Høgskulen
på Vestlandet

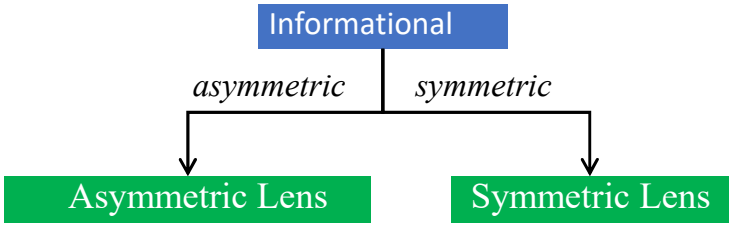
Bidirectional Transformation, Part 2

From Concepts to an Algorithm for Consistency Restoration

Lenses are a formal concept for the management of correlated system components. In this part, we propose a general solution for symmetric update propagation which is applicable in many scenarios.

Recap

- For model spaces $\mathcal{M}_1$ and $\mathcal{M}_2$, we defined a **conceptual base**, which enables consistency restoration of model pairs, avoids loss of information, and differentiates between organizational symmetry and asymmetry



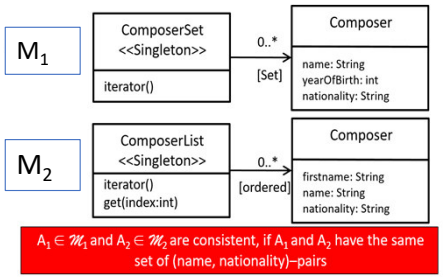
2

Moreover, it is clear that a general implementation for a forward or backward propagation $\overline{R}$: M x M′ -> M′ can be reused for the backward propagation as well as the put-function in an asymmetric situation. I.e. in the sequel we focus on an implementation for $\overline{R}$ only.

The implementation is decomposed into several steps. On the next slide we consider an example for backward propagation, which shows that there are more problems than already found in part 1.

But we have more problems

- Consider the following situation in the composer example (the original setting)
  - A_1 = {(Bach, 1685, German), (Bach, 1714, German), (Ravel, 1875, French)}
  - A_2 = [(Carl Philipp Emanuel, Bach, German), (Johann Sebastian, Bach, German), (Maurice, Ravel, French)]
- Some eager historian changes the nationality of C.P.E. Bach to „Prussian“, i.e.
  - A'_2 = [(Carl Philipp Emanuel, Bach, Prussian), (Johann Sebastian, Bach, German), (Maurice, Ravel, French)]
- Now, knowledge of A_1 and A'_2 to apply $\overleftarrow{R}$: $m_1 \times m_2 \rightarrow m_1$ is not enough!
  - Because we cannot identify the right Bach in A_1 (we cannot infer the year of birth from the first name without further knowledge)
  - *How can we solve this issue?*



3

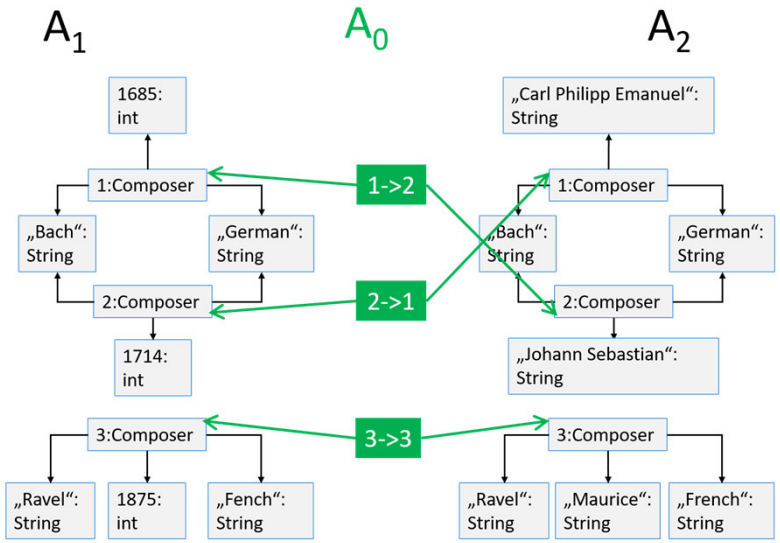
We start now with some typical situations that you encounter when working with data in ordinary enterprise applications. To keep things simple, we adhere to the composer example from Part 1.
Now, there are two Bach‘s in both composer collections.

(Actually, there are even more famous composers in the Bach Family. E.g. Carl Philipp Emanual Bach was known as the "Berlin Bach" during his residence in that city, and later as the "Hamburg Bach" when he worked as Kapellmeister there. He was called the Berlin Bach, because he obtained an appointment in the service of Crown Prince Frederick of Prussia, the future Frederick the Great. That’s why the eager historian changes Carl Philipp’s nationality - note that In the 18th century there was no German Country as we know it now, there were only small states, one of them was Prussia).

After this change, consistency has to be restored, but this poses the question which Bach in A_1 must be updated.

Correspondences

- Solution: *We must keep track of correspondences*

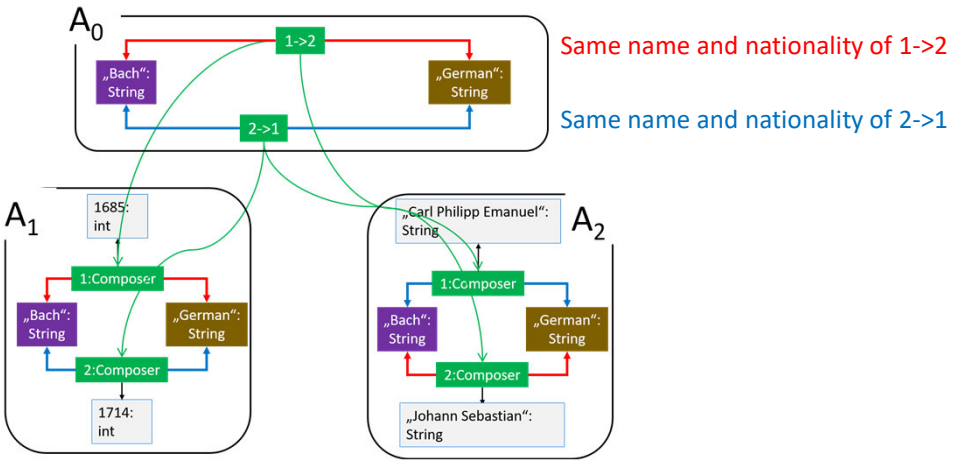


4

We need means to keep track of correspondences. Correspondences are entities in their own right (the green entities above) which relate two objects, one from A_1 the other from A_2, specifying their **sameness**.

Model Relations

- But there are more correspondences, see below!



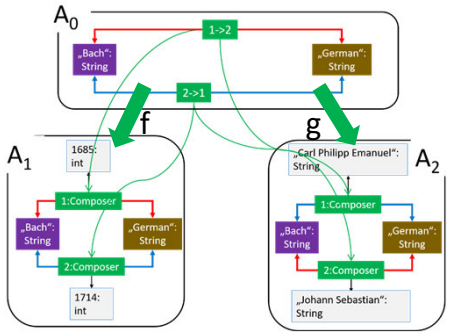
→ In such a way, we could also identify „same“ persons although there may be typos in the names

5

Of course, we should not identify objects only, but also sameness of features of the objects, e.g. the red arrows in A_0 specify sameness of the name and nationality features of the respectively identified composers. The remark below is important, because in such a way we can also identify sameness of entities although the respective values of a common feature may differ. Unfortunately this situation happens often in companies in which customers or other types of business partners can be recorded in different locations. Whereas one employee records Carl Philipp Emanuel Bach as the new Kapellmeister of the Prussian orchestra (in the 18th century cloud based IT system…), in another database by another employee the same composer is recorded as Carl Phillipp Emanuel Bach (note the two l's in the first name), making it difficult to identify these two later.
In fact we encounter this situation in insurance companies and banks.

Model Relation

- A relation on model elements can be specified by a pair of model mappings, if underlying metamodels coincide
- Elements of A_0 are called *representations* of pairs in the relation
  - Any $x \in A_0$ represents the pair *(f(x), g(x))*
- A pair $f: A_0 \rightarrow A_1$, $g: A_0 \rightarrow A_2$ of mapping is called a *Model Relation* between models A_1 and A_2.

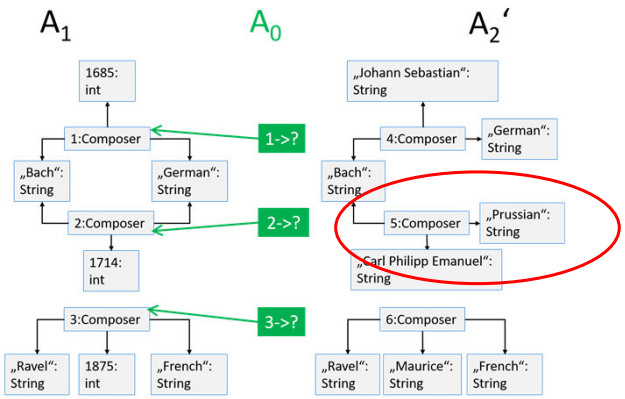


6

If we consider elements of A_0 as pairs (x_1,x_2) which define x_1 from A_1 and x_2 from A_2 **to be the same,** they possess projections $(x_1,x_2) \mapsto x_1$ and $(x_1,x_2) \mapsto x_2$ making up mappings $f: A_0 \rightarrow A_1$ and $g: A_0 \rightarrow A_2$ which map objects and references (links) between objects as shown here. After that these pairs can be renamed arbitrarily, the sameness specification remains valid.
A model relation is – more or less – a binary relation and can be seen as a linking table whose lines relate those elements that are referenced by foreign keys.
Formally, we obtain two *graph homomorphisms f* and *g* .
In our case the pair *(f,g)* is called a *model relation*. With model relations there are a lot of possibilities to relate different structures, e.g. if g is non-injective, different elements of A_1 (e.g. “Apple” and “Pear”) can be defined to be corresponding to a single element in A_2 (e.g. “Fruit”). Hence, with this technique, you can also specify *abstractions*. But we will not utilize this particular effect here.

Realignment

- Let‘s find a concept how to deal with the problematic change of nationality from “German” to “Prussian”



Note that keys can change or may not exist at all (e.g. in XML, JSON, …),
I.e. key 2 for Johann Sebastian (in A_2) is now key 4 (in A_2´)

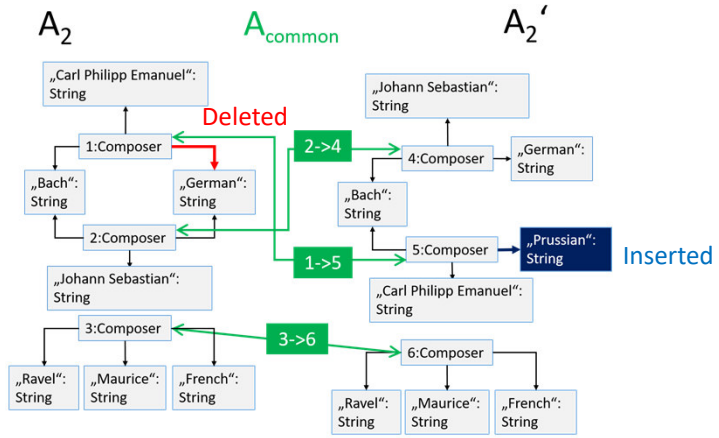
7

The change from A_2 to A'_2 may result in stored data, in which primary keys have changed. Think e.g. of the runtime identification of objects before and after a stop of a program or even XML documents with unclear identifications. Keys also change during copying from one database to another.

The necessary rearrangement (discussed on the next slides) of the commonalities in the model relation is called *realignment* (after an update of one side).

Realignment

- For this we must concretely specify the update $A_2 \rightarrow A'_2$



Span of graph morphisms $A_2 \leftarrow A_{common} \rightarrow A'_2$

A_{common}: Preserved during update

$A_2 - A_{common}$: Deleted

$A'_2 - A_{common}$: Inserted

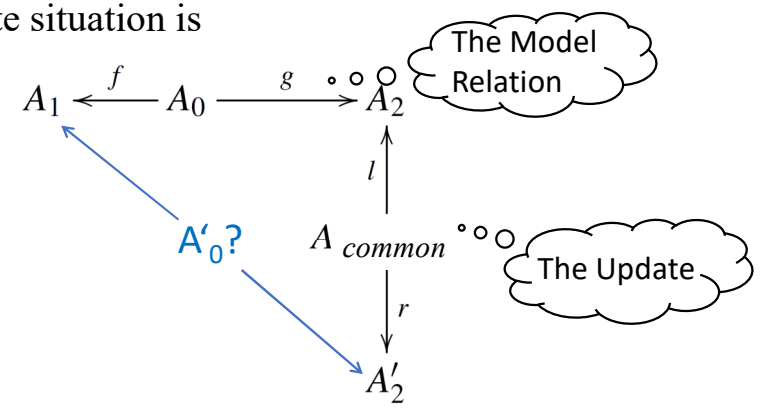
8

Updates are formalized in the same way as in graph transformation: Given a graph transformation rule, its application yields a span G <- D -> H of graph morphisms specifying the old state (G), the new state (H), and the remaining elements (D). Hence G – D are deleted parts, H – D are added elements.

In the figure, only the some (but the important) parts of A_0 are shown in green.

Realignment

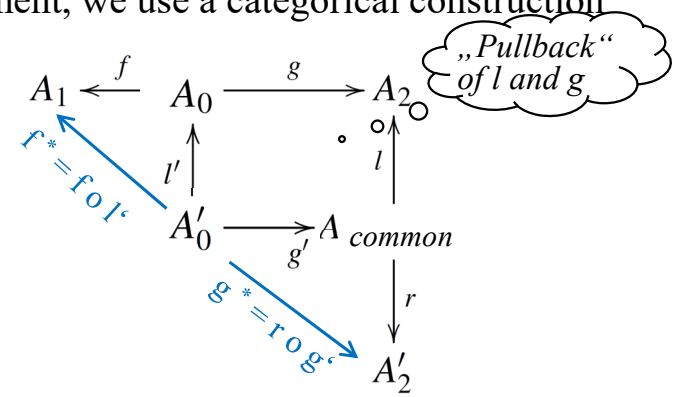
- The complete situation is



Realignment:
Find two mappings $A_1 \leftarrow A'_0 \rightarrow A'_2$, which identify the right Bach in A_1 with C.P.E. Bach in A'_2, hence the prerequisite for consistency restoration after the „Prussian change“.

Realignment

- For realignment, we use a categorical construction



$A'_0 = \{(repr,common) \in A_0 \times A_{common} \mid g(repr) = l(common)\}$

→ A'_0 contains exactly the elements that represent samenesses of A_1 and A_2 and whose A_2-elements remain in A'_2

10

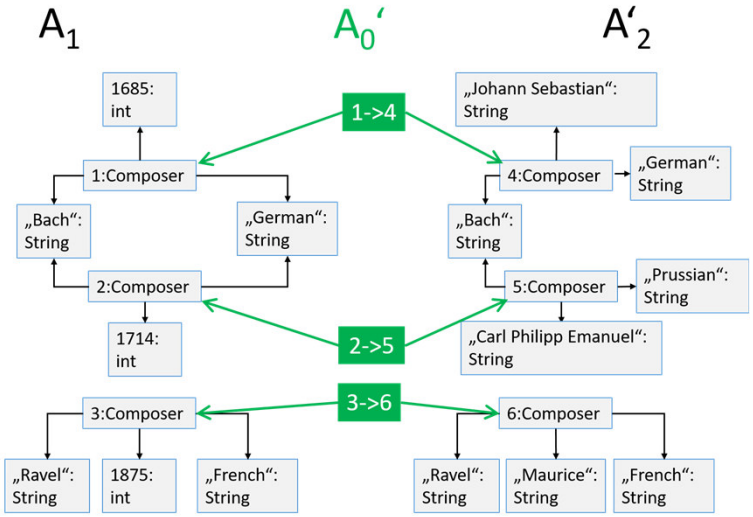
We use a construction from category theory which we explain here only descriptive (i.e. vividly).
One defines A'_0 to consist of those pairs (repr,common) $\in A_0 \times A_{common}$ for which $g(repr) = l(common)$, i.e. exactly the elements that have a partner in A_1 (because they appear as an image of g) and are preserved during the update (because they appear in A_{common}). These pairs admit projections l‘ and g‘ which composed with f and r make up the new model relation.

Of course it might be necessary to manually add new commonalities, e.g. if in the update a composer is added which was already contained in A_1. E.g. someone adds Edvard Grieg to A_2 and there is a Grieg in A_1 with year of birth 1843. A computer cannot decide whether he is really the same, such that a user (with adequate knowledge about the vita of Grieg) will have to decide.

Such amendments can be carried out fully automatically only, if there are strong identification properties available.

Realignment

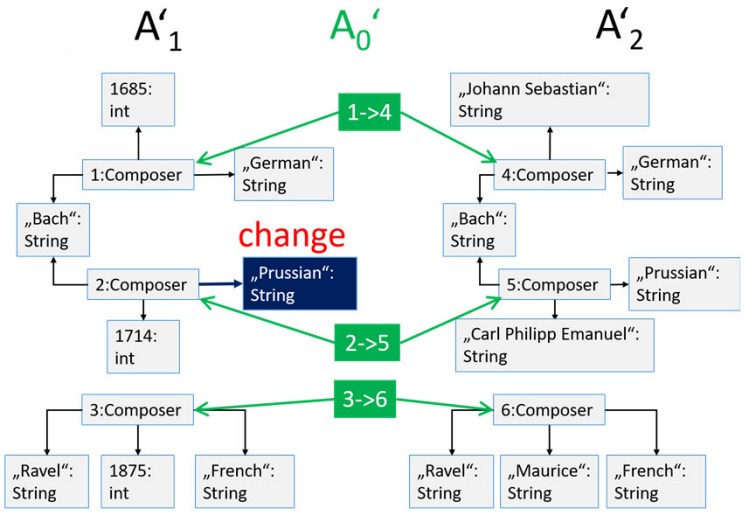
- The realignment result:



This is the realignment result according to the construction before. Except for the possible manual step mentioned before it can be carried out automatically, if model relation and update are specified accordingly.

Consistency Restoration

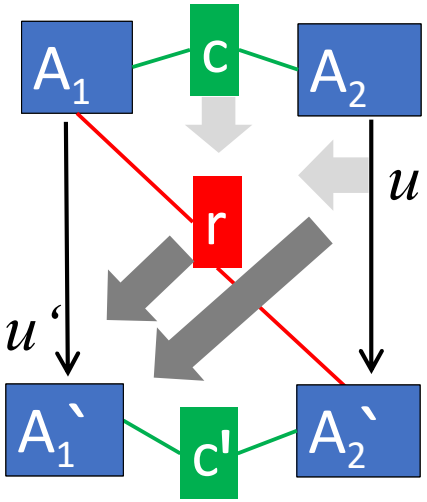
- The final result
  - (A general algorithm for the actual restoration is still to come)



The overall inconsistency of (A_1,A'_2) can then be repaired by changing the nationality of the composer in A_1, which is related to Carl Philipp Emanuel Bach in A_2 via the relation element 2->5. This makes up the *first part* of a general restoration algorithm, which is shown for the backward case on the next slide.

Backward Restoration in Symmetric Lenses

Input: Consistent Model Relation c and update u = (l,r)
Output: Update u‘: A_1 → A'_1 such that A'_1 and A'_2 are consistent (c‘)



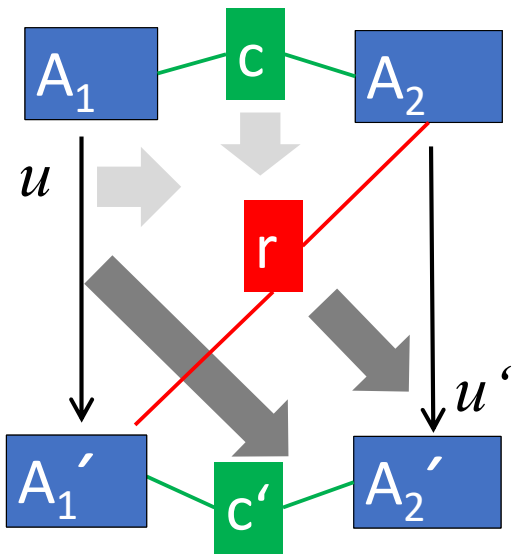
(c‘,u‘)=R(c,u)

```
r = Realign(c,u);
if(!r.isConsistent())
        return doRestore(r,u);
else
        return(r, id_A1)
```

Why is *u* input for doRestore?
Because in large models it helps to locate (smaller) areas, where A_1 has to be analyzed

u should be input for doRestore, because in large models we want to identify the update location (for this we need u) in order to isolate the area in A_2, which is affected by the change and hence have the option to isolate areas in A_1 (via c) where restoration takes place.

Forward Restoration in Symmetric Lenses



$(c',u') = \vec{R}(c,u)$

```
r = Realign(c,u);
if(!r.isConsistent())
        return doRestore(r,u);
else
        return(r, id_A2)
```

Currently we know, how to implement realign, but we do not know, how to implement

1. isConsistent and
2. doRestore

Summary up to now

- We specified a way to implement a (forward/backward) restoration, but only one step is clear:
  - Realignment ✓
  - Consistency Check ✗
  - Actual Consistency Restoration (Model Repair) ✗
- Moreover, we did it still a bit informally, because we did not take into account that models can be typed over different metamodels, the later differing to a bigger extent than in the composer example

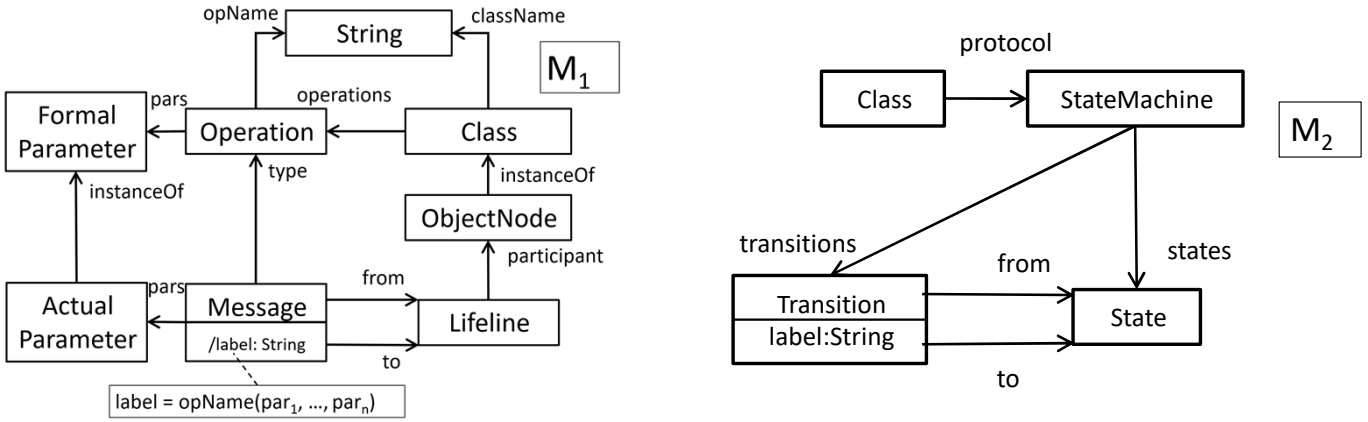
Next, let's see how the consistency check works, the challenge being the fact that constraints may spread over several different metamodels.

15

We turn to the consistency check and to the fixing of inconsistencies.

In the composer example, it was quite clear, that a *name* in M_1 means the same than in M_2. The same is true for nationalities. Moreover, on both sides composers were maintained. What happens if concepts of M_1 and M_2 differ to a bigger extent. The next page shows such an example and that it is not that easy to cope with constraints that concern two different metamodels.

Inter-model Constraints



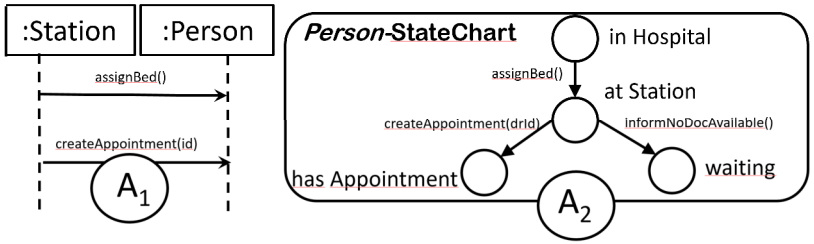
„The Sequence of messages targeted to a participant of type T must be a path in T's protocol State Machine"

16

On the left hand side there is the meta model (excerpt) for UML sequence diagrams. The label of a message is derived from the operation name and the formal parameters' names.
On the right hand side there is the metamodel for the assignment a state machine to a class, which restricts transitions sequences of its objects (this is also part of the UML Spec).
A typical constraint says that, in a runtime scenario, any sequence of messages that are all targeted to an object of class T must be allowed by the protocol state chart of T.

Satisfaction of Inter-Model constraints

[allowed] := „ The Sequence of messages targeted to a participant of type T must be a path in T's protocol State Machine"



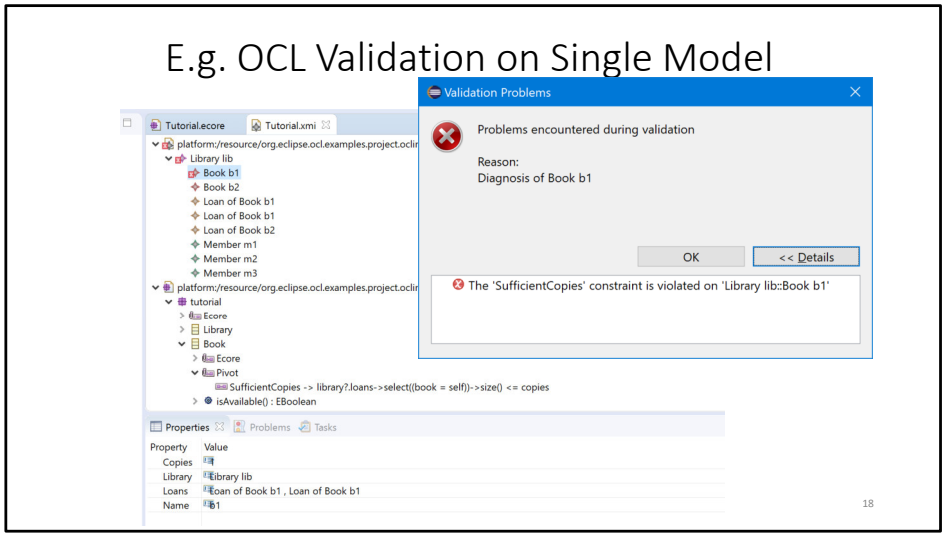
- How can we check and possibly repair $(A_1, A_2) \models$ [allowed] ?
- There are plenty of methods for checking constraints for $A \models e$
  - OCL Parser
  - DPF
  - Tools based on first order logic
- … and some for repair
- … but little methodology for *checking constraints that spread over several metamodels*

17

The blue constraint talks about ingredients of both metamodels and we need to check consistency w.r.t. the interplay of two models A_1 and A_2, the former being typed over M_1 the latter typed over M_2. I.e. formally one needs to investigate the question $(A_1, A_2) \models e$ and later the question of how to restore consistency, if the constraint is violated.

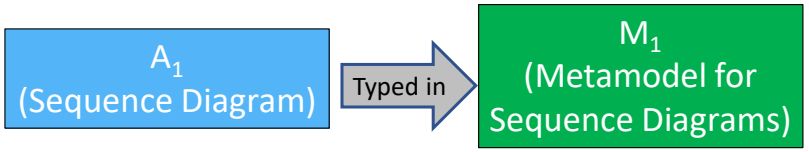
The idea will be to use known methods for checking and possibly repairing **a single** illegal model.

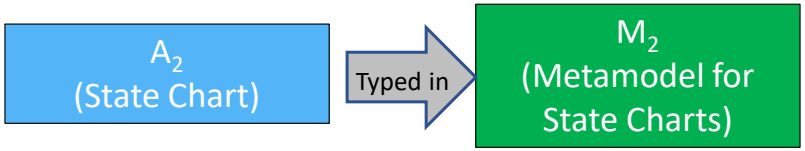
On the next slide we see EMF's built-in OCL constraint check



EMF allows to annotate OCL constraints, e.g. the “Sufficient Copies” constraint in the above screenshot. The goal is to reuse this or similar mechanism for our multi model scenario.

Reduction to Single-Model-Constraint-Check

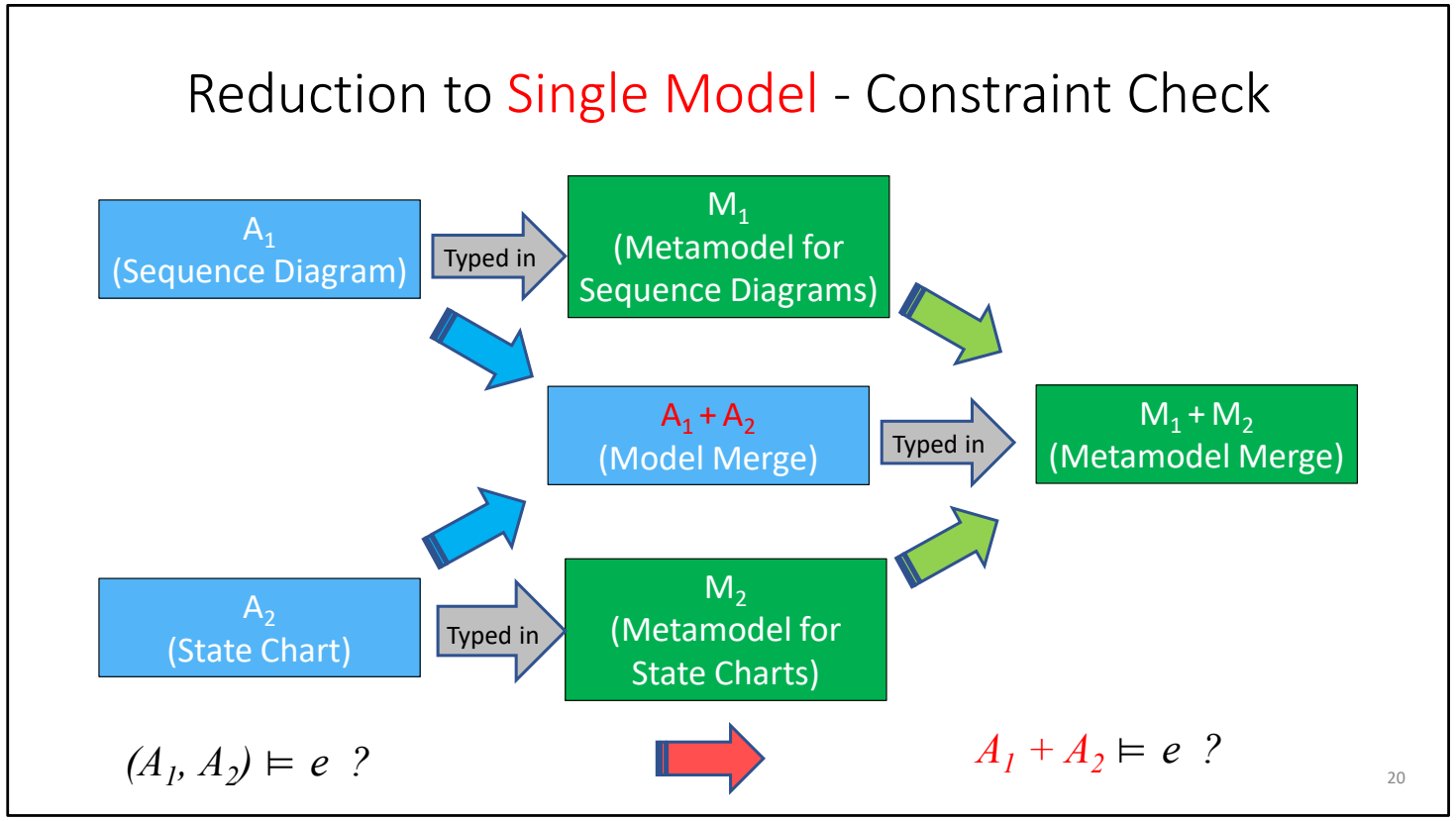




$(A_1, A_2) \models e \ ?$

19

Since constraints are defined in terms of concepts in M_1 and M_2, and we want to investigate constraint satisfaction of models A_1 and A_2, we must consider both the instance and the model level. In order to reduce to the single model scenario, …



… we will show how to merge metamodels as well as models in such a way that the latter is typed in the former. Then the single merged model $A^+:=A_1 + A_2$ can be checked to satisfy the constraints with well-known methods, which can be considered to be formulated over the merged metamodel $M^+:=M_1 + M_2$, in which A^+ is typed.

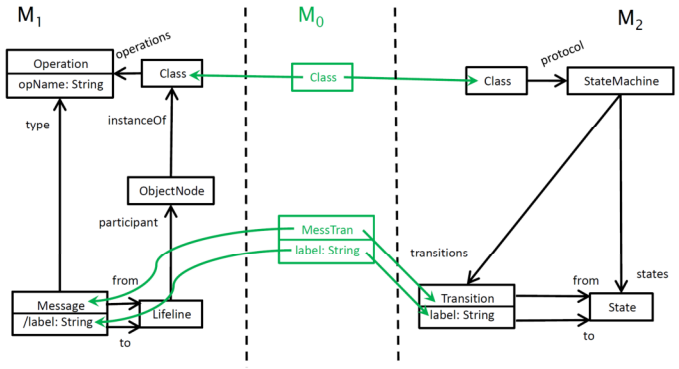
Thus the plan is to
(1) show, how **metamodels** and **model**s can be **merge**d and then
(2) explain methodology for **inconsistency fixing**.

1. Metamodel Merge

- What is needed to merge metamodels M_1 and M_2?

1. Metamodel Merge

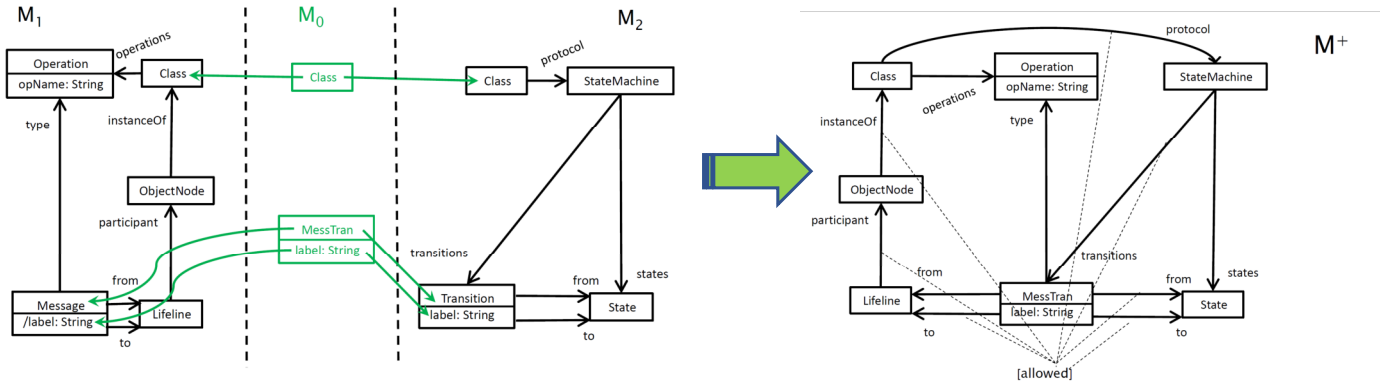
- What is needed to merge metamodels M_1 and M_2?
  - … specification of commonalities (M_0)
  - … such that the merge M^+ becomes a true **union**



If we want to make a statement about objects that are typed either in M_1 or in M_2, we must clearly specify commonalities. This was done in the composer example implicitly: If concepts like “nationality” were named identically, they were considered to be the same. The present example, however, show that this is not enough: The concept “Message” in M_1 must be defined to be the same as Transition in M_2, because otherwise we cannot transfer the term “sequence of message” to the term “path in a state chart”, which is needed to formally defined the inter-model constraint. Additionally the labels are defined to be equal, because in both cases they contain the name of the message/transition., e.g. “createAppointment(drId)”.

1. Metamodel Merge

- What is needed to merge metamodels M_1 and M_2?
  - … specification of commonalities (M_0)
  - … such that the merge M^+ becomes a true **union**
- → Thus constraints can formally be annotated on M^+



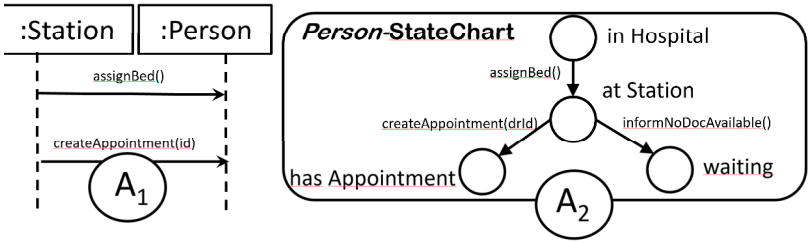
Once commonalties are settled, a merged metamodel M^+ can be calculated as the union of M_1 and M_2. Formally one uses a categorical “pushout” to do that. It is the same construction as in the derivation of graphs from graph transformation rules.

Then the constraint called [allowed] can be annotated on M^+ as is shown in the right figure (its scope is highlighted by dashed lines).

2. Model Merge

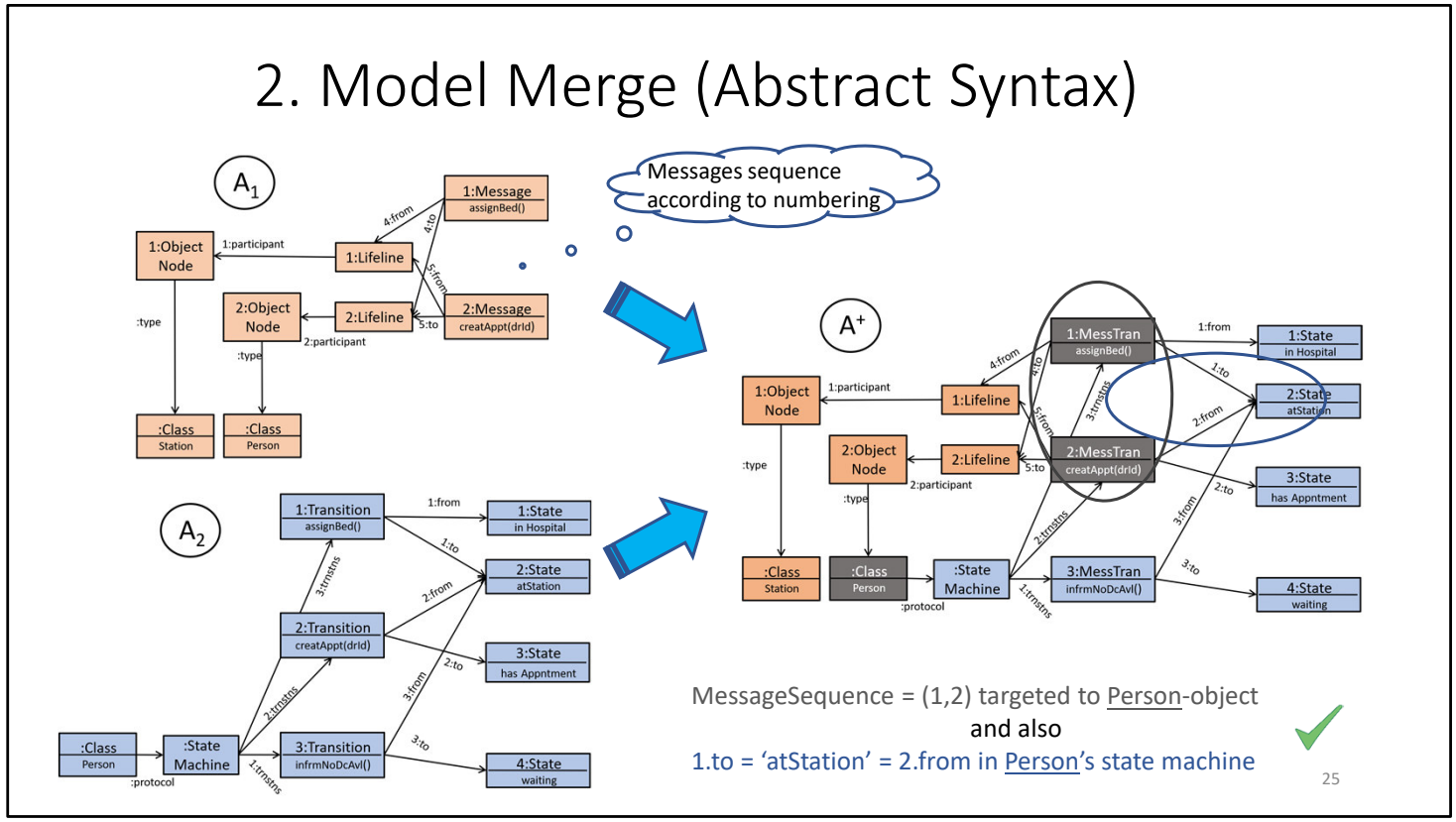
- What is needed to merge models A_1 and A_2?
  - … specification of commonalities (A_0)
  - … such that the merge A^+ becomes a true **union**
  - **… and can be typed over M^+**

Representation $x \in A_0$	$f'(x) \in A_1$	$g'(x) \in A_2$
1:Class	Person:Class	Person:Class
2:MessTran, label = „assignBed()“	:Message, label = „assignBed()“	:Transition, label = „assignBed()“
3:MessTran, label=„createAppointment(drId)“	:Message, label=„createAppointment(id)“	:Transition, label=„createAppointment(drId)“



24

In the same way we must merge models by – again – specifying commonalities and calculating the union, of which it can be shown that it is uniquely typable over M^+.



Because we need a common language for this procedure, the merge has to be carried out in abstract syntax.

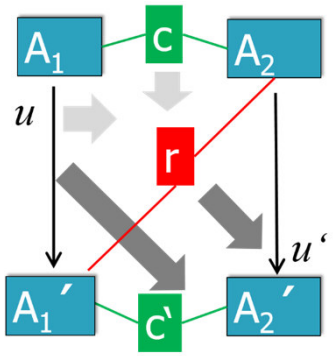
It is now possible to check validity of the constraint with known methods, e.g. OCL. In the example the sequence of messages targeted to the same object resulting from the numbering in the sequence diagram must be reflected by coinciding target and source states of subsequent transitions.

Checking *inter*-model constraints

- Thus inter-model constraints can be checked with techniques known for single models, but now on the merged instance

$$\tau: A^+ \rightarrow M^+$$

- Definition:
  - $(A'_1, A_2) \vDash E : \Leftrightarrow A^+ \vDash E$
  - Where A^+ is the merge of A'_1 and A_2
- Summary:
  - realign
  - isConsistent
  - doRestore



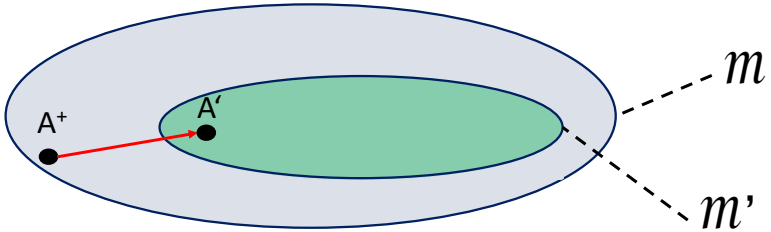

```
r = Realign(c,u);
if(!r.isConsistent())
        return doRestore(r,u);
else
        return(r, id_A2)
```

26

We see that working on a single model realizes the constraint check. And this will also be the case for the final step: Restoration in case of consistency violation. For this we, finally, investigate the field of “Model Repair”.

Model Repair

- The general situation:
  - Given a model space $\mathcal{M}$ and $A \in \mathcal{M}$ ($A := A^+$ in our scenario)
  - Let $\mathcal{M}'$ be a subspace of $\mathcal{M}$
  - Determine a *best* approximation A' of A *(in* $\mathcal{M}'$*)* and propose this as the solution of a restoration
- In our case
  - $\mathcal{M} = TypedGraphs^{?}(M^+,E)$
  - $\mathcal{M}' = TypedGraphs^{!}(M^+,E)$
- Two major Approaches
  - Rule based
  - Search based



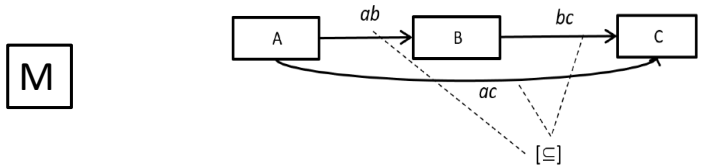
27

The general setting of model repair is an approximation scenario: We try to find a best approximation in a smaller space of some object in a large space. This is an old problem of mathematics, e.g. when approximating differentiable functions by a polynom (Taylor approximation) or data values by a straight line in linear regression methods.

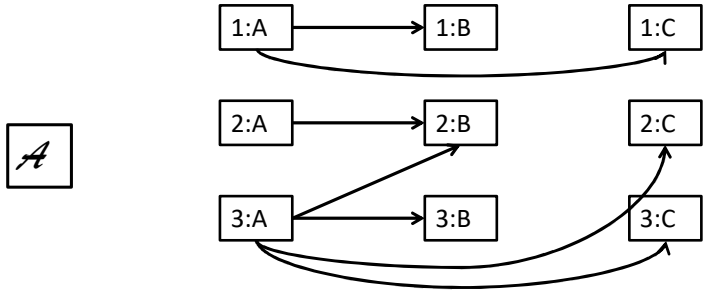
There are two strategies:

1) Rule Based, e.g. there is the rule to approximate a differentiable function by a polynom of degree n by calculating the derivatives up to the n^{th} one.
2) Search Based: An approximating solution is found by calculating the distance to possible candidates. In linear regression, one uses the principle of least squares.

Rule based Model (Graph) Repair



e = *(for all $a \in A$: $a.ac \subseteq a.ab.bc$)*



$\mathcal{A} \not\models e$

How can we control repair of the constraint violation?

28

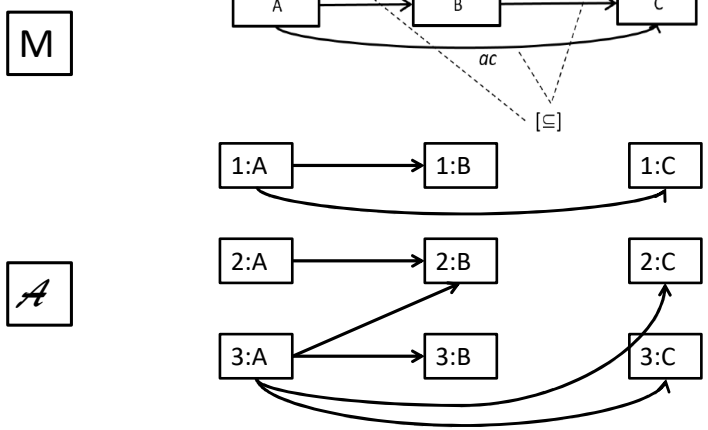
Above all multiplicities are 0..* and we write *a.ab.bc* for the union of all C-Objects referenced via *bc* from B-objects in *a.ab.*

The constraint claims that the set of C-Objects related to A-Objects via the ac-relation must be a subset of. *a.ab.bc*. E.g. in a theater hall *a : A* the set of occupied seats (*a.ac*) must be a subset of all seats *b.bc* in a row *b,* which belong to the hall (*b* ∈ *a.ab*)

This constraint is violated for a ∈ {1:A, 3:A}, because the C-Objects in a.ac have no “B-bridge”.

What are appropriate rules to repair the violation?

Rule based Model (Graph) Repair



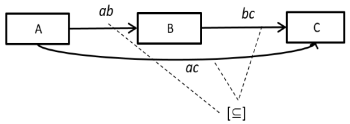
e = (for all a ∈ A: a.ac ⊆ a.ab.bc)

- A possible rule:
  - Given *a:A* with *a.ac* ⊈ *a.ab.ac*
  - For each *c* ∈ *a.ac* − *a.ab.ac* bridge with new *b*: *b* ∈ *a.ab* and *b.bc* = *c*

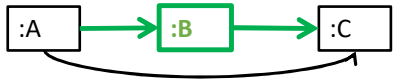
29

Let's see the effect of applying this rule.

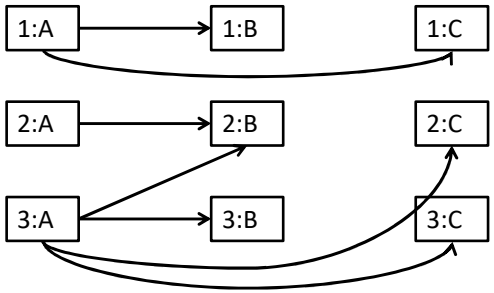
Graph Transformation Rule and Application



If you detect the black pattern in $\mathcal{A}$ and the green pattern is not present, add it at the detected position.

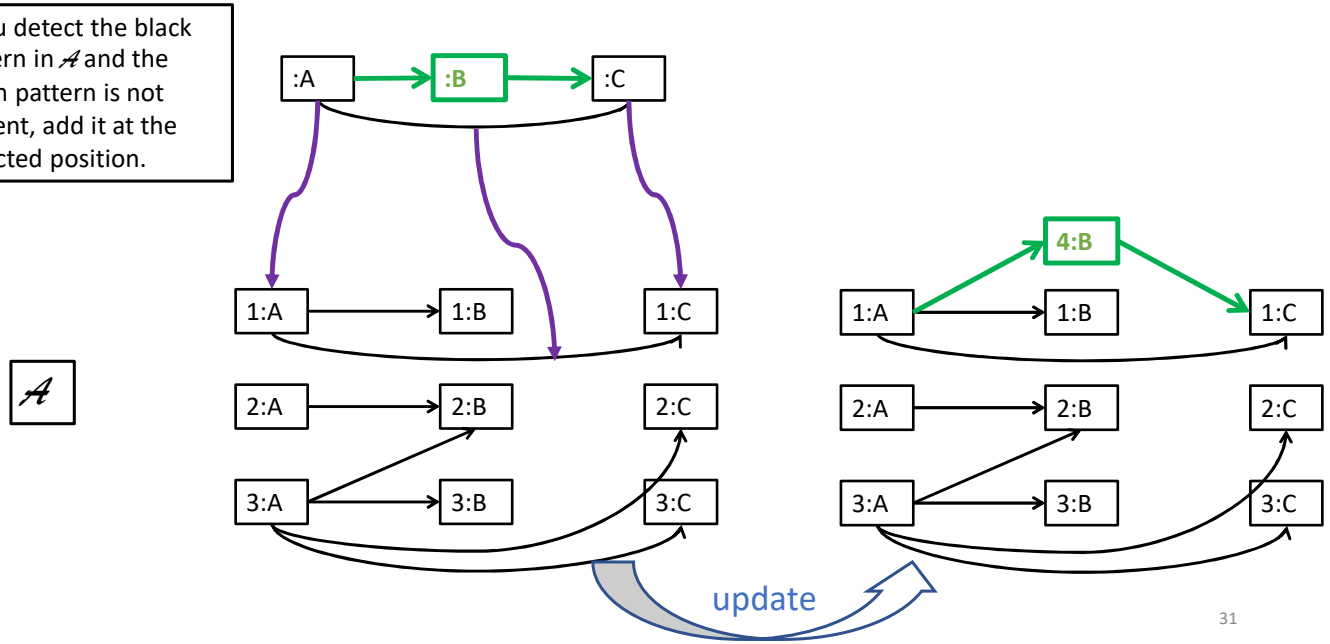


$\mathcal{A}$

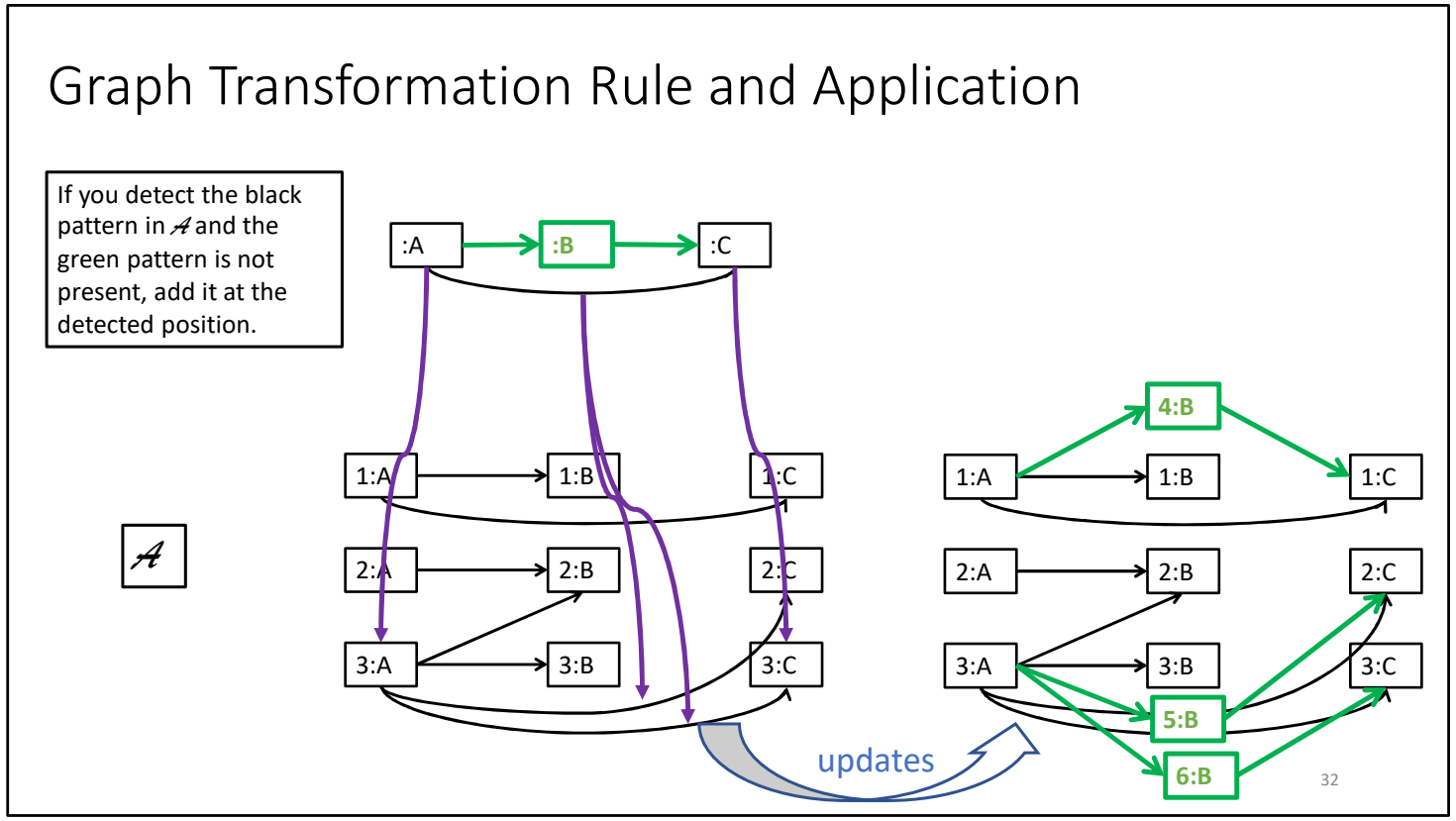


Graph Transformation Rule and Application

If you detect the black pattern in $\mathcal{A}$ and the green pattern is not present, add it at the detected position.



$\mathcal{A}$



The pictures shows two (concurrent) applications of the same rule. Now the instance is consistent, but there are several questions, see next Page.

Rule Based Approaches - Drawbacks

- The shown rule was only one example in a simple structure
- The third application could have reused *5:B* as bridge
- Other rules:
  - For each $c \in a.ac - a.ab.ac$ use bridge *b,* i.e. add $b \in a.ab$ and/or $b.bc = c$
  - What about deletion of references to c or deletion of a:A?
  - Are there other proposals?
- If several rules are applicable, which one to use first?
- In more complex structures, maintenance of a huge set of rules
  - Must be adapted / enhanced with each change of the metamodel (structure and constraints)
- Performance of *Pattern Matching*: Exponential complexity in general graphs. Complexity relaxed only in certain substructures
- → In many cases the rule based approach is not feasible!

33

There are several possible rules:

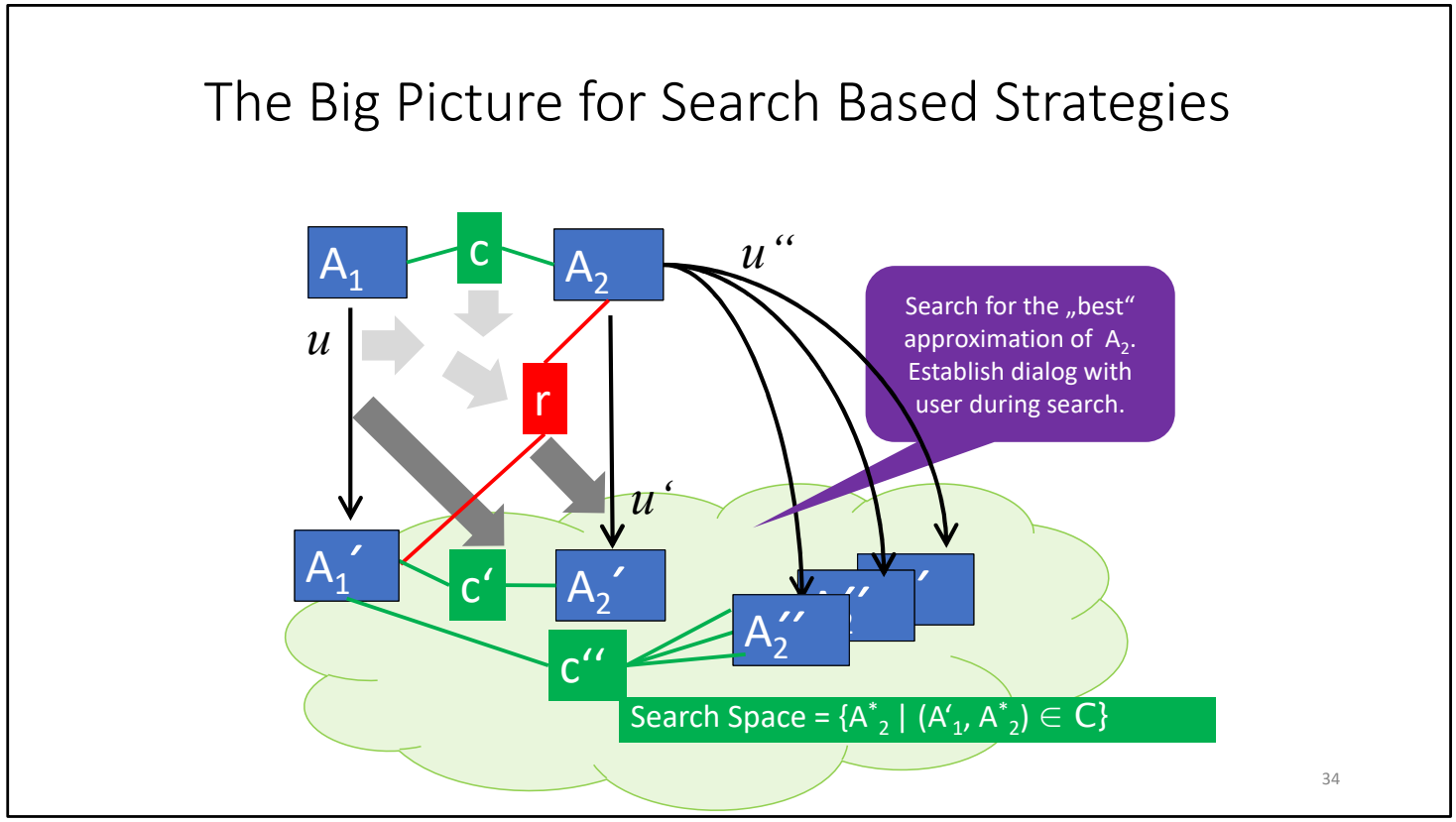
1) Bridge the gap by adding new B-objects
2) Use existing B-Objects as missing bridges
3) Delete the surplus references, i.e. remove 1:C from a.ac
4) Remove the violating A-Objects completely (?)
5) Other proposals may be to reuse only specific B-Objects as bridges and maybe to use only one new B-Object for all bridging

But all these solutions are worth to be discussed whether they are really in the scope of the users of the system. Moreover, it is important to know in which sequence different rules shall be applied. Do they interfere, are they causally dependent? Can they be carried out concurrently?

Graph transformation systems can deal with all these issues, but the technique has still not found its way into practice, because there is a lot of preparatory work to do.

priori no inconsistency is produced when instantiating new structures simultaneously. If deviations occur there are certain undo and redo strategies to find repaired models on the basis of the rules. These approaches are sometimes used in industry solutions.

In either case, precision of rule based approaches does not dominate difficult handling, i.e. rule based approaches have only rarely found their way into practice. So let’s turn to **search based approaches**.



Given r (after realignment) and update u:A_1->A'_1. One searches among all A^*_2 which are consistent with A'_1 and which have the least distance to A_2.

Search Based Approaches

- Let again $\mathcal{M}$ be a model space and $A \in \mathcal{M}$, let $\mathcal{M}'$ be a subspace of $\mathcal{M}$, in which we now *search* for a *best* approximation A' of A
- General requirements when determining A'
  - *Correctness and hippocraticness*
    - $A' \in \mathcal{M}'$
    - $A' := A$ if already $A \in \mathcal{M}'$
  - *Least-Change*: Among all possible restorations the extent of changes $A \rightarrow A'$ should be minimal
  - *Least-Surprise*: Changes should not surprise the modelers too much
  - *Performance:* A' must be determined in adequate time
- Trade-Offs
  - Least-Surprise vs. Least-Change requires interaction with the user
- To find algorithms that fulfill the requirements, we have to measure the extent of changes

35

The setting and requirements are the same as before. But now we need means to measure “vicinity”. It is a s in the example of linear regression where the measurement was the least-square method. The least change principle has to be combined with the principle of least surprise. In order to find a good trade-off, user interaction is necessary. But first we shortly look at methods to measure the extent of changes.

Change Measurement

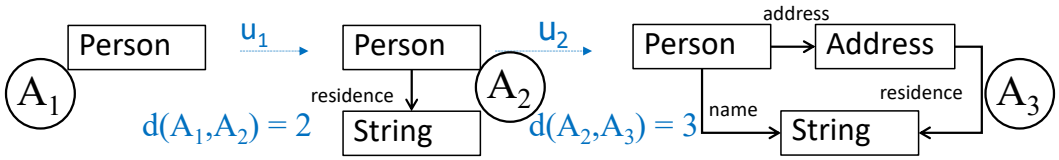
- Metric spaces: Let X be any set and $\mathbb{R}$ be the set of real numbers, then …
- $d: X \times X \rightarrow \mathbb{R}$ is called a **Metric**, if
  1) $d(x,y) = 0 \Leftrightarrow x = y$ (Different elements have positive distance)
  2) $d(x,y) = d(y,x)$ (Symmetry)
  3) $d(x,z) \leq d(x,y) + d(y,z)$ (Triangle inequality: „A detour via y is always longer“)
- A metric always yields positive distances (or 0):
  - $0 = d(x,x) \leq d(x,y) + d(y,x) = 2*d(x,y) \rightarrow d(x,y) \geq 0$

36

For measuring distances, metrics are used. They are an old methodology in mathematics. The method of least squares yields a metric. In order to be a reasonable way of measuring distances the three axioms above are essential. The fact that $d(x,y) \geq 0$ for all x, y follows automatically.

Least Change

- Model spaces are metric spaces
  - Atomic changes yield distance 1 between old and new
- General definition: For *u: A → B* let
  - *n(u)* = the number of atomic changes in *u*
  - *d(A,B) = min{n(u) | u:A → B}*



A possible def. for atomic changes: add/remove one node/edge, extract/inline class

37

A typical metric for model management is shown above.
It can be shown that the three axioms for metrics are satisfied.

Least Change (Ambiguity of Nearest Solutions)

- Another simple example:
  - Let *(X,Y)* be two sets of natural numbers and *Z* another set
  - *(X,Y)* and *Z* are consistent, if $X \cap Y = Z$
  - Consider $\bar{R}((X,Y), Z')$, e.g. $\bar{R}((\{1,2,3\},\{0,2,4\}),\{5\})$
- $d((X,Y), (X',Y')) := |symmDiff(X,X')| + |symmDiff(Y,Y')|$
  - *symmDiff(X,X')* = The number of elements either in *X* or in *X'*, but not both
  - $symmDiff(X,X') := (X - X') \cup (X' - X)$, *e.g.* $symmDiff(\{0,2,4\}, \{0,1\}) = \{2,4,1\}$
- One can show that *d* is reflexive, symmetric and transitive
- Let's find solutions for $\bar{R}((\{1,2,3\},\{0,2,4\}),\{5\})$ with minimal change:
  - ({1,3,5}, {0,4,5}), d = 2 + 2 = 4
  - ({1,3,5},{0,2,4,5}), d = 2 + 1 = 3
  - ({1,2,3,5},{0,4,5}), d = 1 + 2 = 3

38

The shown metric is very similar to the model distance on the previous page.

But there are not always unique solutions:

Possible solutions are

({1,3,5}, {0,4,5}), d = 2 + 2 = 4

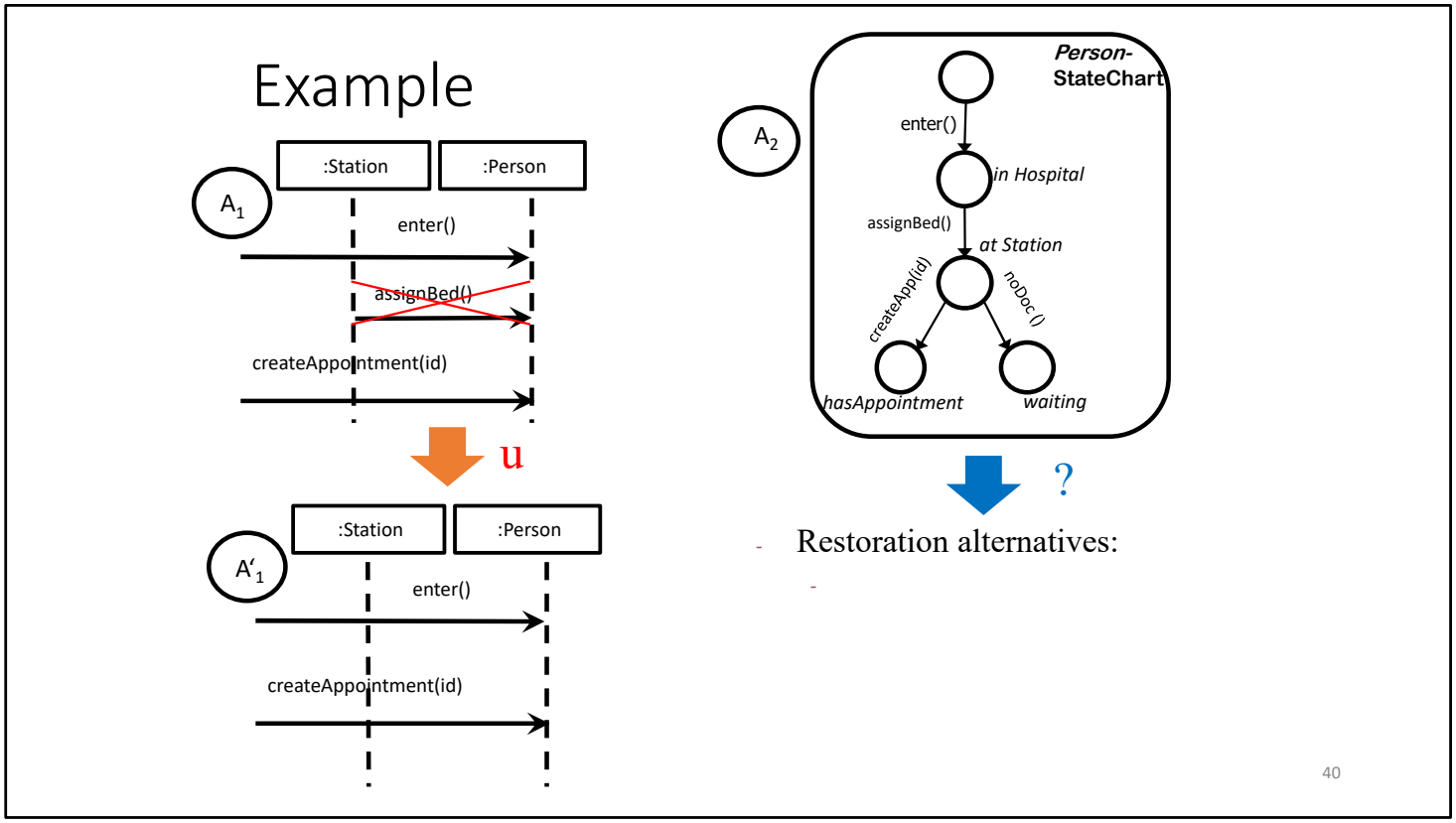
({1,3,5},{0,2,4,5}), d = 2 + 1 = 3

({1,2,3,5},{0,4,5}), d = 1 + 2 = 3

Least Change vs. Least Surprise

- Let $u: A \rightarrow A'$ ($A \in \mathcal{M}$, $A' \in \mathcal{M}'$, $A' \neq A$)
- *Search algorithm* in a metric space for a best repair of A': Let k be a small number (e.g. $k=3$)
  - 1. $d_0 := 0$, $d_{max} := k$
  - 2. Calculate and show all $A'' \in \mathcal{M}' - \{A\}$ with $d_0 \leq d(A', A'') \leq d_{max}$.
  - 3. If a user chooses some of these A'', finish. Else $d_0 := d_{max}$, $d_{max} := d_{max} + k$, back to 2.
- Trade-Off between …
  - … least change (by using metrics)
  - … least surprise (by user interaction)

39

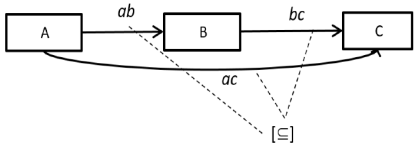


There are several alternatives for repair:

1) Add transition enter() also with target atStation (distance one, surprising?)
2) Remove state inHospital and redirect enter() to atStation and assignBed to start state (distance three), or let assignBed() become a loop atStation (very surprsing)
3) Merge InHospital and atStation (smaller distance, but possibly very surprising)

Search Implementation

- Search implementation in established tools:
- Let E be a set of constraints, e.g. formulated in OCL
  - Fix upper limit n for the number of nodes of each type
  - Encode associations as binary relations, e.g. $ab \subseteq A \times B$
    - E.g. $ab = \{(a_1,b_1), (a_1,b_2), (a_2,b_1)\}$ ➔ $a_1.ab = \{b_1, b_2\}$
  - Represent *all possibilities* for each relation $r \subseteq X \times Y$ by introducing a variable r_{ij} for each pair (x_i, y_j)
    - r_{ij} = true if and only if $(x_i, y_j) \in r$
    - E.g. ab_{ij} for $1 \leq i,j \leq n$
    - In the example $3n^2$ variables
- → An instance typed over a model M with constraints E is valid if and only if these variables satisfy a certain boolean formula
  - In the example, each instance with n objects of type A, B, C, resp., is valid if and only if the following formula is true: $ac_{ik} \Rightarrow \bigvee_{j=1}^{n} (ab_{ij} \wedge bc_{jk})$



for all $a \in A$: $a.ac \subseteq a.ab.bc$

41

ac_{ik} means that $(a_i,c_k) \in$ ac, i.e. a_i.ac contains c_k. The disjunction is true if and only if there is b_j such that $(a_i,b_j) \in$ ab and $(b_j,c_k) \in$ bc. Thus the implication is true if and only if the subset relation is true.

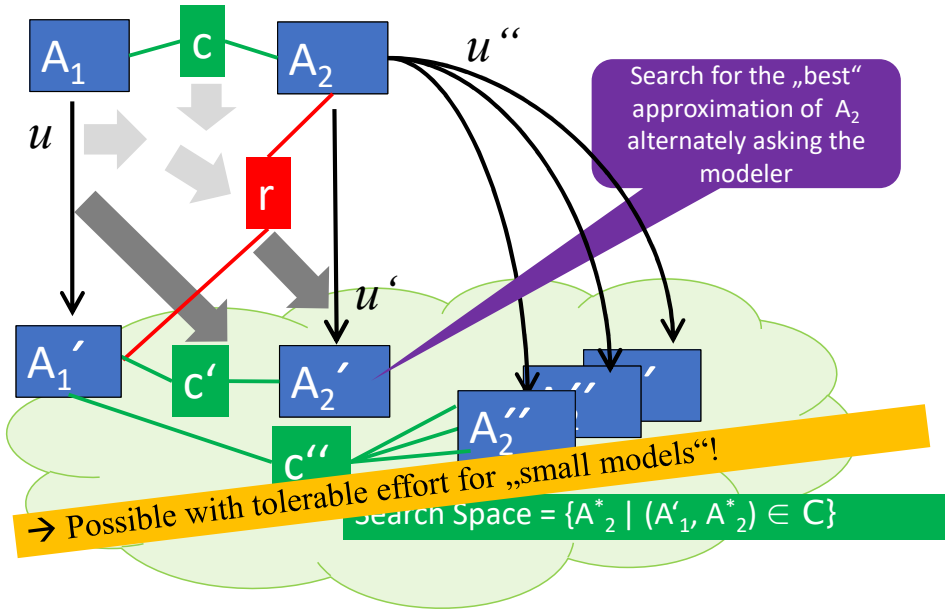
Performance

- Finding a valid instance A' near a given instance A
  - Determine upper and lower limits for the size of A'
  - Searching near A means to determine these limits according to the size of A (Least change)
  - Determine formulas, which fix certain values in the solution
    - E.g. $\vec{R}((A'_1, A_2)$ may specify all values in A'_1 to be unchangeable, i.e. $r_{ij} = true$ as part of the formula whenever (x_i,y_j) is a relation element in A'_1
- Finding valid instances by satisfying the formula(s) corresponding to the constraints
  - Again integrate user interaction when analyzing search proposals
- SAT-Solver are established tools for this task
  - The SATisfaction Problem: *Given a boolean formula. Find variable assignment such that the formula becomes true!*
- Unfortunately SAT is NP-complete
  - I.e. it exhibits exponential time complexity and …
  - … we can *never* expect to find a faster algorithm

42

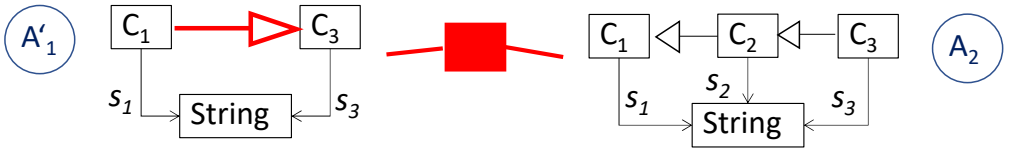
Tell the story of NP-completeness (the word *never* is relati.ve)

Conclusion: Consistency Restoration with Search Based Strategies



Exercise 1

- Consider the following two models, which have become inconsistent, after the red specialization has been added in A_1 (yielding A'_1), because cyclic inheritances are prohibited



- Discuss options how to restore consistency …
  - … by carefully the sets of attributes of all classes and …
  - … without surprising the modelers too much
  - … by changing as little as possible in the models
  - … without disturbing the modeler of A_1 on the one hand
  - … with the option of amending A'_1 on the other hand

44

You are free to propose solutions. It is, however, not that easy, right?

Exercise 2

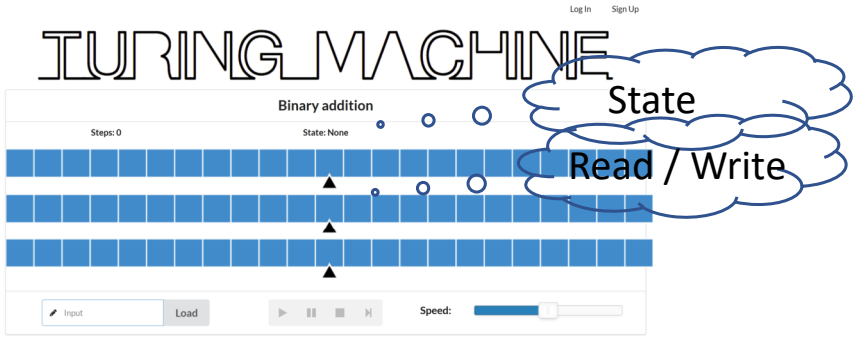
- Let A and B classes and *ab* an association from A to B. How do you express the multiplicities 1..* and 0..1 at *ab*'s association end at class B?
- Hint: You need an additional relation id_B, for which the variables $(id_B)_{ij}$ are constantly set to true, if $i = j$, and false otherwise
- Answer:

Reservoir: $\wedge$ $\vee$ $\Rightarrow$

Optional Exercise:
Multiplicity 0..1: Forall a_i:A. $ab_{ij} \wedge ab_{ik} \Rightarrow (id_B)_{jk}$
Multiplicity 1..*: Forall a_i:A. $ab_{i1} \vee ab_{i2} \vee \ldots \vee ab_{in} = \text{true}$

Appendix: NP-Completeness

- In order to quantify time complexity of algorithms universally, we need a technology independent „concept of calculation“
- In Computer Science the universal „program“ are Turing Machines
- For a pair *(s,a)* of current state *s* and read character the „program“ yields
  - … a triple *(s‘, b, M)* (deterministic TM)
  - … a set of such triples (non-deterministic TM)



Appendix : NP-Completeness

- Now we can measure time complexity of a TM M by comparing the length $|x|$ of the initial input x with the number $\#_M(x)$ of computation steps it takes until a final state is reached
- Let $\mathcal{O}^{\mathrm{D}}(f(n))$ / $\mathcal{O}^{\mathrm{N}}(f(n))$:= The set of all languages A, for which there is a deterministic / non-deterministic TM M such that
  - $A = L(M)$
  - For all x: $\#_M(x) \leq C(f(|x|))$ for some $C > 0$

$\mathcal{P}$:= $\{A \mid$ there is $k \in \mathbb{N}$ such that A is in $\mathcal{O}^{\mathrm{D}}(n^k)\}$

$\mathcal{NP}$:= $\{A \mid$ there is $k \in \mathbb{N}$ such that A is in $\mathcal{O}^{\mathrm{N}}(n^k)\}$

SAT $\in \mathcal{NP}$

- Formally we consider the language

  $SAT = \{F \mid F \text{ is a satisfiable formula}\}$

- Here we let |F| = number of used variables
  - In practical problems there is a proportional dependency between length of the formula and number of used variables, thus this is no real change
- Proof sketch: TM guesses values
  - Guessing = Non-Determinism

    $F = (!x_0 \vee x_1 \vee x_2) \wedge (x_0 \vee !x_1)$

- If all guesses are good, e.g. $x_0 = false,\ x_1 = false, x_2 = true$
  - → Success in linear time (evaluation also takes linear time)
- If not, e.g. $x_0 = false \rightarrow x_1 = true \rightarrow x_2 = true$
  - No success in this case

→ SAT $\in \mathcal{NP}$

- It is also easy to see that SAT $\in \mathcal{O}^D(2^n)$
  - Because a deterministic machine *must be programmed* to try all possible value assignments for each variable
  - Adding a variable doubles the number of possibilities

$\mathcal{P} \neq \mathcal{NP}$?

- So we have two statements

$$\text{SAT} \in \mathcal{NP}$$

and

$$\text{SAT} \in O^D(2^n)$$

- The second one gives SAT *exponential complexity* based on the brute force algorithm of trying every assignment – this is the case for many other problems
- A rule of thumb is: $\mathcal{NP}$ = *bad performance* , $\mathcal{P}$ = *good performance*
- Of course $\mathcal{P} \subseteq \mathcal{NP}$, since each det. TM is a non-det. TM
- But it is an open problem whether

$$\mathcal{P} \neq \mathcal{NP}$$

- *I.e. no one has found a problem in* $\mathcal{NP}$ *that is provably not in* $\mathcal{P}$*!!*
- **This is considered to be the most important open problem in computer science**
  - The problem is considered important, because there are many algorithms which exhibit exponential complexity (are in $\mathcal{NP}$), e.g. *traveling salesman*, *coloring* problems, …
  - … where it is hence unclear whether there are faster algorithms

Moreover …

- There is the following astonishing fact

There is a class of problems with exponential complexity, called *NP-complete* problems, with the following property:

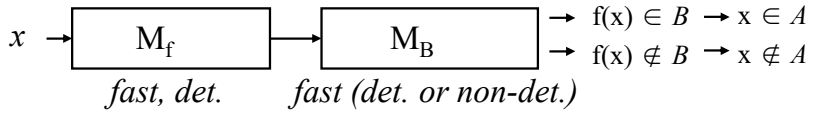
If a faster algorithm is found for **only one** NP-complete problem A (moving A to class $\mathcal{P}$), then

$$\mathcal{P} = \mathcal{NP}$$

- For the definition of NP-completeness, we use the following reduction methodology
- A is *reducible* to B, written $A \preceq B$, if there is a deterministically fast computable function f such that

$$x \in A \Leftrightarrow f(x) \in B$$

- This gives $(A \preceq B, B \in \mathcal{P} \Rightarrow A \in \mathcal{P})$ and $(A \preceq B, B \in \mathcal{NP} \Rightarrow A \in \mathcal{NP})$



NP-Completeness

- Last slide:
  - A is *reducible* to B, written $A \preceq B$, if there is a deterministically fast computable function f such that
    $$x \in A \Leftrightarrow f(x) \in B$$
  - This gives
    $$A \preceq B, B \in \mathcal{P} \Rightarrow A \in \mathcal{P} \qquad (*)$$
    $$A \preceq B, B \in \mathcal{NP} \Rightarrow A \in \mathcal{NP} \qquad (**)$$
- Definition: B is called **NP-complete**, if
  - $B \in \mathcal{NP}$ and
  - **For all** $L \in \mathcal{NP}$: $L \preceq B$
- *Theorem*: If there is NP-complete B and $B \in \mathcal{P}$, we obtain $\mathcal{P} = \mathcal{NP}$
  - Proof: Let $L \in \mathcal{NP}$. NP-completeness $\Rightarrow L \preceq B$. By *(*)* $L \in \mathcal{P}$. $L \in \mathcal{NP}$ was arbitrary!!

It remains to find an NP-complete problem:

- *SAT* is NP-complete.

Proof: Let $L \in \mathcal{NP}$ be arbitrary. There is non-det. TM M with $L = L(M)$. For each tape input of M one can algorithmically (fast and deterministically) compute a formula F such that

$$x \in L \Leftrightarrow F \in SAT$$

Variables express the state of M, e.g. $state_{t,s} = true$, if at computation time t M has state s, and certain implications simulate computation steps. The assignment $x \mapsto F := f(x)$ yields a reduction, i.e. $L \leq SAT$. Since L was arbitrary, *SAT* is NP-complete.

- *Corollary*: If someone finds a fast algorithm for SAT, then many, many other problems with slow algorithms possess a fast algorithm!
- *That makes it very unlikely, that SAT has a fast implementation!!*
- Remark: A similar reduction techniques reveals many other problems to be NP-complete, as well.
- *Corollary*: If someone finds a fast algorithm ***for any NP-complete problem***, then many, many other problems with slow algorithms possess a fast algorithm!