Bidirectional Transformations

Jeremy Gibbons

SSBX, Oxford, July 2016

1. Scenarios

Bidirectional transformations ('BX') maintain

> *different representations of shared data*.

They *restore consistency* when either copy changes.

For engineering reasons, prefer *one bidirectional* specification rather than *two unidirectional* ones.

(I'm only going to address the *binary* case.)

Data conversions

```
BEGIN:VCARD
VERSION:3.0
N:Gibbons;Jeremy;;;
FN:Jeremy Gibbons
ORG:University of Oxford;
EMAIL;type=INTERNET;type...
TEL;type=WORK;type=pref:...
TEL;type=CELL:07779 7972...
item1.ADR;type=WORK;type...
item1.X-ABADR:gb
PHOTO;BASE64:
  /9j/4AAQSkZJRgABAQAAAQ...
X-ABUID:6EEE2835-745D-4F...
END:VCARD
```

⇔

A *bijective relationship* is a special (and degenerate) case.

View–update in databases

Staff

Name	*Room*	*Salary*
Sam	314	£30k
Pat	159	£25k
Max	265	£25k

Projects

Code	*Person*	*Role*
Plum	Sam	Lead
Plum	Pat	Test
Pear	Pat	Lead

```
SELECT
  Name, Room, Role
FROM
  Staff, Projects
WHERE
  Name=Person
AND
  Project="Plum"
```

$\Longrightarrow$

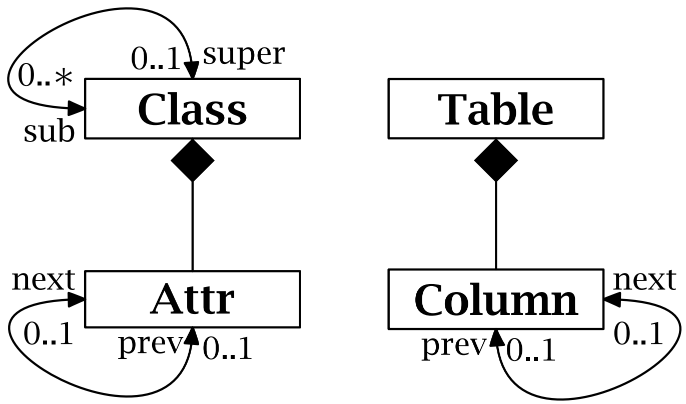
View

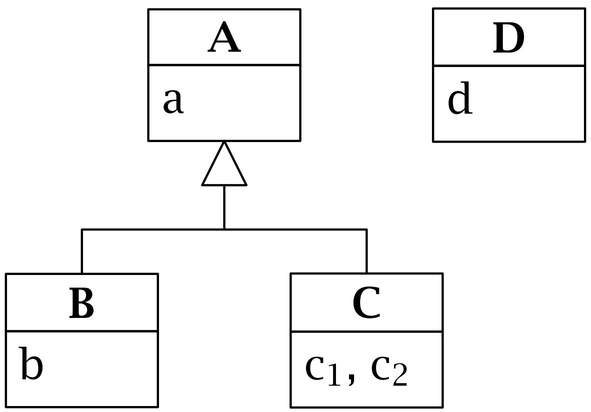
Name	*Room*	*Role*
Sam	314	Lead
Pat	159	Test

MDD

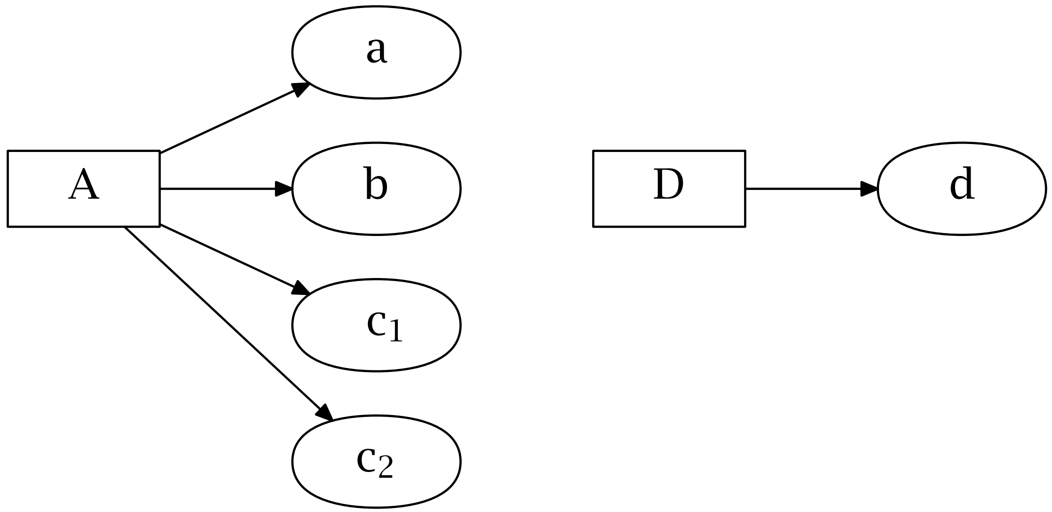
Object-relational mapping:

- classes, single inheritance, ordered attributes
- tables, ordered columns
- one table per hierarchy





⇔



Composers

State spaces

$$M = \{ Name \times Dates \times Nationality \}$$
$$N = [Name \times Nationality]$$

where $m : M$ is consistent with $n : N$ if they have the same *set* of $Name \times Nationality$ pairs:

$$
\begin{aligned}
m \;=\; \{ & (\text{“Jean Sibelius”}, && 1865\text{–}1957, && \text{Finnish}), \\
& (\text{“Aaron Copland”}, && 1910\text{–}1990, && \text{American}), \\
& (\text{“Benjamin Britten”}, && 1913\text{–}1976, && \text{English}) \}
\end{aligned}
$$

$$
\begin{aligned}
n \;=\; [& (\text{“Benjamin Britten”}, && \text{English}), \\
& (\text{“Aaron Copland”}, && \text{American}), \\
& (\text{“Jean Sibelius”}, && \text{Finnish})]
\end{aligned}
$$

Various ways of restoring consistency: ordering, dates…

(BX repository, `http://bx-community.wikidot.com/examples:composers`)

2. Approaches

A bestiary for the week's fauna:

relational: see eg Stevens'

- *"Equivalences Induced on Model Sets by BX"* (BX 2012)
- *"Bidirectional Model Transformations in QVT"* (SoSyM 2010)

lenses: see eg

- Foster *et al.*'s *"Combinators for BX"* (POPL 2005)
- Hofmann *et al.*'s *"Symmetric Lenses"* (POPL 2011)

ordered, delta-based, categorical: see eg

- Hegner's *"An Order-Based Theory of Updates"* (AMAI 2003)
- Diskin *et al.*'s *"From State- to Delta-Based BX"* (JOT 2011)
- Johnson *et al.*'s *"Lens Put-Put Laws"* (BX 2012)

triple-graph grammars: see eg

- Schürr's *"Specification of Graph Translators with TGGs"* (WG 1994)
- Anjorin *et al.*'s *"20 Years of TGGs"* (GCM 2015)

Relational

A *BX* $(R, \overrightarrow{R}, \overleftarrow{R}) : M \nLeftrightarrow N$ between model spaces (sets) M, N consists of

- a *consistency relation* $R \subseteq M \times N$
- a *forwards consistency restorer* $\overrightarrow{R} : M \times N \to N$
- a *backwards consistency restorer* $\overleftarrow{R} : M \times N \to M$

The idea is that given inconsistent models m', n, forwards consistency restoration yields $n' = \overrightarrow{R}(m', n)$ such that $R(m', n')$ holds. And vice versa.

The BX is *correct* if consistency is indeed restored:

$$\forall m', n.\ R(m', \overrightarrow{R}(m', n)) \qquad \forall m, n'.\ R(\overleftarrow{R}(m, n'), n')$$

and *hippocratic* if restoration does nothing for consistent models:

$$\forall m, n.\ R(m, n) \Rightarrow \overrightarrow{R}(m, n) = n \qquad \forall m, n.\ R(m, n) \Rightarrow \overleftarrow{R}(m, n) = m$$

and *history-ignorant* if

$$\forall m, m', n.\ \overrightarrow{R}(m', \overrightarrow{R}(m, n)) = \overrightarrow{R}(m', n)$$ -- and vice versa

Lenses

A *lens* $(get, put) : S \rightleftharpoons V$ from source S to view V consists of two functions

$$get : S \to V$$
$$put : S \times V \to S$$

The idea is that $get\ s$ projects a view from source s,
and $put\ s\ v'$ restores a modified view v' into existing source s.

The lens is *well-behaved* if it satisfies

$$\forall s, v.\ put\ (s, get\ s) \quad = s \qquad \text{(GetPut)}$$
$$\forall s, v.\ get\ (put\ (s, v)) = v \qquad \text{(PutGet)}$$

It is *very well-behaved* (rather strong) if in addition it satisfies

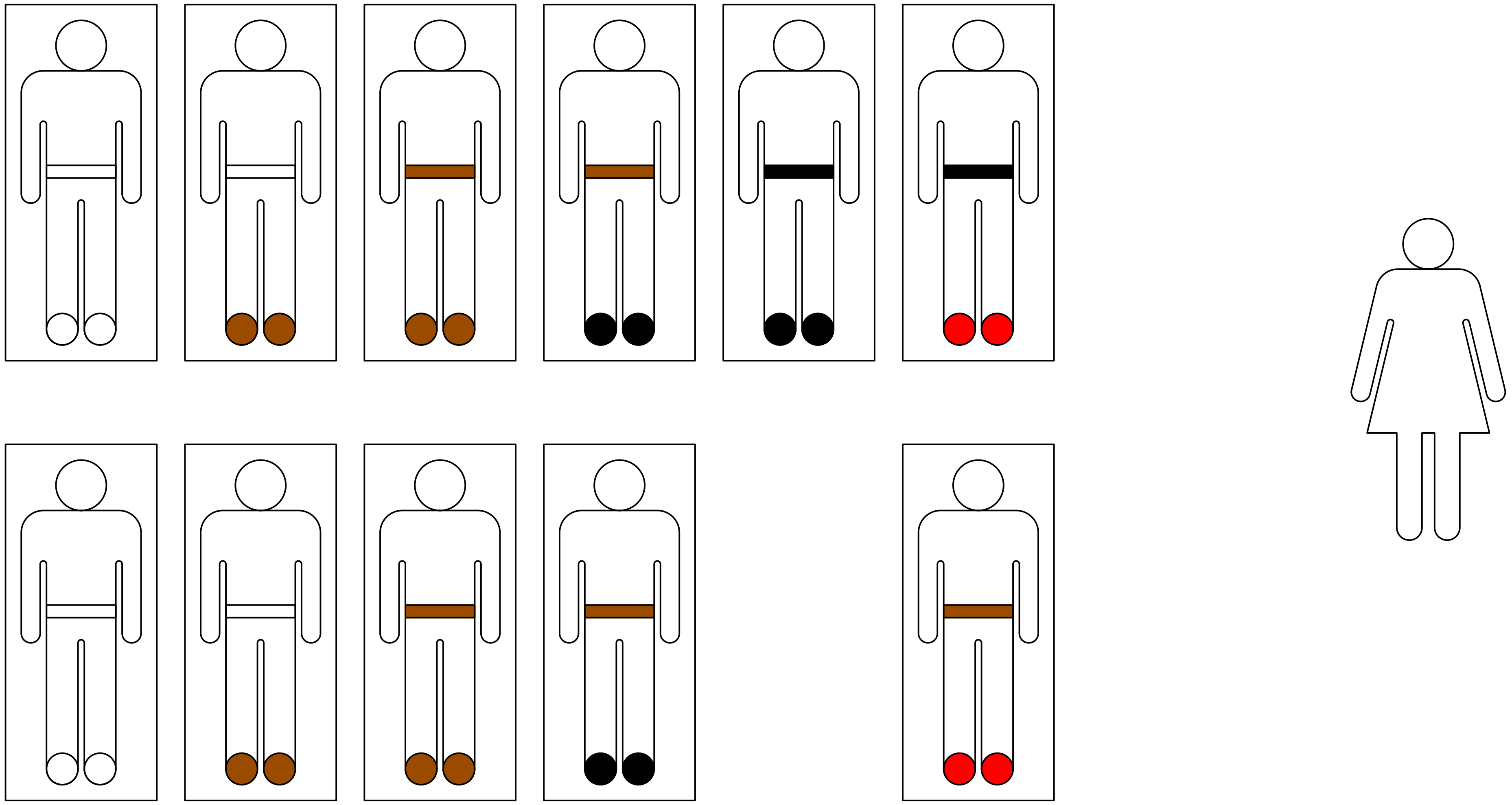
$$\forall s, v, v'.\ put\ (put\ (s, v), v') = put\ s\ v' \qquad \text{(PutPut)}$$

Then $S \simeq V \times C$ for some *complement* type C—"*constant complement*".

Note asymmetry: source S is primary, and completely determines view V.

History-ignorance, very well-behavedness

A parable about me and my shoes.



Symmetric lenses

A *symmetric lens* $(putr, putl) : A \leftrightharpoons_C B$ consists of a pair of functions

$$putr : A \times C \to B \times C$$
$$putl : B \times C \to A \times C$$

satisfying two *round-tripping* laws:

$$\forall a, b, c, c' .\ putr\ (a, c) = (b, c') \Rightarrow putl\ (b, c') = (a, c') \qquad \text{(PutRL)}$$
$$\forall a, b, c, c' .\ putl\ (b, c) = (a, c') \Rightarrow putr\ (a, c') = (b, c') \qquad \text{(PutLR)}$$

Induces *consistent states* (a, c, b) such that $putr\ (a, c) = (b, c)$ and $putl\ (b, c) = (a, c)$.

Again, ‘put–put’ laws

$$\forall a, a', b, c, c' .\ putr\ (a, c) = (b, c') \Rightarrow putr\ (a', c') = putr\ (a', c)$$
$$\forall a, b, b', c, c' .\ putl\ (b, c) = (a, c') \Rightarrow putl\ (b', c') = putl\ (b', c)$$

are rather strong.

Ordered

'Put–put' laws are about *composition* of updates.

The unwanted strength of put–put arises from the unreasonable expectation that *arbitrary updates* can be combined into one. Two 'simple' updates do not necessarily make another 'simple' update.

What if we relax that constraint? Only require composition of 'compatible' updates, whatever that means.

In particular, consider the case in which 'states' are sets of elements, and the *simple* updates are to insert some elements, *or* to delete some elements—but not both.

The state space is ordered (here, by inclusion), and the simple updates are monotonic wrt that ordering.

Now, the composition of two *similar* simple updates (both inserts, or both deletions) is again a simple update. For simple updates, 'put–put' is not overly strong.

Delta-based

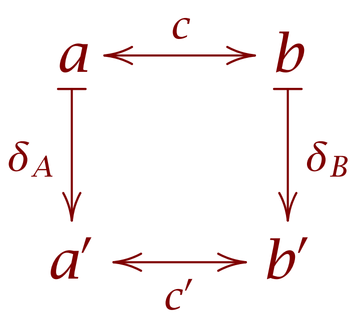
Alternative perspective on put–put problem: it arises from taking a *state-based* approach to BX: the input to a *put* operation is a new state.

Then the *put* operation has two tasks:

alignment: find out what has changed

propagation: translate that change

A *delta-based* approach separates those two tasks. In particular, the input to consistency restoration is not just a new state a', the result of an update, but the update $\delta : a \mapsto a'$ itself (so alignment is no longer needed).



Forwards propagation takes *correspondence* c and update δ_A to update δ_B and corr c'.
Backwards propagation takes c, δ_B to δ_A, c'.

This approach has rather nicer properties.

Another parable

Categorical

The ordered and delta-based approaches can be unified and generalized categorically.

Represent a state space A and its transitions $\delta : a \to a'$ as a category $\mathbf{A}$ (think "directed graph"). A lens $(G, P) : \mathbf{A} \rightleftharpoons \mathbf{B}$ is a pair where

- $G : \mathbf{A} \to \mathbf{B}$ is a functor
- $P : |G/\mathbf{B}| \to |\mathbf{A}^2|$ is a function, taking a pair $(a, \delta_B : G(a) \mapsto b')$ to a transition $\delta_A : a \mapsto a'$

satisfying certain properties analogous to (PutGet), (GetPut), (PutPut).

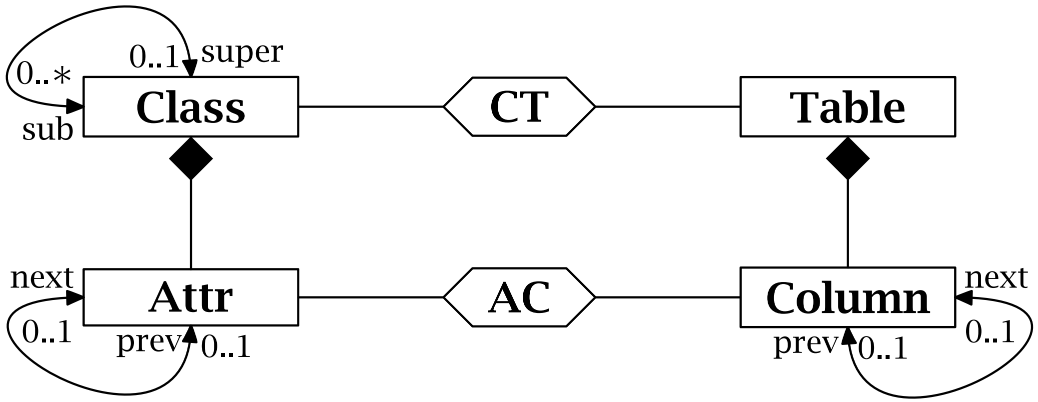
Recover the set-based approach via the *codiscrete* category, which has precisely one arrow between any pair of objects.

Recover the ordered approach by considering the poset as a category.

Triple graph grammars

Arising from work in graph rewriting, 1980s–:

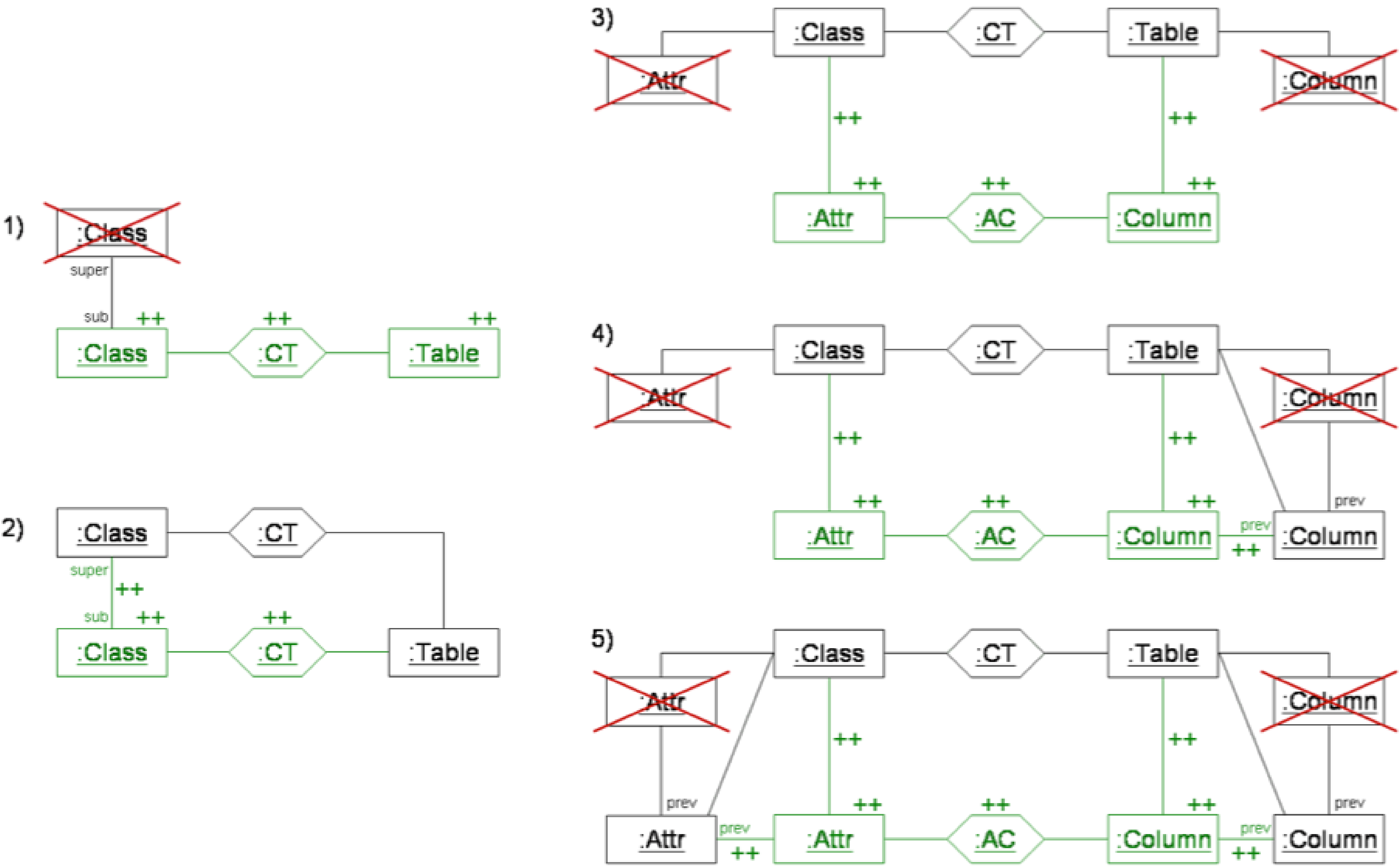
- *grammar* specifies allowable graphs
- *correspondence* structure relating two graphs



(from Andy Schürr's *"15 Years of TGGs"*)

- forward/backward *transformations*,
  from graph to partner plus correspondence

For example. . .


1)
:Class
super
sub
++
:Class
++
:CT
++
:Table
2)
:Class
:CT
super
++
sub
++
:Class
++
:CT
:Table
3)
:Class
:CT
:Table
:Attr
:Column
++
++
++
++
++
:Attr
:AC
:Column
4)
:Class
:CT
:Table
:Attr
:Column
++
++
++
++
++
prev
:Attr
:AC
:Column
prev
++
:Column
5)
:Class
:CT
:Table
:Attr
:Column
++
++
prev
++
++
++
prev
:Attr
prev
++
:Attr
:AC
:Column
prev
++
:Column