164–173. ACM (2007)
38. Zhang, J., Lin, Y., Gray, J.: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In: Model-driven Software Development—Research and Practice in Software Engineering, pp. 199–217. Springer (2005)