Towards an algebraic theory of bidirectional transformations

Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Special thanks
to the PC chairs, to Leonid Libkin and to Benjamin Pierce

Plan

1. Bidirectional model transformations
2. What is sanity?
3. A special case: lenses
4. Algebra of lenses – beginnings!
5. Open issues and conclusions

Introduction

Why model transformations?

Model-driven development is fashionable. But what is it?

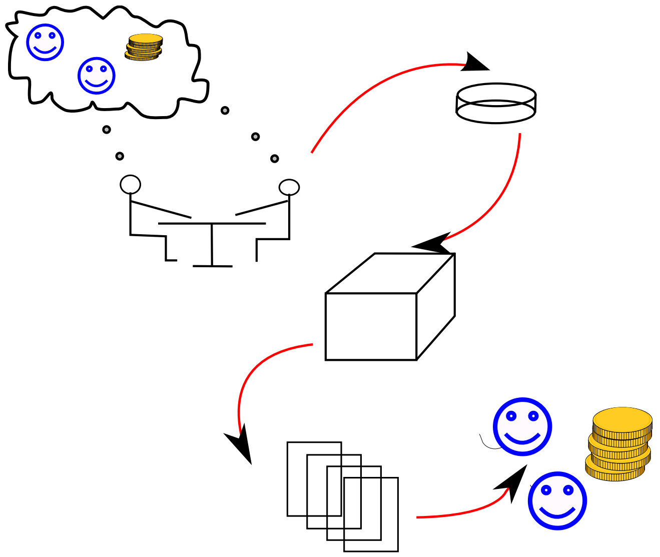
What's a model?

Let's say: a model is any precise representation of some of the information needed to solve a problem using a computer.

Where do model transformations come in?

Model transformations are the engine of MDD.

MDD idea(I)

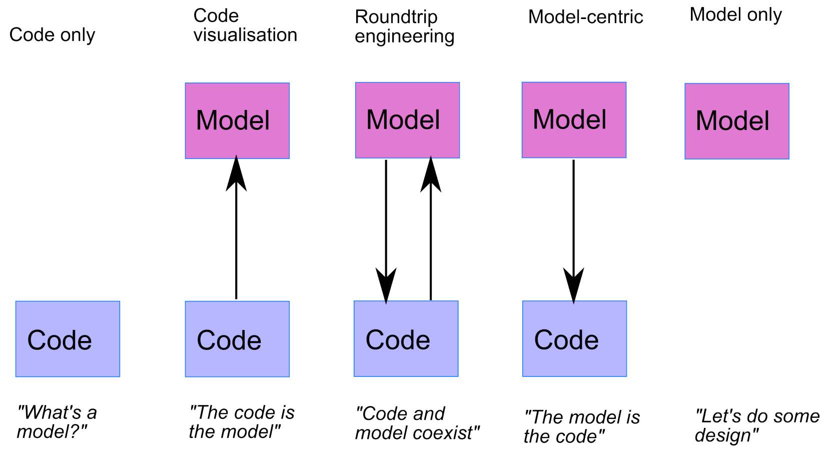


Benefits

- Generate boilerplate code, don't write it: "write once"
- Do analysis and design in a graphical language that domain experts can understand
- Manage commonalities between versions/families of systems
- Make it easy to adapt to changes in e.g. requirements or e.g. library version
- Eliminate "slips"

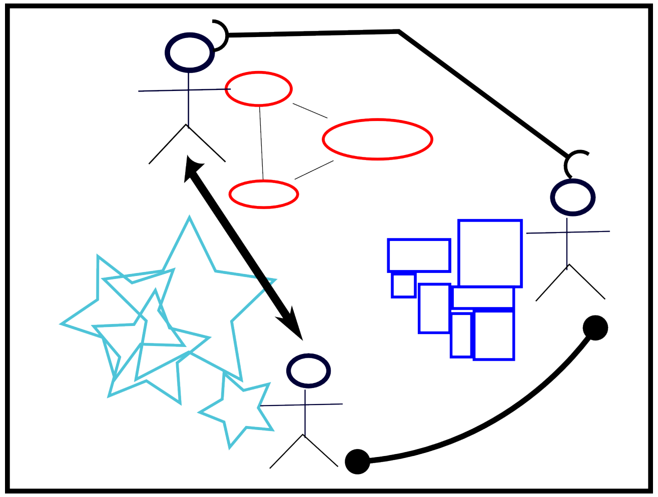
See `http://www.omg.org/mda/products_success.htm`

Rising importance of models



Alan Brown, IBM, *An introduction to MDA*

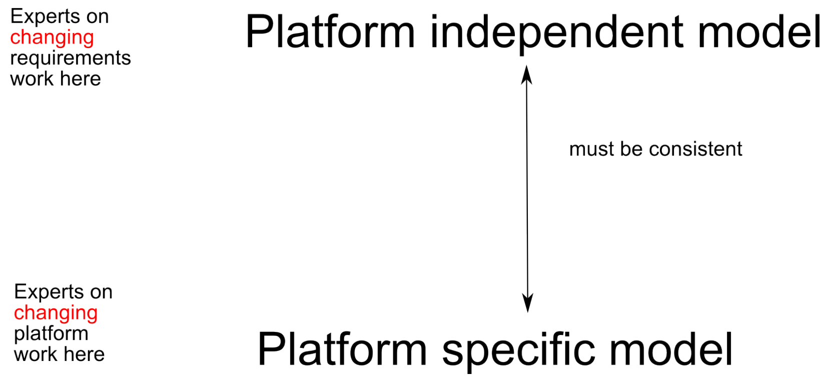
But inside the black box



The reality is that any of these models may change, with knock-on effects.

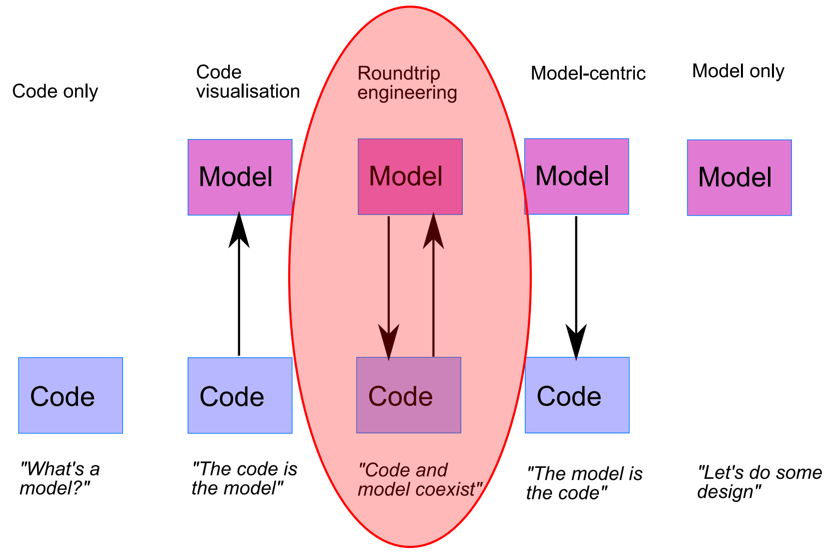
Multiple models even not counting code

For example, here's the classic OMG Model Driven Architecture picture:


Experts on changing requirements work here
Platform independent model
must be consistent
Experts on changing platform work here
Platform specific model

So we'll (practically) always be in the bidirectional case.

Rising importance of models



Transformations as restoration of consistency

Consistency

The most basic concept is that of two models being consistent – this is what we want to check, and what we want somehow to enforce.

Generally, consistency will depend on some, but not all, of the information in the models – to check consistency, you won't usually have to look at every detail.

It is a relation

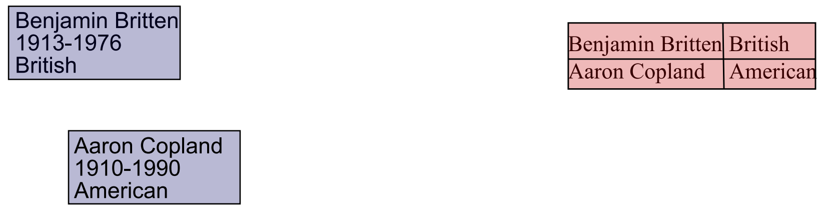
$$R \subseteq M \times N$$

Classic examples

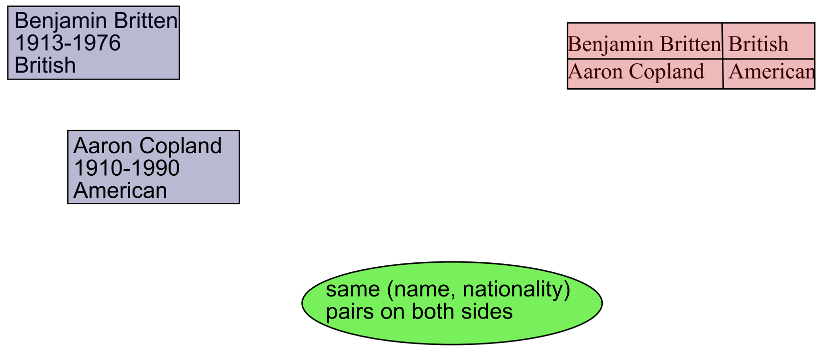
1. UML class diagrams ↔ RDBMS schemas
2. UML models ↔ code

In both cases, each side has information not represented on the other.

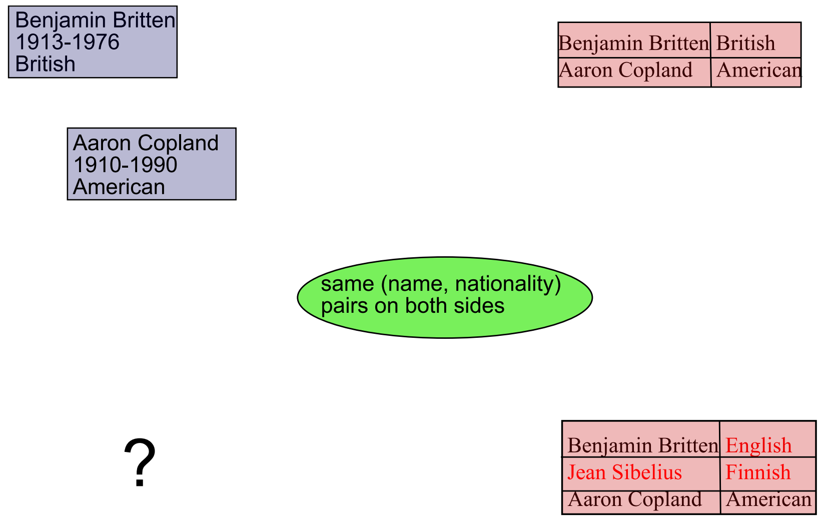
Simple example (after Foster, Pierce et al.)



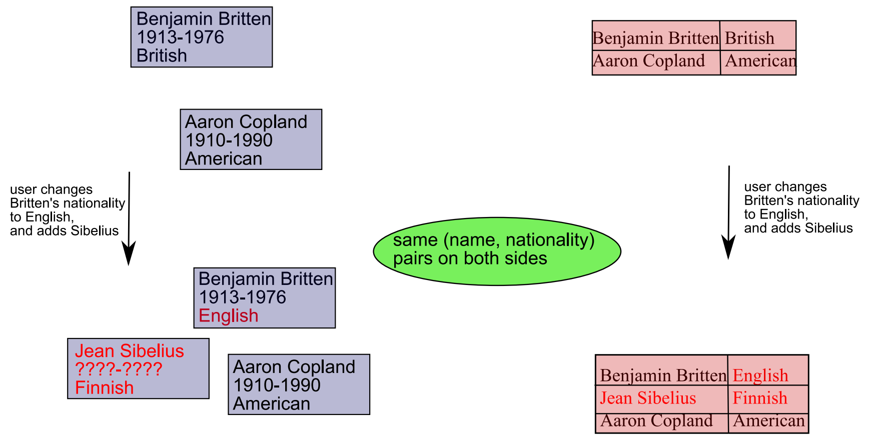
Simple example (after Foster, Pierce et al.)



Simple example (after Foster, Pierce et al.)



One of many reasonable choices


Benjamin Britten 1913-1976 British
Aaron Copland 1910-1990 American
user changes Britten's nationality to English, and adds Sibelius
Benjamin Britten 1913-1976 English
Jean Sibelius ????-???? Finnish
Aaron Copland 1910-1990 American
same (name, nationality) pairs on both sides
Benjamin Britten | British
Aaron Copland | American
user changes Britten's nationality to English, and adds Sibelius
Benjamin Britten | English
Jean Sibelius | Finnish
Aaron Copland | American

Good, we restored consistency

But in doing so, we made choices – there was more than one LH model consistent with the updated RH model.

To decide which was best, we needed to look at latest available versions of both models.

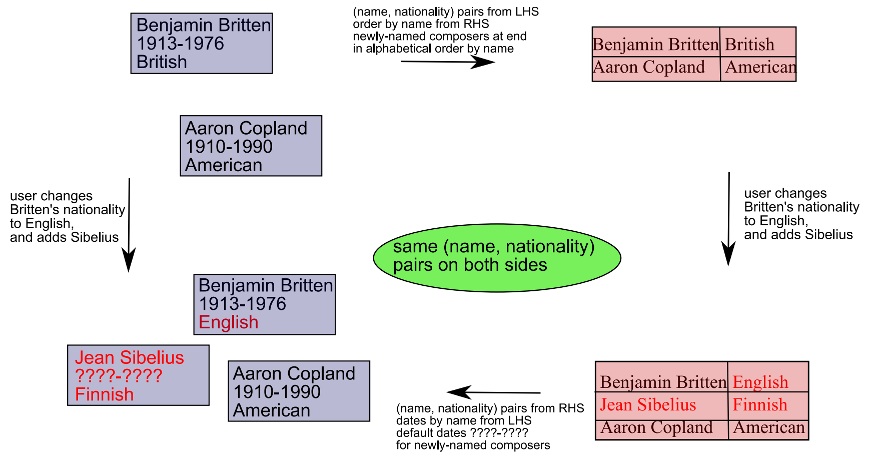
Might even use more info, if feasible...

Not an unfortunate accident, but essential characteristic of the problem area.

Think about it: if you could get away with modifying only one model, regenerating the other when you wanted, *why wouldn't you do that?* Unfortunately, important information may simply not be there.

Moral: if a bidirectional transformation is needed at all, it is probably non-bijective.

Systematically, how did that transformation happen?


Benjamin Britten 1913-1976 British
(name, nationality) pairs from LHS order by name from RHS newly-named composers at end in alphabetical order by name
Benjamin Britten | British
Aaron Copland | American
Aaron Copland 1910-1990 American
user changes Britten's nationality to English, and adds Sibelius
same (name, nationality) pairs on both sides
user changes Britten's nationality to English, and adds Sibelius
Benjamin Britten 1913-1976 English
Jean Sibelius ????-???? Finnish
Aaron Copland 1910-1990 American
(name, nationality) pairs from RHS dates by name from LHS default dates ????-???? for newly-named composers
Benjamin Britten | English
Jean Sibelius | Finnish
Aaron Copland | American

How to deal with non-bijective consistency

So, consistency will normally be a non-bijective relation: for any source model, there will be a (possibly infinite) set of consistent target models.

Is it acceptable for the tool to choose any one?

Absolutely not! Tool must be deterministic.

Shall we try to make the tool choose the simplest one?

That's no better! Mustn't lose the user's data.

Well, shall we just say "OK, when there's more than one solution, the tool must be interactive: the user has to choose"?

Sometimes that might be the best we can do.

But ideally, we should let a programmer specify exactly how consistency should be restored.

No problem!

Programmers love specifying things. We don't even need special programming languages: models are saved as XML after all...

Programmer writes one Perl/OCaml/Java program to roll changes forward, another to roll changes backwards... done!

Umm... programmers (and their managers) never do silly things, do they? Like:

- writing programs that violate basic sanity conditions;
- modifying one program, but forgetting to keep something else consistent.

What should a model transformation (never) be allowed to do?

What is sanity?

What's a sensible transformation?

Two conditions are easy to identify and justify, on the basis that

the job of the transformation is to make those changes which are necessary to restore consistency, and ONLY those changes

1. Correctness – after the transformation, the models are consistent
2. Hippocraticness – if they were already consistent, the transformation does nothing.

Notice that in the case of a non-bijective consistency relation, this is already restrictive.

It already proves that no pair of functions $M \leftrightarrow N$ can do the job.

A pair of functions is not enough

Suppose we had a non-bijective consistency relation and functions
$f : M \longrightarrow N$
$g : N \longrightarrow M$
which are supposed to give a correct and hippocratic transformation.

Pick (wlog) (m, n) and (m, n') consistent, $n \neq n'$.
$f(m) = n$ and $f(m) = n'$ by hippocraticness: contradiction!

This is why we need:

$R \subseteq M \times N$	– consistency
$\overrightarrow{R} : M \times N \to N$	– roll a change in M forward to N
$\overleftarrow{R} : M \times N \to M$	– roll a change in N backwards to M

Example

R :	same (name, nationality) pairs on both sides
$\overrightarrow{R}(m,n)$:	build n' from m and n by: take the (name, nationality) pairs from m, take the ordering from n, add newly-named composers at the end of n in alphabetical order by name; return n'
$\overleftarrow{R}(m,n)$:	build m' from m and n by: take the (name, nationality) pairs from n, take the dates by name from m, use default dates ????-???? for any newly-named composers; return m'

But that's not enough

Some utterly insane transformations are correct and hippocratic.

E.g., in the composers example (consistency as before):

$\overrightarrow{R}(m,n) =$ if $R(m,n)$ then n,
else (name, nationality) pairs alphabetically ordered by name

(discarding user's chosen order from n).

$\overleftarrow{R}(m,n) =$ if $R(m,n)$ then m,
else a set of composer objects with name and nationality taken from n, and all dates set to 2005-2009

(discarding the dates that were already in m).

What else does it take for a transformation to seem reasonable?

What's wrong here?

Two ways to look at it (at least):

1. the silly transformation is needlessly discarding information: so transformations are not undoable;
2. it's taking advantage of the fact that if the models are inconsistent at all, it's utterly unconstrained.

Aim: behaviour is "reasonably composable and undoable", and if the models get "slightly" out of sync, then the transformation should propose a "small" change.

Existing methods – graph transformations, bidirectional programming languages, etc. – tend to achieve something like this by having big transformations built up from small pieces, each of which will be correct and hippocratic.

Let's think about undoability first.

Undoability

More controversial.

Suppose you are working with model m, which is consistent with model n.

You modify m to m'.

You apply the transformation, getting an updated n' consistent with m'.

You realise you made a mistake, and revert to m.

You apply the transformation again.

Do you expect to get back exactly n, i.e., to where you started?

If yes...

... then you expect transformations to be *undoable*:

$$R(m,n) \implies \overrightarrow{R}(m, \overrightarrow{R}(m', n)) = n$$

$$R(m,n) \implies \overleftarrow{R}(\overleftarrow{R}(m, n'), n) = m$$

Why undoability may be too strong

Suppose the change you made was to *delete* some information from m to get m'.

(E.g., you deleted a composer on the LHS)

When you applied the transformation, it deleted the "corresponding information" from n, yielding n'.
(E.g., deleted that composer's entry in the ordered list)

But it also deleted any information which was "stuck" to that information in n, even if it wasn't represented in m.
(It forgot where in the list the composer had been.)

So when you reverted to m, you restored all the information that was visible to you...
(You recreated your composer, dates, and nationality...)

... but maybe not all the information that had been deleted. Maybe some has to be replaced with "default values", so that you don't get back to exactly where you started.
(The composer ends up at the end of the list.)

Thinking structurally

What would an algebraic approach be?

Would like some "more detailed" kind of principled way to talk about conditions that are true of transformations

- that would still be independent of how the transformation is defined;
- ideally, that would let us do composition, including of transformations written in different languages;
- that would "explain why" undoability is so useful but so strong;
- that would give us some notion of how small changes should correspond to small changes...

(Don't get your hopes up for today!)

How does the transformation structure the model spaces?

Suppose we have a transformation $R : M \leftrightarrow N$, with forward and backward transformations $\overrightarrow{R} : M \times N \rightarrow N$ etc.

Given $m \in M$, which other elements of M relate to it via R?

- the set of elements which are *the same* from the point of view of N (none of the difference is visible wearing N glasses):

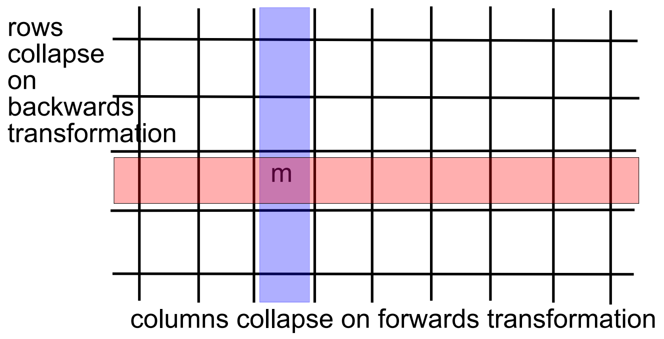
$$\{m' \in M : \forall n \in N. \overrightarrow{R}(m,n) = \overrightarrow{R}(m',n)\}$$

- the set of elements *all of whose difference from m* is visible wearing N glasses:

$$\{m' \in M : \forall n \in N. \overleftarrow{R}(m,n) = \overleftarrow{R}(m',n)\}$$

We get two "orthogonal" equivalence relations.

Suggestive picture


rows collapse on backwards transformation
m
columns collapse on forwards transformation

NB in general not every square will be occupied; and the space may in fact be uncountable.

Edits

Edits can be

- total, or not... but we can fix that by decreeing that if they aren't applicable they do nothing.
- invertible (permutations) or not.

Doing nothing is an edit; edits can be composed.

When m is edited to m', there can legitimately be many edits that could have that effect. Some may be permutations, some not.

We'll insist that there should always be at least one – i.e., that the monoid of edits is transitive on the space.

(Later, we'll insist that edits be permutations, i.e., invertible.)

Corresponding changes

Suppose we have a consistent pair of models (m, n), and the user edits m.

The edit is thought of intentionally: e.g. "rename all classes named `*Impl` to `*Implementation`"; "add a state diagram to every active object"; "delete all sequence diagrams"; etc.

What a transformation has to explain is:

"if the user does edit g on model m, what edit should be done on corresponding model n? And vice versa?"

$G \leftrightarrow H$

Now restrict to a special case

in which N is a strict abstraction of M

This is what's considered in Pierce, Foster et al.'s *lenses*.

get : $C \longrightarrow A$

put : $A \longrightarrow C \longrightarrow C$

(also create(a) short for $put\ a\ \Omega_C$)

$R(c,a)$	$get\ c = a$
$\overrightarrow{R}(c,a)$	$get\ c$ (NB independent of a)
$\overleftarrow{R}(c,a)$	$put\ a\ c$

Correctness and hippo correspond to the lens laws:

$$get\ \Omega_C = \Omega_A$$

$$put\ (get\ c)\ c = c \qquad \textsc{GetPut}$$

$$get\ (put\ a\ c) = a \qquad \textsc{PutGet}$$

Lens programming

Lens programs built up from basic lenses using combinators.

Proofs of lots of results like "if we combine lenses which satisfy the lens laws using a combinator defined [thus], then the result will satisfy the lens laws".

Lens programs take models as inputs – they do not talk about edits.

What do the lens laws look like algebraically?

Suppose Π_C is a suitable transitive monoid of edits of C.

Lifting a lens to the edit monoids

$$\mu : \Pi_C \longrightarrow M(A)$$

defined as

$$(\mu g)(a) = get\ (g(create\ a))$$

is a monoid homomorphism.

If we write H for the image of μ, it turns out to act transitively on A. The other way:

$$\lambda : H \longrightarrow \Pi_C$$

given by

$$(\lambda h)(c) = put\ h(get\ c)\ c$$

is well-defined, and $\mu\lambda$ is the identity on H.

However λ is *not* necessarily a homomorphism.

In other words...

We have a short exact sequence of monoids

$$1 \to K \to \Pi_C \xrightarrow{\mu} H \to 1$$

So what? So nothing useful, so long as it doesn't *split*.

That means we need a monoid homomorphism $H \to \Pi_C$ which is a right inverse of μ.

λ would do fine, if only it were a homomorphism...

Getting λ to be a homomorphism

In a SES arising from a lens as above,

λ is a monoid homomorphism if and only if the transformation is undoable.

(In lens terminology, iff it is *very well-behaved*, i.e. satisfies PutPut

$put\ a'\ (put\ a\ c) = put\ a'\ c$

– the lens analogue of the undoability condition.)

From now on we'll work with undoable transformations only.

From monoids to groups

It's convenient to restrict now to considering only the invertible edits, i.e., to work with groups rather than monoids.

(Possible justification: in practice, we don't get to know what edit the user had in mind, only what it did on the model(s) we see. They *may* have meant a permutation, so our transformation certainly has to do the right thing if they did. And if that already restricts it enough to be sensible...)

So in the special case

(i.e. 1. one model-space a strict abstraction of the other; 2. undoable transformation; 3. invertible edits)

we get a *split* short exact sequence of edit *groups*:

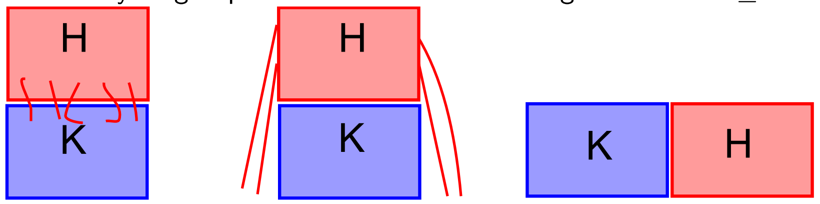
$$\begin{tikzcd} 1 \arrow[r] & K \arrow[r] & G \arrow[r, shift left, "\mu"] & H \arrow[r] \arrow[l, shift left, "\lambda"] & 1 \end{tikzcd}$$

and $G \simeq K \rtimes H$, i.e. $K \trianglelefteq G$, $H \leq G$ and

- $KH = G$
- $K \cap H = 1_G$

Extremely informal picture!

Three ways a group G can be built from H together with $K \trianglelefteq G$:



Not a product	Semi-direct product	Direct product
Dependent	Semi-independent	Independent
Maybe $H \not\leq G$	$H \leq G$	$H \trianglelefteq G$
	$g = h.k \neq k.h$	$g = h.k = k.h$

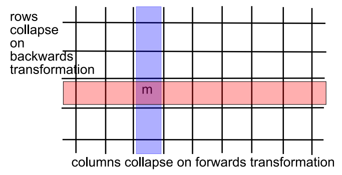
Now,

G acts on (is the edits of) C,

H acts on A,

– what does K act on, and how does it fit together?

Structure of the semi-direct action


rows collapse on backwards transformation
m
columns collapse on forwards transformation

Now all grid squares are filled; write $C = L \times A$

G permutes the squares, and can be written as $K \rtimes H$

H acts on A – stabilises rows, simply permutes the elements of each row in the same way: $h(l, a) = (l, ha)$.

K acts on L – stabilises columns, permuting the elements of each column, maybe *not* all in the same way.

And this is why undoability is so strong.

Most trivial possible examples

Any case where a concrete model consists of the abstract model plus some more completely independent information puts us in the direct product case.

Suppose A comprises UML models, with edit group H

L comprises nursery rhymes, with edit group K

Of course $C = L \times A$ has edit group $K \times A$, acting pointwise.

$get\ (l, a) = a$

$put\ a\ (l, a') = (l, a)$

Trivial, but at least demonstrates that there *are* non-bijective undoable transformations!

Non-bijective, non-direct-product example

C : positions of solid wooden equilateral triangle with numbered corners and one red face; edit group $G \simeq S_3$

A : a single boolean ("is the red face uppermost?"); edit group $H = \{1_H, h\} \simeq C_2$

In lens terms:

get c returns true iff c had red face uppermost

put a c checks whether a correctly reports whether c has red face uppermost. If so, it does nothing (returns c); if not, it flips c about its top corner and returns that.

In algebraic terms:

μg is h iff g involved flipping the triangle, otherwise 1_H

λh is flipping the triangle about its top corner.

$$1 \to K \to G \xrightarrow{\mu} H \to 1$$

Turns out $K \simeq C_3$, acting on a three-element set L which we can

Building a lens from a sequence

Suppose we have a short exact sequence of edit monoids: can we build a lens from it?

Well, not necessarily – the sequence must be *lens-like*, i.e., must not use *intentional* information about the user's edits.

e.g. if there are two edits g_1, g_2 which have the same effect on a model c, then for the sequence to be lens-like, $\mu(g_1) = \mu(g_2)$.

That given, it works as expected: we can build a lens, and if the sequence originally came from a lens, the one we build is the one we started with.

Conclusions

- Bidirectional transformations are interesting and important :-)
- especially non-bijective ones, which are more than pairs of single-argument functions.
- They should be correct and hippocratic, and we might want them to be undoable...
- Lenses, the special transformations that work between a space and a strict abstraction of it, "are" short exact sequences of edit monoids.
- Considering only invertible edits,
  undoability corresponds to the splitting of the sequence, which imposes a semi-direct product structure on the original group of edits.
- There is lots more still to do...

Open questions/ongoing work

- (How) can we exploit group theory to understand structure of complex transformations?
- Clarify the roles of the restrictions we imposed.
- Non-lens-like SESs
- Topology?

How does all this relate to:

- graph transformations?!
- the database literature on data exchange (recoveries, (quasi-)inverses etc.)?
- other work from the Harmony group, e.g., quotient lenses?

A few references

The paper in the proceedings, and:

S., *Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions*, MODELS'07

S., *A landscape of bidirectional model transformations*, post-proceedings of GTTSE'07

Foster, Greenwald, Moore, Pierce, Schmitt: *Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem*, ACM TOPLAS 2007, etc.

Fagin, Kolaitis, Miller, Popa: *Data exchange: semantics and query answering*, ICDT'03, etc.

Arenas, Pérez, Riveros: *The recovery of a schema mapping: bringing exchanged data back*, PODS'08

Extra slides

And what about topology?

Temptingly obvious first thoughts:

if there is an information order on the models, e.g.

$m_1 \leq m_2$ if m_2 contains all the model elements from m_1

(with Ω at the bottom, obviously)

then it's natural to require that the $\overrightarrow{R}$ and $\overleftarrow{R}$ be pointwise monotonic; then there's a standard notion of continuity...

Unfortunately, not that simple :-(

Consider *M* class diagrams, *N* instance diagrams. Consistent iff the instance diagram conforms to the class diagram. Transformations restore consistency. But sometimes adding model elements to the class diagram might necessitate taking some away from the instance diagram: model elements can function as constraints.

Why can't I just...

- force bijectivity by hiding the info from one model inside the other?
  - Commonly done, can be good. But terribly fragile. No good if you don't have complete control of the model.
- force undoability using standard tool mechanisms – literally remember what changes were made?
  - Great, do that – provided you have access to all the necessary information (distributed team? using different tools?)