On Leveraging UML/OCL for Model Synchronization

Robert Bill
TU Wien, Institute for Software Technology and Interactive Systems
Favoritenstraße 9-11
A-1110 Wien
bill@big.tuwien.ac.at

Martin Gogolla
Universität Bremen,
Department for Mathematics and Computer Science
PO Box 330440
D-28334 Bremen
gogolla@tzi.de

Manuel Wimmer
TU Wien, Institute for Software Technology and Interactive Systems
Favoritenstraße 9-11
A-1110 Wien
wimmer@big.tuwien.ac.at

ABSTRACT

Modelling complex system often results in different but overlapping modelling artifacts which evolve independently. Thus, inconsistencies may arise which lead to unintended effects on the modelled system. To mitigate this situation, model synchronization is seen as a recurring and crucial maintenance task which requires to restore consistency between multiple models using the most suitable changes. Currently, different languages and tools are used for inter-model consistency management than for intra-model consistency where UML/OCL is an accepted solution. Consequently, the result of synchronizing models solely based on inter-model constraints might result into inappropriately evolved models w.r.t. intra-model constraints.

In this paper, we present a synchronization model formalized in UML/OCL which covers explicit consistency and change models including costs and which considers both, inter-model and intra-model constraints at the same time. Instances of this synchronization model represent successful synchronization scenarios. In particular, models can be synchronized, also taking into account their predecessor versions, by finding a constraint violation-free extension of a partial model including those instances which may be optimized for minimal cost. We prototypically implemented this approach using a model finder to automatically retrieve synchronized models and the change operations to compute them by completing the partial model.

Keywords

model consistency; model synchronization; model evolution

1. INTRODUCTION

Modern systems are inherently much more complex to design, develop, and maintain than classical systems due to different properties such as size, heterogeneity, distribution, and multi-disciplinarity. One way to cope with such complexity is by resorting on model-driven engineering approaches [4,5] and dividing the engineering activities 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 [26, 27]. There are more and more approaches which allow to define different views of a system resulting in partially overlapping models. Unfortunately, this separation of concerns by using different viewpoints, potentially expressed in different domain-specific modeling languages, comes with the price of keeping those viewpoints consistent [13]. Thus, model synchronization has become an indispensable duty where inconsistencies between different models need to be resolved in an efficient and correct way.

There are already several approaches to handle model integration based on synchronization [23]. However, there is a recurring pattern in most of these approaches: a dedicated language is used to define a mix of synchronization steps and inter-model consistency relationships. There are two potential challenges using such approaches: (*i*) for consistency relationship formulation, the intermingling of consistency relationships and synchronization might make it difficult to specify how to realize context-dependent resolutions of inconsistencies in heavily constrained models, and (*ii*) the increased mental load for modelers who need to learn a new synchronization language.

In this paper, we propose a new methodology for unifying inter-model and intra-model constraints for model synchronization using UML/OCL. We employ OCL in the classical setting for defining intra-model constraints and present a method how to define inter-model constraints based on dedicated UML/OCL models between two models in the spirit of bi-directional model transformation languages such as QVT Relations [25] or TGGs [30]. In order to define general-purpose and domain-specific synchronization properties, we introduce a formalized change model that is the basis for explicitly modeling such properties based on our UML/OCL approach. For instance, the use of different cost functions for changes allows the definition and usage of different synchronization strategies such as least-change and beyond. Having these ingredients, UML/OCL based model finders can be employed to compute the model synchronizations taking into account both inter-model and intra-model constraints and the stated properties which should be fulfilled by the synchronization. We show this by applying a model finder for UML/OCL, which has already proven to be usable for transformations models [16].

The remainder of this paper is structured as follows. In Section 2 we describe the architecture of our synchronization approach, the types of models we consider, and the running example of this paper. In Section 3 we formulate model changes as a UML/OCL model and how to select the synchronization strategy by associating changes with cost functions. Subsequently, in Section 4 we describe how inter-model constraints can be expressed using a consistency model based on UML/OCL. In Section 5 we discuss the prototypical implementation of our approach to automatically find model synchronizations. Finally, in Section 6 we discuss related work and conclude with an outlook in Section 7.

2. MODEL SYNCHRONIZATION ARCHITECTURE AND RUNNING EXAMPLE

In this section, we introduce our model synchronization architecture by-example.

2.1 Model Synchronization Architecture

A synchronization problem, as depicted in Figure 1(a) and (b), occurs when two models, which were potentially consistent in their current state, are subsequently changed independently leading to potential inconsistencies. To synchronize both models, the goal is to find a suitable set of changes for both models to make them consistent again. As there may be a huge amount of different change sets to reestablish consistency, the question also arises which one is the most appropriate change set for a given situation. Before we go into details on this aspect, we discuss the general architecture of our approach, in particular, how we represent the model synchronization problem by utilizing change and consistency models.

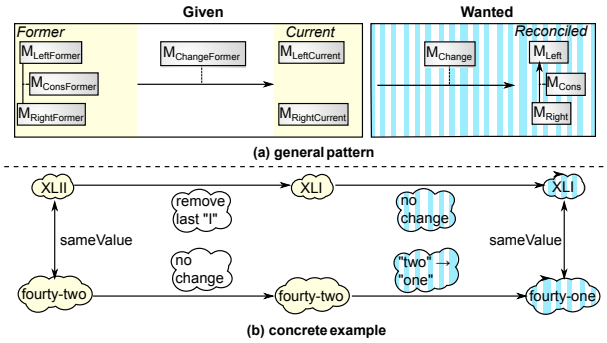

Figure 1: Model synchronizing using change models and consistency models: (a) general pattern and (b) concrete example.

In the following, we will describe the general architecture of our synchronization model as depicted in Figure 1. Our approach synchronizes models by consistently completing a synchronization model containing the history of the models ($M_{\text{LeftFormer}}$, $M_{\text{RightFormer}}$, $M_{\text{LeftCurrent}}$ and $M_{\text{RightCurrent}}$), the previous consistency relation $M_{\text{ConsFormer}}$, which might not be a valid consistency relation in the current state, the previous change set $M_{\text{ChangeFormer}}$, the wanted state of the models M_{Left} and M_{Right} with a consistency relation M_{Cons} and changes M_{Change} leading to a consistent state. In general, model finders like the USE ModelValidator[1] search for instances of a metamodel. A model finder can synchronize models by consistently completing synchronization models consisting of former and current models with consistent change and wanted models by adding new objects, associations and attributes. Since a model finder may complete the model in many ways, including the application of `delete` changes, in the extreme case to delete the complete model, we also foresee to model change costs to guide the model finder in the right direction. In the concrete example shown in Figure 1, the model finder may find and add two change objects (no change; "two" → "one") and two consistent state objects (XLI; forty-one).

Summing up, in contrast to many other approaches, we do not use explicit consistency restoring transformation rules for synchronization, but just the declarative consistency models. Among others, this has the advantage that this approach is, in principle, not limited to synchronize two models, but any number of models, with or without circular consistency dependencies between them. Since both model states are considered at the same time, the typical problem of making model A consistent with model B, thus requiring additional changes in model B which themselves require additional changes in model A etc. can never occur. Also, we do not use any model diffing algorithm, but again a fully declarative change model, i.e., the constraints which have to hold for a model explaining the difference between two model versions. We copy left, right and consistency model two times to build former and current model while removing all constraints, including multiplicity constraints. This is done to ensure that (*i*) the change model actually describes changes between model versions and (*ii*) the model finder can find a consistent model extension defining only the wanted model.

For completeness reasons please note when integrating left models and right models into global models, name clashes may occur. This can be simply avoided by pre- or postfixing names of classes and properties. Since this step is trivial but might clutter the overall architecture and descriptions, it is not further discussed throughout the paper.

2.2 Running Example

As a running example[2], let us consider two viewpoints for developing and maintaining computer networks as depicted in Figure 2: (a) the requirements viewpoint and (*b*) the implementation viewpoint. The metamodels realizing these two viewpoints are illustrated in Figure 3. When taking a closer look on Figure 2, we see that the left model stores the requirements of a computer network. There are named machines which provide a certain amount of communication speed and others which consume a certain expected amount of data. The right part of Figure 2 contains a model of the implemented system which does not only include servers providing data and computers consuming data, but also routers and cables needed to transfer the data from servers to computers. There are two types of cables, namely `GlassFiberCable`s for high-speed connections and `Copper-Cable`s for low-speed connections. There are constraints on the right model to ensure that (*i*) each server can serve the cables it is connected to, (*ii*) each computer gets enough bandwidth for its needs and (*iii*) each router does not produce any data and thus can fulfill the outgoing bandwidth with the incoming bandwidth and its own processing speed.

For the given example, the models' consistency relationship is defined such that each provider needs a server with the same name and at least that speed and each consumer needs a computer with the same name and the same speed and vice versa.

In Figure 2, also the evolution scenario for our running example is shown. It is assumes that the models were changed independently from each other. In the requirements model, the provider `p1` should be made ready for the future and gets a higher speed. Thus, its speed attribute was increased from 3 to 4. At the same time in the implementation model changes were performed. It was discovered that the computer `w2` was never used and thus it was removed together

[1] `https://sourceforge.net/projects/useocl/files/Plugins/ModelValidator/`

[2] A slightly simplified version of the example can be downloaded from `http://cosimo.big.tuwien.ac.at/findsync/`

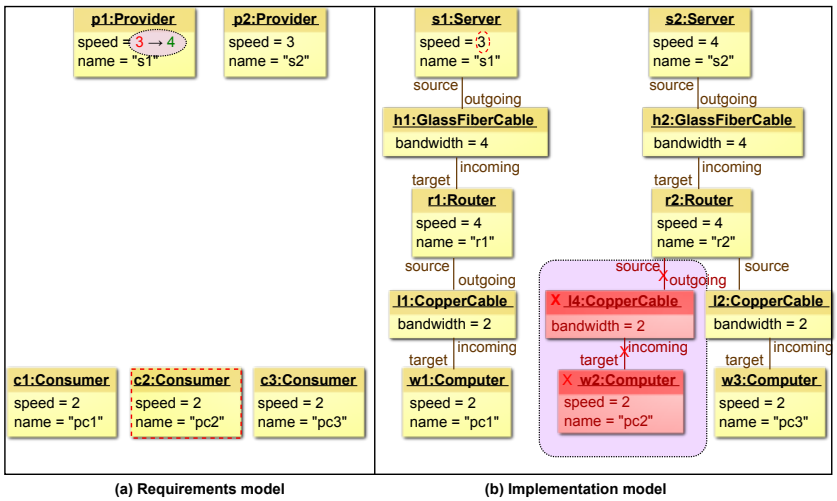

Figure 2: Example synchronization scenario: a requirements model, its corresponding implementation model, and their uncoordinated evolution.

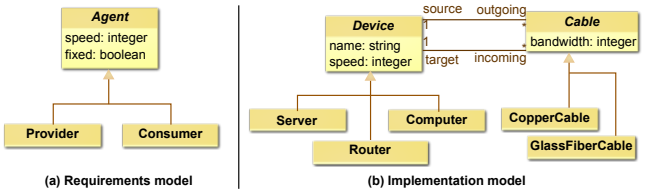

Figure 3: Metamodels of the running example: (a) requirements metamodel and (b) implementation metamodel.

with its connected cable `l4`. Now there are two violations w.r.t. the aforementioned consistency relationship: The speed of server `s1` is not sufficient and the consumer `c2` has no correspondence on the right side. Of course, one now may come up with some reconciliation actions for this small example. One obvious model synchronization would be to delete consumer `c2` in the requirements model and to set the speed variable to 4 for server `s1`. For larger examples, of course automation support is needed in order to reason about appropriate synchronizations. How this is realized by our approach following the previously described architecture is the content of the following sections.

3. CHANGE MODEL

Our change model approach consists of an abstract part which is the same for all modeling languages and a language-specific part which is generated for each used metamodel individually. In general, we follow the ideas presented in [9] to generate language specific change models to represent changes between two models. However, we also go beyond the ideas presented in [9], by providing also the conditions for finding a valid change model. Thus, we do not only explicitly model the abstract syntax of change models but also explicate their semantics in terms of OCL constraints.

3.1 Abstract Change Types

Figure 4 shows the abstract structure of our change model. There are two types of changes, namely atomic changes and composite changes [21]. An atomic change connects the original object to the revised object. If an object has been deleted, there is no revised object. If an object has been created, there is no original object. If an object is preserved, there are both original and revised objects. In the change model, every original object must specify what happens in the future and every revised object must specify what happened to it in the past. For set-valued features, the set of future values must be equal to the set of past values with the deleted values removed and the created features added. For bag features, the number of occurrences of a future feature must be equal to the occurrence count of this feature in the past object plus the sum of all added feature counts minus the sum of all deleted feature counts. For ordered features, the sequence resulting from the deletion of all deleted elements from the past object feature must be the same as the sequence resulting from deleting all created elements from the future object feature which is expressed by inserting values at specific list indexes, sorted from bottom to top, to the common base sequence.

A composite change builds higher level changes from lower level changes [21]. For example, type changes cannot be directly represented with atomic changes. A general cast combines an atomic delete change and an atomic create change to express that semantically, the object has not been deleted and another created, but the object is still the same. Similarly, a general move combines an atomic feature delete change and an atomic feature create change for the same feature and the same value to specify that the feature has moved and was not deleted and re-added. A feature change is a composite change defined by an OCL operation which takes several parameters and has a postcondition defining which changes are done to the object. The last change type resembles composite changes as proposed in literature

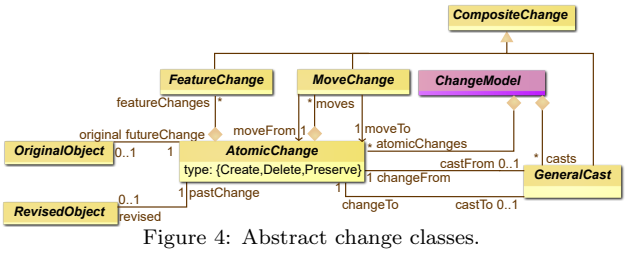

Figure 4: Abstract change classes.

(e.g., [19]) while the others could also be semantically regarded as atomic changes, but are regarded as composite changes from an implementation point of view. In principle, such explained changes could be derived (semi-)automatically for various postconditions [24]. If such technologies are used, only the cost remains to be specified.

Currently, these are the only composite changes supported, but others might be added in the future. Also, it is currently not possible to let a feature change explain features of multiple objects.

3.2 Language-specific Change Language

Beside the change model itself, the metamodel is duplicated twice as well where class and feature names are changed to avoid name clashes. In particular, all model classes and associations of the previous version might be suffixed with `Prev` while the metamodel to store current objects and feature values is suffixed with `Ori` and names do not change for the synchronized model.

Two parallel class hierarchies resembling the class hierarchy of the original metamodel are created. In the `Change`-hierarchy, all classes are replicated with a Change prefixed name inheriting from `AtomicChange`. For every structural feature in the original class *X*, two structural features of the same type *T* named as the original structural feature *attr* plus a suffix of `Add` and `Del` are created in the `Change`-class as shown in List. 1. They denote which values have been added or deleted by the change operation. The `Add` attribute is derived as values which exist in the revised model, but not the original model and the `Del` attribute is derived as values which exist in the original model, but not the revised model. Unique features generate a Set type, non-unique features generate a Bag type. The derived attributes *attr*`AddCostly` and *attr*`DelCostly` contain unexplained addition and deletion changes. Associations are derived in a similar fashion. To ensure that added and deleted classes can be derived via set difference, the change objects have to be used instead of the original objects, i.e. `revised.`*attr*`.futureChange`/`original.`*attr*`.pastChange` is used instead of `revised.`*attr*/`original.`*attr*.

List. 1: Change calculation for unordered features.

```
context ChangeX:
 attrAdd: [Set|Bag](T) derived = if revised =
     null then [Set|Bag]{} else revised.attr->
     asBag() endif - if original = null then [Set
     |Bag]{} else original.attr->asBag() endif
 attrDel: [Set|Bag](T) derived = if original =
     null then [Set|Bag]{} else original.attr->
     asBag() endif - if revised = null then [Set|
     Bag]{} else revised.attr->asBag() endif
 attrAddExpl: [Set|Bag](T) derived =
     featureChanges.attrAddExpl->union(
     castChanges.attrExpl)->union(movedFrom.
     attrExpl)
 attrDelExpl: [Set|Bag](T) derived =
     featureChanges.attrDelExpl->union(
     castChanges.attrExpl)->union(movedTo.
     attrExpl)
 attrAddCostly: [Set|Bag](T) derived = attrAdd -
     attrAddExpl
 attrDelCostly: [Set|Bag](T) derived = attrDel -
     attrDelExpl
```

For every ordered structural feature, three additional features named as the original structural feature plus `AddEl`, `DelEl` and `Same` are created in the `Change` class as shown in List. 2. The feature `*Same` contains a base feature value which is extended by inserting the values in `*AddEl` to get the revised feature value and extended by inserting the values in `*DelEl` to get the original feature value. The feature `*Same` has the same type as the original feature, but `*AddEl` and `*DelEl` are sequences of pairs of `(Integer,`*T*`)`. List. 2 shows the connection between all features.

List. 2: Additional change calculation for ordered features.

```
context ChangeX:
  attrAddEl: Sequence(Tuple{i: Integer, v: \myvar
      {T}})
  attrDelEl: Sequence(Tuple{i: Integer, v: \myvar
      {T}})
  attrSame: Sequence(T)
  attrMoved: Bag(T) derived = attrAddEl.v -
      attrAdd
  attrMovedExpl: Bag(T) derived = featureChanges.
      attrMovedExpl->union(castChanges.
      attrMovedExpl)

  inv orderedSequence: Set{attrAddEl,attrAddEl}->
      forAll(s | s->isUnique(i) and s->sortedBy(i
      ) = s)
  inv addToRevised: (revised = null) or (revised.
      attr = attrAdd->iterate(t, full =
      <@attrSame | full->insertAt(t.i, t.v)))
  inv delFromOriginal: (original = null) or (
      original.attr = attrDel->iterate(t, full =
      <@attrSame | full->insertAt(t.i, t.v)))
```

The sequence `*Same` is an auxiliary sequence that represents a common sequence between both ordered features. The feature *attr*`AddEl` contains not only elements which have been added to the revised feature from the original feature, but also elements which are not in the common sequence because they have changed their position. Elements which are moved are those which are not in the common sequence but whose value has not been added to the common bag as well. The constraints guarantee a certain order of insert applications and ensure that the original and revised feature value actually can be built as previously described. Unlike the change object above, these feature values are not necessarily directly determined by original and revised objects. In fact, the sequence in the *attr*`Same` feature might be too short; then too many moves are determined. However, a larger number of moves yields higher costs and thus a solution with less superfluous moves will be preferred.

Each `Change` object also has a `original` and `revised` association of the corresponding classes in the original and the revised model which redefines the `original` and `revised` association in `AtomicChange` to ensure the correct type.

Consider Figure 5 as example for an excerpt of the change model for atomic changes of our running example. Since `Device` contains a name attribute, `ChangeDevice` contains two attributes to define which names have been added and deleted. Also, the `incoming` and `outgoing` associations are used to connect to added and deleted cables. The class `ChangeRouter` adds two attributes for the `maxSpeed`. It does not replicate the `name` attribute changes since they are available in the superclass.

Let us now consider another excerpt of an instance of the change model as depicted in Figure 6. There are two original objects, the original cable `oc1` and the original router `or1`. The cable has been deleted, so the speed attribute is deleted as well as the incoming connection. The router

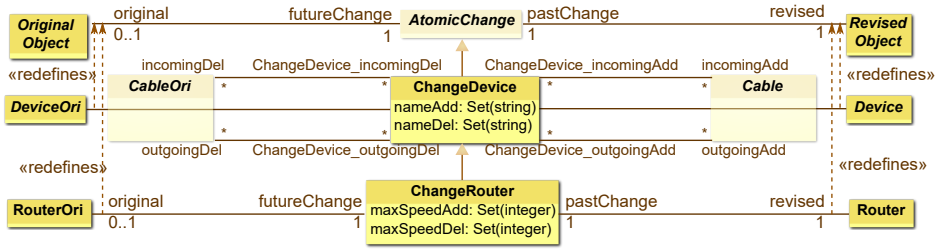

Figure 5: Change model excerpt 1 of the running example.

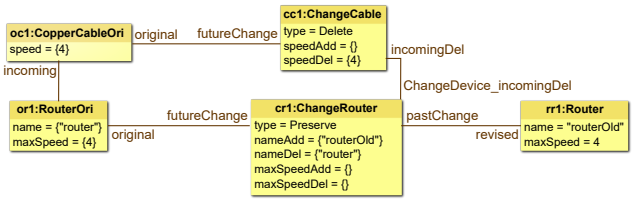

Figure 6: Change model excerpt 2 of the running example.

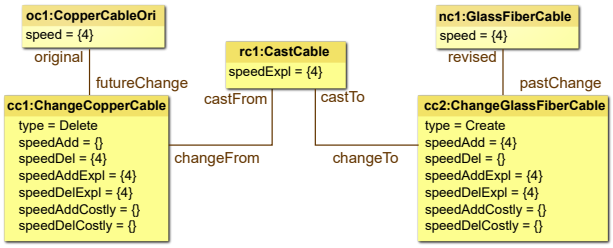

Figure 7: Cast change excerpt for the running example.

just has been taken out of usage, so its name was changed to *routerOld* from *router*. In terms of attribute changes, this equals the addition of *routerOld* to the name and the deletion of *router*. The speed has not been changed, so no additions or deletions are defined there. The fact that all constraints, including multiplicity constraints, are removed from all but the target model is indicated by the set-valued name and speed attributes for the original objects.

Composite changes.

In the `Cast` class hierarchy similarities between cast objects are stored. For every structural feature in the original class, a structural feature of the same type named as the original structural feature plus a suffix of `Expl` is created in the `Cast` class. It denotes which values are the same in both objects and is defined by the intersection of values between original and revised object and thus explains those ostensible changes. List. 3 demonstrates the structure of `Cast` objects. An invariant ensures that the cast object is connected to a single object which is deleted and an object which is added.

List. 3: Change calculation for casts

```
context CastX:
 attrExpl: [Set|Bag](T) derived = castFrom.
     attrDel->intersection(castTo.attrAdd)
 inv changeTypes: changeFrom.type = ChangeType.
     Delete and changeTo.type = ChangeType.Create
```

Consider Figure 7 for an example of such a cast change for our running example. The model has only changed by retyping `oo1` from `CopperCable` to `GlassFiberCable`. This is expressed as two changes, namely deleting the CopperCable and creating a GlassFiberCable with the same values. Since the speed has not changed, the same value was deleted for the CopperCable that was added to the GlassFiberCable. Thus, the cast explains this value change and it is not considered for costs.

Similarly, a `Move` change for a certain aggregation attribute connects two change objects where an association to a certain object X was deleted in one object, but created in another object. However, there is no requirement that any object must be deleted or created.

List. 4: Change calculation for moves

```
context MoveX:
 attrExpl: [Set|Bag](T) derived = moveFrom.
     attrDel->intersection(moveTo.attrAdd)
```

In the example depicted in Figure 8, an incoming edge was moved from the Router `or1` to `or2`. The two router change objects show that the edge was deleted (`cr1`) and added (`cr2`). The move object `m1` connects both changes. The intersection of `incomingAdd` for `cr2` and `incomingDel` of `cr1` is `cc1`, so this is the attribute change explained by the move object. Thus, it is also explained in both `cr1` and `cr2` and no change of `incoming` is a `costly` change.

For every operation O in a class X, a corresponding `FeatureX_O` class is created which inherits from `FeatureX`. This class contains an attribute for each operation parameter and it has a bidirectional association to the change class of the class the operation is defined in. The postcondition of the operation X is converted into an invariant of the generated class by changing the `self` context of each `@pre` expression to the `original` object of the associated change while other `self` contexts are transferred to the `revised` object of the associated change. For features changed in the referenced object, this class contains a `T[Add|Del]Expl` attribute for each attribute in the original class. Postconditions which have the pattern `feature = feature@pre->including([expr])` or `feature = feature@pre->excluding([expr])` are converted into the invariant `T[Add|Del]Expl = Set{[expr]}`.

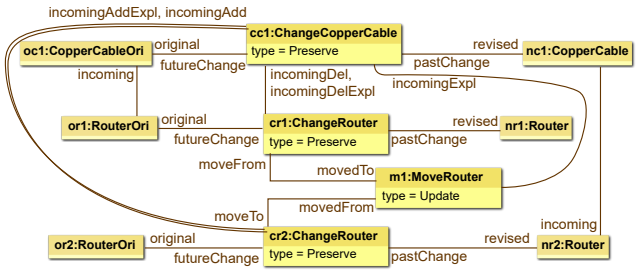

Figure 8: Move change excerpt for the running example.

Additionally, invariants are generated to ensure all features not occurring in any such postcondition are empty. An invariant is generated to let the cost attribute of the class be equal the operation result.

List. 5: Simple change operation defined in UML/OCL.

```
context Router::disconnectServer(c: Cable)
  post disconnectedIncoming: incoming = (
       incoming@pre)->excluding(c)
  post disconnectedOutgoing: outgoing = (
       outgoing@pre)->excluding(c)
```

This example shows a simple operation which just disconnects a single cable from a router. The postconditions state that after this operation has been called, the disconnected cable is not connected to the server any more.

List. 6: Translation of the change operation of List. 5 into the synchronization model

```
class FeatureRouter
attributes
  ...
end

class FeatureRouter_disconnectServer <
    FeatureRouter

constraints
 inv disconnectedIncoming:  incomingDelExpl = Set
     {c}
 inv disconnectedOutgoing:  outgoingDelExpl = Set
     {c}
 inv noChange: incomingAddExpl = Set{} and ...
end

association
    assoc_FeatureRouter_disconnectServer_c
    between
  FeatureRouter_disconnectServer[*] role
      FeatureRouter_disconnectServer_c
  ChangeCable[1] role c
end
```

This example is translated as shown in List. 6. For each postcondition with the specified pattern, an invariant is generated which explains the according changes. The third invariant guarantees that all other attributes are not changed due to this operation call. The operation parameter is translated as association. Like for atomic changes, a parameter of type Cable is translated to a ChangeCable.

3.3 Change Costs

While the change model describes what has changed, for synchronization purposes, the severity of changes should be known. To describe this aspect as well, a cost function is added to each `Change`-class. The specification of the exact cost function depends on the synchronization strategy. In we following, we discuss how different synchronization strategies can be implemented with our approach.

Least change.

The least change [22] strategy cost function sums up the size of all unexplained add and deletion collections in the `Change` objects and adds 1 if the change type is `Delete` or `Create`. If objects have been casted, one change should not be counted. This function can be easily changed to support a **least creation** or **least deletion** strategy by adding a higher value instead of 1 to `Delete` or `Create` changes. Positive costs ensure hippocraticness [32], i.e., that the synchronization does nothing for consistent models, because every change would have positive cost while consistent models would require no change which has no and thus less cost.

List. 7: Least change cost calculation.

```
class ChangeObject
attributes
  costObject: Integer derived = if type =
       ChangeType.Create or type = ChangeType.
       Delete then if castTo != null then 0 else 1
        endif else 0 endif
end

class ChangeCable
attributes
  costCable: Integer derived = speedAddCostly->
       size() + speedDelCostly->size() +
       sourceAddCostly->size() + sourceDelCostly->
       size() + targetAddCostly->size() +
       targetDelCostly->size()
end

class CostSummer
attributes
  cost = ChangeObject.allInstances().costObject->
       sum() + ChangeCable.allInstances().
       costObject->sum() + ...
end
```

List. 7 shows an excerpt of the implementation of the least cost strategy for the running example. The cost of deleting or adding objects is defined as attribute of the `ChangeObject` class. If an object is casted to another object, the cost is not accounted. The `ChangeCable` class calculates the cost by summing up the costs of its direct attributes. The `CostSummer` calculates the total costs by summing over all cost attributes.

Real costs.

Models might be representations of the real world where changes induce real costs, e.g., working time for switching cables or the costs of a new server. Then, it is natural to use a cost function resembling real costs. These costs will typically occur as operation costs.

List. 8: Domain-specific cost calculation.

```
context Router::disconnectServer(c: Cable)
  body: if c.isKindOf(CopperCable) then 5 else 10
        endif
==>
class FeatureRouter_disconnectServer
attributes
  cost: Integer derived = if (c.revised.isKindOf(
       CopperCable)) then 5 else 10 endif ...
```

List. 8 shows a simple example of operation costs. It might be more difficult to disconnect glass fiber cables than to disconnect copper cables, so the costs could be twice as high. Such costs are transformed into an invariant as explained in the previous section.

Organizational asymmetry.

If model A is considered more important than another model B, changes in model A should be avoided. By assigning much higher costs to changes in the model A, changes in

model A are avoided but enforced in model B. For instance, this strategy may be also of relevance for the running example of this paper.

Avoiding undos.

Usually, a model was modified for a good reason. Thus, the synchronization step should not return to the previous model version. To achieve that, additional costs can be introduced if some attributes in the target re-appear: (*i*) if they were deleted before or vice versa using the sum of symmetric difference sizes of `attrAdd` and `attrDelete` of future and past change objects in the current model, (*ii*) if objects are deleted when they were created before by counting create/delete pairs, and (*iii*) if objects are created when they were deleted by counting created objects in the target model which are similar to deleted objects in the former model.

List. 9 shows a simple example for avoiding undos. Each attribute which is reassigned to the same value as before will induce a cost of 10 because both the old value is added which should not be the case and the newly set value is deleted. If an attribute whose value was changed has its value changed again the cost would only be 5. If an object is deleted which was created before, the cost increases by 100. Likewise, if an object is recreated which has the same name as an object which was deleted, the cost is increased by 200.

List. 9: Costs for avoiding undos.

```
class ChangeRouter
attributes
  avoidRedoCosts: Integer derived = if original.
      pastChange.type = ChangeType.Create then 5*
      original.pastChange.speedDel->intersection(
      speedAdd)->union(original.pastChange.
      speedAdd->intersection(speedDel))->size()
      +...
  else if type = ChangeType.Delete then 100 else
      0 endif endif + if type = ChangeType.Create
       and ChangeRouterOri.allInstances()->select
      (c | c.type = ChangeType.Delete).original->
      select(r | r.name = revised.name) then 200
      else 0 endif
end
```

Retaining existing traces.

A consistency model might be just descriptive or prescriptive. In the former case, changes in the consistency model should have no cost, while in the latter case, consistency model changes, especially deletions, might have high costs if the associated objects were not deleted.

Please note that there are design decisions to take which synchronization properties to define as constraints and which via costs. For example, undos could not only be avoided using costs as described in List. 9, but also by just using a constraint like in List. 10. However, then no synchronization would be possible at all if conflicting model changes would have been performed which could only be resolved by undoing a change. This also holds in the case of asymmetric changes. If a model should never change, this model can just be used as target models with additional constraints that all attributes and associations should remain equal.

List. 10: Constraints for avoiding undos.

```
class ChangeRouter
constraints
  inv avoidRedo: if original.pastChange.type =
      ChangeType.Create then type <> ChangeType.
      Delete and if original.pastChange.type =
      ChangeType.Delete then type <> ChangeType.
      Create endif
end
```

4. CONSISTENCY MODEL

In this section, we will show how inter-model consistency concerns are modeled with UML/OCL. In general, we follow the main idea of triple graph grammars (TGGs) [30] and triple patterns [10] to build an explicitly modeled structure between two models. If the models are defined with UML/OCL we end up with an unified representation for intra- and inter-model concerns.

As we now show in this section, by using UML/OCL it is possible to describe the interconnection between models by additional associations, constraints and possibly even additional other elements such as classes and attributes.

Since we aim for a non-intrusive addition of synchronization logic, we assume open models or some form of module import, at least for defining new associations. In the following, `@override` denotes the extension of the specified class with attributes and associations contained by the consistency model. We now show several examples how to employ UML/OCL to define consistency models in terms of correspondences.

One-to-one correspondences.

One-to-one correspondences are easily defined in UML/OCL by adding an association between the corresponding classes having as lower and upper bounds 1 on both ends. Figure 9 shows an example where classes correspond based on name equality. Such constraints can either be added to a class (as shown in graphical syntax) or added to the association (as shown in textual syntax) as preferred.

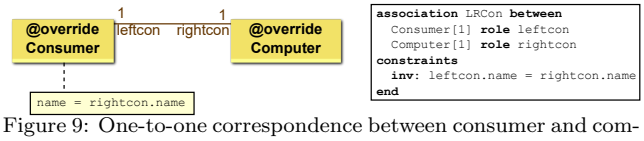

Figure 9: One-to-one correspondence between consumer and computer in graphical and textual syntax.

One-to-many correspondences.

One-to-many correspondences can also be expressed by associations. Figure 10 shows an example where a server in the requirements model might correspond to a whole cluster in the implementation model. In particular, the constraint `inv1` specifies that a single cluster provider corresponds to many nodes if all servers have exactly one router and nothing else as target of their outgoing cables. Then, all servers connected to this router constitute the cluster, else a provider corresponds to a single cluster.

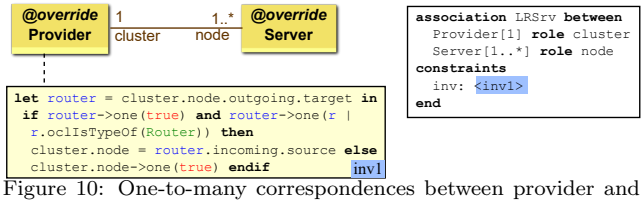

Figure 10: One-to-many correspondences between provider and server in graphical and textual syntax.

Many-to-many correspondences.

Many-to-many can also be expressed either by using n-ary associations of UML or by introducing additional classes which connect more than one class on both sides by associations. Figure 11 shows a more complex correspondence between provider and server. Servers may be either part of a typical cluster where a service is provided by multiple servers or part of a virtual cluster, where services may be assigned to many servers and each server might handle multiple services.

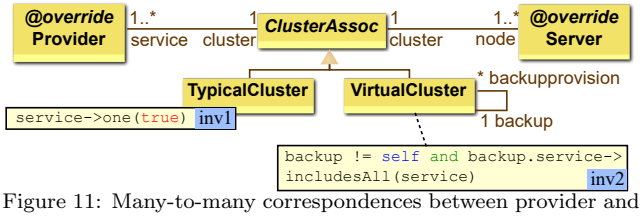

Figure 11: Many-to-many correspondences between provider and server in graphical syntax.

Correspondence dependencies.

A full consistency model may also contain dependencies between correspondences modeled as associations. Figure 11 shows that each virtual cluster requires a second virtual cluster as backup which must provide the all services.

5. TOOL SUPPORT BASED ON THE USE MODEL VALIDATOR

The presented synchronization approach has been prototypically implemented as an USE plugin and can be downloaded from `http://cosimo.big.tuwien.ac.at/findsync`. While the consistency model has to be hand-crafted, the USE plugin merges the different involved models automatically, generates the change model, and finds the minimum cost synchronized models. To give an idea about the complexity of the automatically produced models, consider the running example of this paper for which the generated model contains 79 classes, 124 associations, and 379 invariants.

The costs are currently fixed to a constant for each primitive change operation. Due to limitations in the USE model validator, some constraints and structures had to be reformulated and only set-typed features are supported. The optimization process is run by iteratively finding a model completion with less cost than the previously found solution. If the model finder is not able to find any better solution, the cost-optimal solution has been found. Bounds have to be given for the number of instances of each class, thus cost-optimality is only guaranteed with regards to these bounds.

5.1 Evaluation

We ran our performance evaluation using a simplified synchronization scenario of the one shown in Figure 2 on a Intel i5-6500 3.2 GHz machine with 64 GB RAM, running Ubuntu Linux 16.04. For simplicity reasons, all costs were assumed to be one.

[3]Version '15, available at: `http://fmv.jku.at/lingeling/lingeling-bal-2293bef-151109.tar.gz`

[4]Version 2.2.0, available at: `http://minisat.se/downloads/minisat-2.2.0.tar.gz`

[5]Version 2.3.1, integrated in the ModelValidator jar, available at: `http://forge.ow2.org/project/download.php?group_id=228&file_id=17186`

Solver	Time (FS)	Costs (FS)	Time (OS)
lingeling[3]	4 min 52 sec	21	38 min
plingeling[3]	6 min 26 sec	17	77 min
MiniSat[4]	8 min 49 sec	46	OOM
Sat4J[5]	65 min	3	85 min

Table 1: Execution times and costs for the running example (FS: first solution, OS: optimal solution, the cost for producing the optimal solution is 2, OOM: out of memory).

Table 1 shows the average runtime of three runs to find any solution and a cost-minimal solution with different solvers. The runtimes vary greatly with each run, but still some trends can be observed. The runtimes also indicate that with the current state of the USE model validator, the approach cannot be directly used to find a good synchronized instances within a reasonable time frame for large models. Still, we see a huge difference between solvers. While lingeling and plingeling are quite fast finding the first solution which already provide a good quality, Sat4J requires a longer time to find any solution, but this first solution is nearly the optimal one. MiniSat had quite some problems for the studied scenario. It produced did not produce appropriate solutions and went out of memory without finding the best solution.

In addition to computing synchronizations, we see also an alternative usage scenario for the presented approach. It can be also used to validate whether an existing synchronization found by another tool is a valid synchronization by using the validation capabilities of OCL. Here, the full approach can be used since USE is able to check all constraints used in the approach.

5.2 Threats to Validity

In the evaluation, we have only shown the general feasibility of the approach, but not that it is fast enough for larger instances. We assume that faster, maybe heuristic model finding approaches may significantly improve the performance but we do not have any concrete evidence for this yet. Thus, we can not claim that the performance can be actually improved in order that the approach scales to the synchronization of larger real-world models in the matter of minutes.

6. RELATED WORK

While transformation models using UML/OCL have been already introduced back in 2006 [3], to the best to our knowledge, the model synchronization aspect has not been considered in such type of models. However, since model synchronization is an important topic in model-driven engineering, there is already a great variety of approaches to support different synchronization scenarios. For example, there is specific work on model synchronization by specifying how to deal with inconsistencies [2, 11, 14] and by using bidirectional transformation languages. For instance, triple graph grammars [20, 30] are often employed for model synchronization scenarios such as reported in [1, 15, 17]. In a similar fashion, QVT Relational [33] allows synchronization, also in conjunction with unidirectional transformation languages such as ATL [22]. Furthermore, there are other, mostly rule-based, approaches available (cf. [18] for a survey). One benefit of our approach when comparing it to existing work

is that we make use of an explicit change model which allows to adjust the synchronization strategy in a domain-specific way.

There are also some existing approaches using constraint solving for model synchronization such as the Janus Transformation Language (JTL) [8], also with compressed state space [12], and the CARE approach [29]. JTL [8] is using answer set programming (ASP) to find a synchronized result. In contrast to JTL, we do not map a relation-based language into ASP, but we aim to describe everything with UML/OCL. In addition, we also provide a method to prefer certain change results over others by having cost models attached to change models. In previous work, we have presented CARE [29], an approach using a constraint solver (ASP) to re-synchronize models with their evolving metamodels. However, CARE is a specific approach for the metamodel/model co-evolution problem. The approach presented in this paper may be also employed in the future to reproduce the results of the CARE approach. An alternative approach for metamodel/model co-evolution is presented in [28] where the variability of different model migration solutions is formalized as a feature model. We see this research direction as an interesting line for future work as this would allow to concisely report equally good model synchronization solutions, i.e., having the same cost, to the user.

There have been many change models proposed in the literature, e.g., [6, 9, 31, 34]. However, to the best of our knowledge, we do not know any constraint-based approach to define change models.

7. CONCLUSION AND FUTURE WORK

In this paper, we have shown how to transform the problem of model synchronization into the problem of defining a suitable consistency model and a change model with UML/OCL. This can be used to check whether a given synchronization strategy was performed successfully and to find synchronization strategies by model completion. As the evaluation has shown, the approach can be used to build an incremental transformation which is used for our approach itself. The approach has also been prototypically implemented which shows that automation is feasible.

In the future, we need to further develop both the conceptual approach and the implementation. Sometimes, you explicitly require a change in a model if a specific change happened in another model [35]. Within this approach, this would correspond to a consistency model constraining change objects. A sensible consistency model or an alternative representation translated into a consistency model for that has to be found. The approach could be adapted to support metamodel/model co-evolution by (*i*) translating the metamodel to a model a consistency relation between this generated model and the model to co-evolve or (*ii*) allowing the target metamodel to differ from the source metamodel. The first approach would be more general since it would not only cover model evolutions based on metamodel changes but may also cover metamodel evolutions based on model changes, but at the same time more challenging since OCL constraints would have to be translated into a model and interpreted using OCL. Thus, we are currently evaluating the latter approach.

The performance and scalability of the implementation has to be assessed in order to increase its practical usefulness. Including optimization strategies in the USE model validator for this problem, which has been done in the past for similar problems, may increase the performance by orders of magnitude and make this approach usable in practice. Also, we need to evaluate in which cases, if any, a suitable cost definition could let our approach find *least surprising* changes [7].

8. ACKNOWLEDGMENTS

Acknowledgment: This work has been funded by the Vienna Business Agency (Austria) within the COSIMO project (grant number 967327) and by the Christian Doppler Forschungsgesellschaft, the Federal Ministry of Economy, Family and Youth and the National Foundation for Research, Technology and Development, Austria.

9. REFERENCES

[1] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In *Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA)*, volume 8569 of *LNCS*, pages 1–17. Springer, 2014.

[2] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations - change (in) the rule to rule the change. *SoSyM*, 11(3):431–461, 2012.

[3] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.

[4] J. Bézivin, R. F. Paige, U. Aßmann, B. Rumpe, and D. C. Schmidt. Manifesto - model engineering for complex systems. *CoRR*, abs/1409.6591, 2014.

[5] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[6] E. J. Burger. Flexible views for view-based model-driven development. In *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*, pages 25–30. ACM, 2013.

[7] J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Towards a principle of least surprise for bidirectional transformations. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, pages 66–80, 2015.

[8] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 183–202. Springer, 2011.

[9] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[10] J. de Lara, E. Guerra, and P. Bottoni. Triple patterns: Compact specifications for the generation of

operational triple graph grammar rules. *ECEASST*, 6, 2007.

[11] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. Change management in multi-viewpoint system using ASP. In *Workshops Proceedings of the International IEEE Enterprise Distributed Object Computing Conference (EDOCW)*, pages 433–440. IEEE, 2008.

[12] R. Eramo, A. Pierantonio, and G. Rosa. Managing uncertainty in bidirectional model transformations. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 49–58, 2015.

[13] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *Proceedings of the 4th European Software Engineering Conference (ESEC)*, volume 717 of *LNCS*, pages 84–99. Springer, 1993.

[14] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Trans. Software Eng.*, 20(8):569–578, 1994.

[15] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *SoSyM*, 8(1):21–43, 2009.

[16] M. Gogolla, L. Hamann, and F. Hilken. On static and dynamic analysis of UML and OCL transformation models. In *Proceedings of the Workshop on Analysis of Model Transformations co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, pages 24–33, 2014.

[17] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 7212 of *LNCS*, pages 178–193. Springer, 2012.

[18] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based classification of bidirectional transformation approaches. *SoSyM*, 15(3):907–928, 2016.

[19] M. Javed, Y. M. Abgaz, and C. Paul. Composite ontology change operators and their customizable evolution strategies. In *Proceedings of the 2nd Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn)*, pages 1–12. CEUR-WS.org, 2012.

[20] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, tr-ri-07-284, University of Paderborn, 2007.

[21] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[22] N. Macedo and A. Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *SoSyM*, pages 1–28, 2014.

[23] N. Macedo, J. Tiago, and A. Cunha. A feature-based classification of model repair approaches. *CoRR*, abs/1504.03947, 2015.

[24] P. Niemann, F. Hilken, M. Gogolla, and R. Wille. Assisted generation of frame conditions for formal models. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 309–312, 2015.

[25] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. `http://www.omg.org/spec/QVT/1.0/`.

[26] J. E. Rivera, J. R. Romero, and A. Vallecillo. Behavior, time and viewpoint consistency: Three challenges for MDE. In *Models in Software Engineering, Reports and Revised Selected Papers of Workshops and Symposia at MODELS 2008*, pages 60–65, 2008.

[27] J. R. Romero and A. Vallecillo. Well-formed rules for viewpoint correspondences specification. In *Workshops Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (ECOCW)*, pages 441–443, 2008.

[28] D. D. Ruscio, J. Etzlstorfer, L. Iovino, A. Pierantonio, and W. Schwinger. Supporting variability exploration and resolution during model migration. In *Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA)*, volume 9764 of *LNCS*, pages 231–246. Springer, 2016.

[29] J. Schoenboeck, A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Schwinger, M. Wimmer, and M. Wischenbart. CARE: A Constraint-based Approach for Re-Establishing Conformance Relationships. In *Proceedings of the 10th Asia-Pacific Conference on Conceptual Modelling (APCCM)*, pages 19–28. Australian Computer Society, 2014.

[30] A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 151–163, 1994.

[31] C. Seidl, I. Schaefer, and U. Aßmann. Deltaecore - A model-based delta language generation framework. In *Proceedings of Modellierung*, pages 81–96, 2014.

[32] P. Stevens. Towards an algebraic theory of bidirectional transformations. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT)*, pages 1–17, 2008.

[33] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *SoSyM*, 9(1):7–20, 2009.

[34] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *SoSyM*, 13(1):239–272, 2014.

[35] M. Wimmer, N. Moreno, and A. Vallecillo. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS)*, volume 7304 of *LNCS*, pages 336–352. Springer, 2012.