Towards modularity and compositionality in the engineering of model transformations

Perdita Stevens

September 2011
Plan

- Background: MDD, model transformations, QVT-R
- *when* and *where* for structuring QVT-R transformations
- QVT-R game
- It’s logic, Captain, but not as we know it
- Beyond logic
- The sequential composition and bidirectionality problem
- Engineering issues
Model-driven development

involves placing models at the centre of the software development process.

A model is an abstraction of some aspect of (part of) the system, usually represented graphically.
Model-driven development

involves placing models at the centre of the software development process.

A model is an abstraction of some aspect of (part of) the system, usually represented graphically.

The hope is that that abstraction makes it better to work with models than with programs.

However, software development involves making lots of models, abstracting away different things. They all change, and have to be kept consistent.

Model transformations are used to keep models in sync.
Model-driven development

involves placing models at the centre of the software development process.

A model is an abstraction of some aspect of (part of) the system, usually represented graphically.

The hope is that that abstraction makes it better to work with models than with programs.

However, software development involves making lots of models, abstracting away different things. They all change, and have to be kept consistent.

Model transformations are used to keep models in sync.

(A model transformation is, err, a program.)
Bidirectionality

Where models have to be kept consistent despite deliberate changes to any of them, we need:

▶ to be able to specify what “consistent” means to us
▶ to be able to specify exactly how consistency should be restored when it is lost

NB consistency is generally a many-many relation.

The precise notion of consistency, and how to restore it, is itself (a) hard to decide (b) likely to change often during development.

The models concerned may be very large.

(mathematics coming in few slides, patience...
Bidirectional modelling languages

Decisions and why you might want to make them:

- Specify consistency and how to restore it (whatever changes) in one text – why? so that they change together
- Prioritise the specification of consistency – why? because that’s generally the hard part, and what engineers need to understand
- Try to let parts of the transformation go with parts of the models – why? because both transformation and model are probably too complex for anyone to be interested in the whole thing at once.*
Bidirectional modelling languages

Decisions and why you might want to make them:

▶ Specify consistency and how to restore it (whatever changes) in one text – why? so that they change together
▶ Prioritise the specification of consistency – why? because that’s generally the hard part, and what engineers need to understand
▶ Try to let parts of the transformation go with parts of the models – why? because both transformation and model are probably too complex for anyone to be interested in the whole thing at once.*

*PS We – or rather, they – seem to have decided that the solution, i.e. transformation, structure should match the problem, i.e. model, structure, in this case. Sort of...
Setting (for today – there are others)

Let $M$ and $N$ be sets of models to be related. A *transformation* between $M$ and $N$ consists of:

- a relation $R \subseteq M \times N$ ("is consistent with")
- a function $\overrightarrow{R} : M \times N \rightarrow N$ ("propagate forwards")
- a function $\overleftarrow{R} : M \times N \rightarrow M$ ("propagate backwards")

NB state-based, i.e. no intensional information about the edit that happened. Also, no stash of extra information, e.g., no complement.

* just one – see later
QVT Relations (QVT-R)

OMG standard language,
QVT Relations (QVT-R)

OMG standard language, dating back several years,
QVT Relations (QVT-R)

OMG standard language,
dating back several years,
years in which MDD has become increasingly mainstream.
QVT Relations (QVT-R)

OMG standard language, dating back several years, years in which MDD has become increasingly mainstream yet QVT-R is not in serious use.

+ Clear, usable syntax.

Declarative, bidirectional, based on specifying relations on parts of models.
QVT Relations (QVT-R)

OMG standard language,
dating back several years,
years in which MDD has become increasingly mainstream
yet QVT-R is not in serious use.

+ Clear, usable syntax.

Declarative, bidirectional, based on specifying relations on parts of models.

- But hardly any available tool support: Medini QVT (open source engine, uses QVT-R syntax only); TATA’s ModelMorf...
QVT Relations (QVT-R)

OMG standard language, dating back several years, years in which MDD has become increasingly mainstream yet QVT-R is not in serious use.

+ Clear, usable syntax.
Declarative, bidirectional, based on specifying relations on parts of models.
- But hardly any available tool support: Medini QVT (open source engine, uses QVT-R syntax only); TATA’s ModelMorf...

Why? Sufficient problem: this kind of language really needs clear semantics!
How QVT-R is used

A QVT-R transformation is a single text, defined in terms of metamodels.

You run the transformation:

- in a direction: examine one model, regarding other(s) as authoritative
- in
  - checkonly mode: is m2 OK according to authoritative m1? Say m1 and m2 are consistent if both directions succeed.*
  - enforce mode: modify m2 so that it is OK according to authoritative m1.

“Check then enforce”: enforce must not do anything if checkonly returns true.
How QVT-R is used

A QVT-R transformation is a single text, defined in terms of metamodels.

You run the transformation:

- in a direction: examine one model, regarding other(s) as authoritative
- in
  - checkonly mode: is \(m_2\) OK according to authoritative \(m_1\)? Say \(m_1\) and \(m_2\) are consistent if both directions succeed.\(^*\)
  - enforce mode: modify \(m_2\) so that it is OK according to authoritative \(m_1\).

“Check then enforce”: enforce must not do anything if checkonly returns true.

Questionable decision, and trivialises our nice Galois connection, but in the standard.
Implicit assertions (implied by QVT)

1. Transformations are functions, i.e., deterministic – because it will not be acceptable for tools to interpret the same transformation text on the same models differently.

2. Transformations may need to look at both models, not just one – otherwise we will be limited to bijective transformations.
What else needs to be true?

1. correctness
2. hippocraticness

Maybe, a stronger condition such as *undoability* – not talking about that today.
Correctness

The “job” of the transformations is to enforce the relation. So they better had do that:

$$\forall m \in M \ \forall n \in N \quad T(m, \overrightarrow{T}(m, n))$$

$$\forall m \in M \ \forall n \in N \quad T(\overleftarrow{T}(m, n), n)$$
Hippocraticness

The QVT standard states that QVT-Relational should have a “check then enforce” semantics: that is, if a pair of models is already consistent, the transformation should not change them. This is automatic for bijective transformations, but quite a restriction for general bidirectional transformations:

\[
T(m, n) \implies \overrightarrow{T}(m, n) = n
\]

\[
T(m, n) \implies \overleftarrow{T}(m, n) = m
\]
A basic relation

relation ThingsMatch
{
  s : String;
  checkonly domain m1 thing1:Thing {value = s};
  checkonly domain m2 thing2:Thing {value = s};
}

Relation ThingsMatch holds of bindings to thing1 in model m1 and thing2 in model m2 provided that thing1.value = thing2.value
A basic relation

transformation Basic (m1 : MM ; m2 : MM)
{
    top relation ThingsMatch
    {
        s : String;
        checkonly domain m1 thing1:Thing {value = s};
        checkonly domain m2 thing2:Thing {value = s};
    }
}

Transformation Basic returns true when executed in the direction of m2 iff for every binding to thing1 in model m1 there exists a binding to thing2 in model m2 such that

thing1.value = thing2.value
Invoking relations: when and where clauses

“The when clause specifies the conditions under which the relationship needs to hold, so the relation ClassToTable needs to hold only when the PackageToSchema relation holds between the package containing the class and the schema containing the table. The where clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. Hence, whenever the ClassToTable relation holds, the relation AttributeToColumn must also hold.”

relation ClassToTable
{
    domain.uml c:Class { ... stuff mentioning p ...}
    domain.rdbms t:Table { ... stuff mentioning s ... }
    when { PackageToSchema(p, s); }
    where { AttributeToColumn(c, t); }
}
So we have a language in which an utterance is supposed to express simultaneously:

1. a (set of) predicate(s) on a tuple of models
2. a way to make a given predicate hold

and a frustrating lack of clear semantics

In an attempt to clarify the meanings of when and where clauses I designed a two-player game (ICMT’09, SoSyM 2011) for checkonly mode only.

I restricted the class of transformations to what the OMG spec made a decent attempt to cover.
Example transformation

transformation Sim (m1 : MM ; m2 : MM) {
  top relation ContainersMatch {
    inter1,inter2 : MM::Inter;
    checkonly domain m1 c1:Container {inter = inter1};
    checkonly domain m2 c2:Container {inter = inter2};
    where {IntersMatch (inter1,inter2);} }
}

relation IntersMatch {
  thing1, thing2 : MM::Thing;
  checkonly domain m1 i1:Inter {thing = thing1};
  checkonly domain m2 i2:Inter {thing = thing2};
  where {ThingsMatch (thing1,thing2);}
}

relation ThingsMatch {
  s : String;
  checkonly domain m1 thing1:Thing {value = s};
  checkonly domain m2 thing2:Thing {value = s};
}
}
The pair of models we’ll check

Model m1

Model m2
QVT Relations checking as a game

Take:

- a pair of metamodels
- a QVT-R transformation;
- models m1 and m2 conforming to the metamodels.

Assume we have an oracle for checking conformance to metamodel and “local” checking inside relations.

Simplification: let when and where clauses contain only relation invocations.
QVT Relations checking as a game

Take:

- a pair of metamodels
- a QVT-R transformation;
- models \( m_1 \) and \( m_2 \) conforming to the metamodels.

Assume we have an oracle for checking conformance to metamodel and “local” checking inside relations.

Simplification: let \( \text{when} \) and \( \text{where} \) clauses contain only relation invocations.

Let’s define game \( G \) to check in the direction of model \( m_2 \).

Two players, Verifier who wants the check to succeed, \( \text{Refuter} \) who wants it to fail.

Semantics: return true if Verifier has a winning strategy, false if \( \text{Refuter} \) does.
top relation ContainersMatch
{
    inter1, inter2 : MM::Inter;
    checkonly domain m1 c1:Container {inter = inter1};
    checkonly domain m2 c2:Container {inter = inter2};
    where {IntersMatch (inter1, inter2);}
}

Refuter
top relation ContainersMatch
{
  inter1, inter2 : MM::Inter;
  checkonly domain m1 c1:Container {inter = inter1};
  checkonly domain m2 c2:Container {inter = inter2};
  where {IntersMatch (inter1, inter2);} 
}
Refuter;Verifier;Refuter

top relation ContainersMatch
{
    inter1, inter2 : MM::Inter;
    checkonly domain m1 c1:Container {inter = inter1};
    checkonly domain m2 c2:Container {inter = inter2};
    where {IntersMatch (inter1, inter2);} 
}
relation IntersMatch
{
  thing1, thing2 : MM::Thing;
  checkonly domain m1 i1:Inter {thing = thing1};
  checkonly domain m2 i2:Inter {thing = thing2};
  where {ThingsMatch (thing1, thing2);}
}
relation IntersMatch
{
  thing1, thing2 : MM::Thing;
  checkonly domain m1 i1:Inter {thing = thing1};
  checkonly domain m2 i2:Inter {thing = thing2};
  where {ThingsMatch (thing1, thing2);}
}
relation IntersMatch
{
    thing1, thing2 : MM::Thing;
    checkonly domain m1 i1: Inter {thing = thing1};
    checkonly domain m2 i2: Inter {thing = thing2};
    where {ThingsMatch (thing1, thing2);}  
}
relation ThingsMatch
{
    s : String;
    checkonly domain m1 thing1:Thing {value = s};
    checkonly domain m2 thing2:Thing {value = s};
}
relation ThingsMatch
{
    s : String;
    checkonly domain m1 thing1:Thing {value = s};
    checkonly domain m2 thing2:Thing {value = s};
}
## Summary of moves (missing out *when*)

<table>
<thead>
<tr>
<th>Position</th>
<th>Next position</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial (Ref.)</td>
<td>(Verifier, $R$, $B$)</td>
<td>$R$ is any top relation; $B$ comprises valid bindings for all variables from $m_1$ domain</td>
</tr>
<tr>
<td>(Verifier, $R$, $B$)</td>
<td>(Refuter, $R$, $B'$)</td>
<td>$B'$ comprises $B$ together with bindings for any unbound $m_2$ variables.</td>
</tr>
<tr>
<td>(Refuter, $R$, $B$)</td>
<td>(Verifier, $T$, $D$)</td>
<td>$T$ is any relation invocation from the <em>where</em> clause of $R$; $D$ comprises $B$’s bindings for the root variables of patterns in $T$, together with valid bindings for all $m_1$ variables in $T$.</td>
</tr>
</tbody>
</table>
Adding *when*-clauses

relation ClassToTable
{
    domain uml c:Class { ... stuff mentioning p ... }
    domain rdbms t:Table { ... stuff mentioning s ... }
    when { PackageToSchema(p, s); }
    where { AttributeToColumn(c, t); }
}
Adding \textit{when}-clauses

relation ClassToTable
{
  domain uml c:Class { ... stuff mentioning p ...}
  domain rdbms t:Table { ... stuff mentioning s ... } 
  when { PackageToSchema(p, s); }
  where { AttributeToColumn(c, t); }
}

Allow player to "counter-challenge" a \textit{when}-clause...

You challenge me to find a match for your bindings: I have a choice. Either I find one, or I accuse you of cheating by making an unfair challenge, one that doesn’t satisfy the \textit{when} clause. To do that I counter-challenge a relation from the \textit{when} clause, and we swap roles and play in that relation.
Winning conditions

You win if your opponent can’t go.

Infinite plays were simply forbidden by fiat.

One way to enforce this: insist graph of relations with *when* and *where* edges be a DAG.
Winning conditions

You win if your opponent can’t go.

Infinite plays were simply forbidden by fiat.

One way to enforce this: insist graph of relations with *when* and *where* edges be a DAG.

NB QVT spec doesn’t address the issue at all – corresponds to infinite regress of its definitions.
slides about a family of 32 really simple examples, in 31 of which this semantics agreed with ModelMorf – whose developer confirmed that the 32nd was bug in ModelMorf.
Why use a game-theoretic approach?

Claims:

- basis for discussion of what the semantics of QVT-R should be, because
- gives useful separation between local and global checking
- precise, without needing heavy machinery
- intuitive way to understand alternation
- winning strategy is solid evidence of result, which can be checked independently from the means of finding it.

Non-claims:

- a free lunch of any kind
- necessarily exactly the meaning practitioners want, in current form
What that game definition left out

*when* and *where* clauses could only contain (conjunctions of) relation invocations.

More importantly, I gave no winning conditions for infinite plays; I wrote:

> There is probably\(^1\) a reasonable alternative that achieves sensible behaviour by allowing the winner of an infinite play to be determined by whether the outermost clause which is visited infinitely often is a *where* clause or a *when* clause: but this requires further investigation. Note that [OMG] has nothing to say about this situation: it corresponds to infinite regress of its definitions. For now, we will forbid infinite plays. One way to achieve this is to declare any QVT-R transformation with a cyclic *when*-where graph to be ill-formed. However, ...

\(^1\)by thinking from first principles about cases in which a play goes through a *when* (resp. *where*) clause infinitely often, but only finitely often through *where* (resp. *when*) clauses; or by intriguing analogy with \(\mu\) calculus model-checking.
It's logic, Captain, but not as we know it
Modal mu-calculus

\[
\phi := \text{tt} \mid \text{ff} \mid \phi \land \phi \mid \phi \lor \phi \mid X \\
\mid [a] \phi \quad \text{after any } \xrightarrow{a}, \phi \text{ will hold} \\
\mid \langle a \rangle \phi \quad \text{there is some } \xrightarrow{a} \text{ after which } \phi \text{ holds} \\
\mid \nu X. \phi \quad \text{maximal fixpoint of } X = \phi \\
\mid \mu X. \phi \quad \text{minimal fixpoint of } X = \phi
\]

Evaluate \( \phi \) on an LTS using (most intuitively) Stirling’s game. In an infinite play, “the winner is the owner of the outermost fixpoint unwound infinitely often” – Verifier for \( \nu \), Refuter for \( \mu \).

For our purposes convenient to include also \( \neg \phi \) – generally remove by pushing negation inwards, if negation doesn’t vanish, illegal as fixpoints not defined (nonmonotonic)...
Aligning the games

Given a QVT-R transformation $T$ and a pair of models $(M_1, M_2)$ we shall build a mu calculus formula $tr(T)$ and an LTS $lts(T, M_1, M_2)$ such that $(M_1, M_2)$ are consistent according to $T$ iff 
$lts(T, M_1, M_2) \models tr(T)$.

What info goes where? There is choice: the key point is that since most of the information in the models is irrelevant to the truth of this particular $T$, it’s convenient if the LTS depends on all of $T, M_1, M_2$. 
Just what is a transformation to us?

To focus on the interesting stuff, we abstract over types, values, expressions, evaluation and type checking of expressions... assume that can be done.

A transformation $T$ consists of relations with names $\text{rel}(T)$ of which $\text{top}(T) \subseteq \text{rel}(T)$ are the names of relations designated top.

A relation consists of: a unique name $R$; some domains each with a typed domain var, some other vars, and a constraint; a \textit{when} clause and a \textit{where} clause.

\textit{when} and \textit{where} clauses are boolean combinations of relation invocations and constraints.

A relation invocation names a relation and provides values for its domain vars.
The LTS we construct has a distinguished initial node.

Other nodes are pairs \((R, B)\) where \(R \in rel(T)\) and \(B : \text{vars}(R) \rightarrow \text{Val}\) is a set of bindings; that is, a partial map from variables of \(R\) to values.

Values are model elements from the models being evaluated, or constants. (We don’t care....)

Write \(B' \preceq B\) when \(B'\) and \(B\) agree where both are defined and \(\text{dom}(B') \supseteq \text{dom}(B)\).

Labels: challenge, response, ext1, ext2, invokes.
Important subsets of \( \text{vars}(R) \)

\( \text{domainvars}(R) \) - the domain (top level) variables
\( \text{whenvars}(R) \) - all variables mentioned in the \textit{when} clause
\( \text{nonkvars}(R) \) - all variables mentioned in domains other than the \( k \)th, i.e. other than the one we’re checking

All these sets may overlap.
LTS transitions I

Initial $\xrightarrow{\text{challenge}} (R, B)$ if $R \in \text{top}(T)$ and $\text{dom}(B) = \text{whenvars}(R) \cup \text{nonkvars}(R)$

$(R, B) \xrightarrow{\text{response}} (R, B')$

if $\text{dom}(B) = \text{whenvars}(R) \cup \text{nonkvars}(R)$ and $B' \succeq B$ and $\text{dom}(B') = \text{vars}(R)$.

$(R, B) \xrightarrow{\text{ext1}} (R, B')$

if $\text{dom}(B) = \text{domainvars}(R)$ and $B' \succeq B$ and $\text{dom}(B') = \text{domainvars}(R) \cup \text{whenvars}(R) \cup \text{nonkvars}(R)$.

$(R, B) \xrightarrow{\text{ext2}} (R, B')$

if $\text{dom}(B) = \text{domainvars}(R) \cup \text{whenvars}(R) \cup \text{nonkvars}(R)$ and $B' \succeq B$ and $\text{dom}(B') = \text{vars}(R)$. 
(\(R, B\) \text{ invokes } (S, B'))

if \(S\) appears in the \textit{where} clause of \(R\) with arguments \(e_i\),
\(\text{dom}B = \text{vars}(R)\) and \(\text{dom}B' = \text{domainvars}(S)\) with \(B'\) binding
each domain var to the result of evaluating \(e_i\) in \(B\).

(Eliding: that this exposes a mildly interesting wrinkle about how
top relations are different.)

\((R, B) \xrightarrow{\text{invokes}} (S, B')\)

if \(S\) appears in the \textit{when} clause of \(R\), with arguments \(e_i\),
\(\text{dom}B \supseteq \text{whenvars}(R)\) and \(\text{dom}B' = \text{domainvars}(S)\) with \(B'\)
binding each domain var to the result of evaluating \(e_i\) in \(B\).
Translating the transformation: idea

Challenge-response choice of bindings modelled by $[ext1]$, $⟨ext2⟩$ (modulo complication at the start where top relations are special).

Permit boolean combinations of relation invocations: boolean ops translate straight over, except...

To effect pushing negation inwards, we carry around a flag in the translation process and flip it when we meet a negation.

Translating an invocation involves introducing a fixpoint var.
Translating the transformation: idea

\[ tr(T) = \bigwedge_{R_i \in \text{top}(T)} tr_1 \emptyset (R_i, \text{true}) \]

\( tr_1 \) handles challenges to top relations:

\[ tr_1_E (R_i, b) = [\text{challenge}] (\langle \text{response} \rangle (tr_2_E (\text{where } (R_i), b) \lor tr_2_E (\text{when } (R_i), \neg b))) \]

Now \( tr_2 \) is used to translate \textit{when} and \textit{where} clauses. Booleans and constraints are simply transliterated/evaluated, with negation corresponding to changing the flag:

\[ tr_2_E (\emptyset, \text{true}) = \emptyset \]

\[ tr_2_E (\emptyset, \text{false}) = \neg \emptyset \]

\[ tr_2_E (e \text{ and } e', b) = tr_2_E (e, b) \land tr_2_E (e', b) \]

\[ tr_2_E (e \text{ or } e', b) = tr_2_E (e, b) \lor tr_2_E (e', b) \]

\[ tr_2_E (\text{not } e, b) = tr_2_E (e, \neg b) \]
Translating relation invocations

Relation invocations are translated using the environment if they have been seen before:

\[
tr_{2E}(R(\_), \text{true}) = \langle \text{invokes} \rangle E[R] \text{if } R \in \text{dom}E
\]

\[
tr_{2E}(R(\_), \text{false}) = [\text{invokes}] (\neg E[R]) \text{if } R \in \text{dom}E
\]

otherwise using fixed points:

\[
tr_{2E}(R(\_), \text{true}) = \langle \text{invokes} \rangle
\]

\[
\nu X. ([\text{ext1}] (\langle \text{ext2} \rangle tr_{2E[R\mapsto X]}(\text{where}(R, \text{true})) \lor
tr_{2E[R\mapsto X]}(\text{when}(R, \text{false})))
\]

\[
tr_{2E}(R(\_), \text{false}) = [\text{invokes}]
\]

\[
\mu X. ([\text{ext1}] ([\text{ext2}] tr_{2E[R\mapsto \neg X]}(\text{where}(R, \text{false})) \land
tr_{2E[R\mapsto \neg X]}(\text{when}(R, \text{true})))
\]
Obvious consequence

\[ tr2_E(R, false) = \neg tr2_E(R, true) \]

That is, negation really is player-swapping here.

Consequence: we’ve only made one choice of fixpoint. If \textit{where} clauses involve \( \nu \), \textit{when} clauses must involve \( \mu \), and vice versa.
Treatment of recursion

Only recursion example in the OMG spec:

relation AttributeToColumn
{ [...] 
    where { [...] ComplexAttributeToColumn(c, t, prefix) [...] }}
}

relation ComplexAttributeToColumn
{
an, newPrefix: String;
checkonly domain uml c:Class {attribute=a:Attribute {name=an,type=tc:Class {}}};
enforce domain rdbms t:Table {};
primitive domain prefix:String;
where {
    newPrefix = prefix+’_’+an;
    AttributeToColumn(tc, t, newPrefix);
}
}

Effect is that all the attributes of a class – and of any classes which are the types of attributes of that class – are to be columns in the same table. (?!)}
So, which kind of fixpoint?

It’s a matter of taste, ultimately – the OMG spec has nothing to say about it of course.

My feeling: if your opponent challenges you infinitely often, so that you have to respond infinitely often, it would be unfair if you lose because of it.

OTOH, if you cry foul infinitely often, maybe it’s because you can’t actually meet the challenge...

ModelMorf on the most trivial imaginable example actually works the other way round, though...
Recursion vs actual infinite plays

In most imaginable cases, plays will not be infinite even though the when/where graph has loops, because going round a loop will involve moving through a (finite) model in a way which can’t be continued for ever.

Not sure how to take advantage of this? Perhaps some neatly-stateable combination of properties of model transformation and metamodel can rule out all infinite plays – thereby making it immaterial whether someone agrees with my choice of fixpoint?
Remind me what we were doing this for?

**Non-plan:** expect explaining things in terms of mu calculus to aid adoption.

**Plan:** use the mu calculus translation to guide an improvement to my original QVT-R game (which does seem to get understood, in itself).

1. We don’t have to forbid infinite plays altogether. What kind of recursion do we have to forbid, though?
2. What should the winning conditions be for infinite plays?
What does non-monotonicity look like?

It looks like coming round to the same relation invocation infinitely often but with the players alternating roles.

Players switch roles on (a) explicit negation (b) relation invocations in *when* clauses.

There must be an even number of those in any loop.

Declare ill-formed any transformation not satisfying this.
Who should win an infinite play?

An infinite play must go through some relation invocation infinitely many times. Look at the play; of those invocations that play goes through infinitely often, which came first?

Verifier wins if this invocation was in a where clause and the players were the normal way round, or it was in a when clause and the players were swapped. Otherwise Refuter wins.

(It can’t be that the players are alternately the right way round and swapped, because that would be non-monotonic.)
Is there any more mileage?

Can we make deductions about

- complexity
- expressivity
- suitable semantics for enforce mode QVT

from this connection?

(Conjecture, JCB: alternation depth 2 is actually all we get)
Enforce mode

“Clearly impossible” in the general case (of anything sufficiently expressive and usable)

To do: identify some easy cases where it is clear what the semantics should be.

Find game paths where Refuter wins and fix them...

Cf problem of modifying a given LTS, “minimally”, so that a given HML/µ formula holds.

look at Dutch dynamic logic work? (CPS)

Probably not much mileage in metrics: what developers care about varies too much with circumstance.
Recall: a bidirection transformation specifies a relation.

We know how to compose relations. This is both a blessing and a curse.

\((T_1; T_2)(m, p)\) iff exists \(n\) s.t. \(T_1(m, n)\) and \(T_2(n, p)\)

Also obvious: how composition should behave on no-information model \(\Omega\), that want identity and associativity.

But composing the enforce functions requires “guessing” an element of the middle domain.
In lieu of impossibility proof: illustrative example

Let $S : M \times P$ and $T : P \times N$ be bidirectional transformations. Let $r : M \rightarrow P$ be a function. Then the composition of $S$ and $T$ with respect to $r$ is defined by:

$$(S; T)(m, n) \iff \exists p \in P : S(m, p) \land T(p, n)$$

$$\overrightarrow{S; T}(m, n) = n \quad \text{if } (S; T)(m, n) \quad \overrightarrow{T}(\overrightarrow{S}(m, r(m)), n) \quad \text{otherwise}$$

$$\overleftarrow{S; T}(m, n) = m \quad \text{if } (S; T)(m, n) \quad \overleftarrow{S}(m, \overleftarrow{T}(r(m), n)) \quad \text{otherwise}$$

Lots of good properties, but does not have identity in general :-( 
Cute (counter)example due to Meertens

Composition of computable bidirectional transformations need not be computable.

Consider bidirectional transformation $T$ between
$M$, the set of words over a given alphabet, and
$N$, the set of context-free grammars over the same alphabet
whose consistency relation is

$T(m, n)$ iff $m$ is in the language generated by $n$.

(It is easy to see that we can define computable forward and backward components to go with this consistency relation.)

Let $T^{op}$ be obtained in the obvious way by swapping the roles of the arguments, and consider $T^{op}; T$.

Under any definition of composition which is relational,
$(T^{op}; T)(n_1, n_2)$ should hold if and only if the intersection of the grammars $n_1$ and $n_2$ is non-empty.

But this is undecidable.
So what sequential composition can we do?

Given $T$ on $M \times N$ and invertible $f : M' \to M$ we can define $(f; T)$ on $M' \times N$ in the obvious way:

$$(f; T)(m, n) \iff T(f(m), n)$$

$$(f; T)(m, n) = \overrightarrow{T}(f(m), n)$$

$$(f; T)(m, n) = f^{-1}(\overleftarrow{T}(f(m), n))$$

Dually, we can define $T; g$ on $M \times N'$ where $g : N' \to N$. Maybe this is often enough in practice?
Engineering issues: high level

Model transformations are programs...

... but compared to most programs:

- they have very complex typical input and output (compared to their own complexity)
- they are not expected to be run often
- typically, there is strong structural similarity between inputs and outputs
- bidirectionality!
A model transformation [language] will not work correctly on complex inputs unless it works correctly on simple inputs. But what is “correctly” and how do we tell?
What properties are of interest?

Provenance, traceability, impact analysis.

If I change this, what may the knock-on effects be?

Why did the transformation change that bit?

If this is frozen, which parts of that are frozen?

Maybe, property preservation/transformation:

How do I ensure that this transformation maintains \( \phi \)?

(Simple example which is always assumed: if the input models conform to their metamodels, and the transformation is “well typed” then the output model will conform to its metamodel.)
QVT-R is an example of a bidirectional model transformation language in a particular style.

We have provided a semantics for QVT-R in checkonly mode, including for recursion where this makes sense.

We have discussed the problem of sequential composition.

**To do:** bring the enforce mode and composition stories to a reasonable conclusion.

**Maybe to do:** develop a theory of testing and verifying bidirectional model transformations

– all in a suitably simplified setting!
Extra slides
Users’ requirements

from 05-02-04 (very interesting but unfinished document)

- **Bidirectional transformations** so that they will be able to synchronize models interactively.
- **Ease of expression** so that they will be able to simply express what they want when using abstractions.
- **Interchangeable Transformation Models** so that they can share and use transformations in ways similar to other forms of digital media.
- **Customizing Transformations** so that they can repeatedly achieve the desired result in the target models.
Undoability

More controversial.

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

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

You modify $m$ to $m'$.
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 $m'$. 
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 $m$. 
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.
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*:

\[ T(m, n) \implies \overrightarrow{T}(m, \overrightarrow{T}(m', n)) = n \]

\[ T(m, n) \implies \overleftarrow{T}(\overleftarrow{T}(m, n'), n) = m \]
Why undoability may be too strong

Suppose the change you made was to delete some information from $m$ to get $m'$. When you applied the transformation, you deleted the “corresponding information” from $n$, yielding $n'$. But you also deleted any information which was “stuck” to that information in $n$, even if it wasn’t represented in $m$. So when you reverted to $m$, you restored all the information you knew about... 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.
Why undoability may be too strong

Suppose the change you made was to delete some information from $m$ to get $m'$. When you applied the transformation, you deleted the “corresponding information” from $n$, yielding $n'$. But you also deleted any information which was “stuck” to that information in $n$, even if it wasn’t represented in $m$. So when you reverted to $m$, you restored all the information you knew about, 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.
Why undoability may be too strong

Suppose the change you made was to delete some information from $m$ to get $m'$. When you applied the transformation, you deleted the “corresponding information” from $n$, yielding $n'$. 

But you also deleted any information which was “stuck” to that information in $n$, even if it wasn’t represented in $m$. 
Why undoability may be too strong

Suppose the change you made was to delete some information from $m$ to get $m'$. When you applied the transformation, you deleted the “corresponding information” from $n$, yielding $n'$. 

But you also deleted any information which was “stuck” to that information in $n$, even if it wasn’t represented in $m$.

So when you reverted to $m$, you restored all the information you knew about...
Why undoability may be too strong

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

When you applied the transformation, you deleted the “corresponding information” from $n$, yielding $n'$.

But you also deleted any information which was “stuck” to that information in $n$, even if it wasn’t represented in $m$.

So when you reverted to $m$, you restored all the information you knew about...

... 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.
Where’s the Galois connection?

Partially order any set of models by considering a model as a set of model elements and using $\subseteq$.

Define $(m, n)^\ast = (m, \rightarrow R (m, n))$

$(m, n)^\dagger = (\leftarrow R (m, n), n)$

Then we expect $(m, n) \subseteq ((m, n)^\ast)^\dagger$ and $(m, n) \subseteq ((m, n)^\dagger)$

However, with our single two-way consistency relation this is trivial – more interesting if we consider the two single-direction consistency relations separately.
Transformations on pairs of boolean model elements

transformation PwhereQ (m1 : BoolMM ; m2 : BoolMM) {
  top relation SameValue {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=i};
    checkonly domain m2 s2:ABoolean {value=i};
    where {FirstIsTrue(s1,s2);}
  }
}

relation FirstIsTrue {
  i : Boolean;
  checkonly domain m1 s1:ABoolean {value=true};
  checkonly domain m2 s2:ABoolean {value=i};
}

Mutatis mutandi, PwhenQ, QwhereP, QwhenP.

Apply each transformation to each pair of models ((T,T), (T,F), (F,T), (F,F)), in each direction. Compare our semantics with ModelMorf.
Reminder: P is SameValue, Q is FirstIsTrue. The invoked relation is not top.

<table>
<thead>
<tr>
<th></th>
<th>(T,T)</th>
<th>(T,F)</th>
<th>(F,T)</th>
<th>(F,F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P where Q</td>
<td>V</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>P when Q</td>
<td>V</td>
<td>R</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Q where P</td>
<td>V</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Q when P</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>R</td>
</tr>
</tbody>
</table>

Our semantics and ModelMorf agree, hooray!
In direction of m2

Reminder: P is SameValue, Q is FirstIsTrue. The invoked relation is not top.

<table>
<thead>
<tr>
<th></th>
<th>(T,T)</th>
<th>(T,F)</th>
<th>(F,T)</th>
<th>(F,F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PwhereQ</td>
<td>V</td>
<td>R</td>
<td>R</td>
<td>V</td>
</tr>
<tr>
<td>PwhenQ</td>
<td>V</td>
<td>R</td>
<td>R</td>
<td>V</td>
</tr>
<tr>
<td>QwhereP</td>
<td>V</td>
<td>R</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>QwhenP</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

Our semantics and ModelMorf agree except on one point.
P when Q on (F,T) in direction →

- Refuter challenges in SameValue with F.
- Verifier can’t match, so she’d like to challenge the *when* clause, FirstIsTrue.
- But to do that, she must find a valid binding of `value` in the F domain, i.e., satisfying the local constraint `value = true`.
- She can’t, so she has no legal move, so Refuter wins.

Instantiating the QVT Ch 7 definition, should be true iff

\[
\text{FirstIsTrue}(s_1,s_2) \Rightarrow s_1.\text{value} = s_2.\text{value}
\]

But what does `FirstIsTrue(s_1,s_2)` mean? Standard doesn’t say. I say: for all valid completions of `s_1`... there exists a valid completion of `s_2`...

(\textit{Could} change the game so that choosing bindings is optional for the challenger: that would save Verifier here, but cause other problems.)