

# Traceability in (bidirectional) model transformations

Perdita Stevens

School of Informatics  
University of Edinburgh

# Traceability

The ability to trace the influence of one software engineering artifact, or part of an artifact, on another, in order to

- ▶ understand the system (“why’s that like that?”)
- ▶ analyse the impact of (proposed) changes to an artefact (“what might break if we change that?”)
- ▶ debug (“what could be causing THAT?!”)
- ▶ communicate (“who should know about that?”)

Evidently this *is* provenance.

Establishing the Connection between Traceability and Data Provenance, Hazeline Assuncion and Richard N. Taylor, UC Irvine TR UCI-ISR-07-09

So...?

# The Traceability Benefit Problem\*

Ensuring traceability is a duty honour'd in the breach.

# The Traceability Benefit Problem\*

Ensuring traceability is a duty honour'd in the breach.

Mountains of research work, over decades, lots of techniques.

# The Traceability Benefit Problem\*

Ensuring traceability is a duty honour'd in the breach.

Mountains of research work, over decades, lots of techniques.

Sometimes mandated; but in many projects, hardly done.

# The Traceability Benefit Problem\*

Ensuring traceability is a duty honour'd in the breach.

Mountains of research work, over decades, lots of techniques.

Sometimes mandated; but in many projects, hardly done.

Why? The cost-benefit comparison doesn't favour it.

Term ?coined by Arkley and Riddle, *Overcoming the TBP*, ICRE'05

## So what's the problem?

The **cost** of collecting and maintaining traceability information is high because of

- ▶ CHANGE, above all; and:
- ▶ different people and groups controlling different artefacts;
- ▶ tool and semantic incompatibility.

The **benefit** is often lower than hoped, e.g. because of poor quality or missing information.

More subtly, sometimes it's clear there are easier ways to achieve the same benefit.

## So what's the problem?

The **cost** of collecting and maintaining traceability information is high because of

- ▶ CHANGE, above all; and:
- ▶ different people and groups controlling different artefacts;
- ▶ tool and semantic incompatibility.

The **benefit** is often lower than hoped, e.g. because of poor quality or missing information.

More subtly, sometimes it's clear there are easier ways to achieve the same benefit.

**In theory**, developing and maintaining traceability links ought to be worth the cost;

**in practice**, it isn't.

## And provenance?

**Glass half full:** maybe using provenance techniques can solve the traceability benefit problem?

## And provenance?

**Glass half full:** maybe using provenance techniques can solve the traceability benefit problem?

**Glass half empty:** maybe traceability is just much harder?

## And provenance?

**Glass half full:** maybe using provenance techniques can solve the traceability benefit problem?

**Glass half empty:** maybe traceability is just much harder?

**Glass being drunk:** maybe in ten years' time people will be writing about the **Provenance Benefit Problem**?

# What's provenance got that traceability hasn't?

At least data provenance, as studied in the database world:

# What's provenance got that traceability hasn't?

At least data provenance, as studied in the database world:

automation

# What's provenance got that traceability hasn't?

At least data provenance, as studied in the database world:

**automation**

Much of the Traceability Benefit Problem derives from the cost and unreliability of **manually** maintaining the information.

If this can be automated, problem solved?

Enter **Model-Driven Development**.

# Models

For purposes of this talk:

A model is any [formal] artefact relating to a software system.

E.g. a diagrammatic abstraction of the design.

That is, a graph of connected **model elements**, which conforms to a set of rules embodied in a **metamodel**.

# Model-driven development

Idea: let models be more than suggestive pictures. Be able to

- ▶ depend on them
- ▶ check their consistency
- ▶ derive one from another
- ▶ generate code, tests, documentation from them

etc.

E.g.: round-trip engineering between UML model and code. NB in practice, need not just generation but synchronisation.

Key necessity: **model transformations**

# Model transformation languages

Confused landscape right now! E.g.

- graph transformation based languages, e.g. triple graph grammars
- OMG's Query Views and Transformations languages (QVT Relations, QVT Operations, QVT Core)
- template-based languages
- or just use your favourite programming language.

See e.g. *A landscape of bidirectional model transformations*, S., Post-proc GTTSE'07

# QVT Relations

OMG standard language, pretty much finalised in 2005.

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

Patchwork of relations, connected by when and where clauses.

Exactly two tools, ever: Medini QVT (open source engine);  
ModelMorf (defunct)

“Implicitly” builds trace information (more anon...)

## A single patch

```
    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 one-patch patchwork

```
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**

$$\text{thing1.value} = \text{thing2.value}$$

# Trace objects in QVT languages

Fix a transformation.

This gives a trace class per relation, with attributes for pattern variables.

## Trace objects in QVT languages

Fix a transformation.

This gives a trace class per relation, with attributes for pattern variables.

Now a *trace model* connects consistent models m1 and m2.

It consists of objects of the trace classes.

Each trace object connects a specific binding of a pattern in m1 to a *corresponding* binding of the corresponding pattern in m2.

## Role of the trace model in QVT languages

Main motivation: efficiency of updating a large target model when a small part of the source model changes. Avoid recalculating the whole thing.

“Implicit” in QVT-R; explicit in QVT-Op and QVT-Core.

A transformation execution creates a set of trace class instances, demonstrating that the relations hold.

(Which they will, after the transformation has finished, anyway – from now on we only consider checking consistency of two models, not modifying them).

# The trace model and the traceability questions

Suppose we have a source model, a trace model, and a target model with a problem. Will the trace model help trace the source of the problem?

- ▶ understand the system (“why’s that like that?”)
  - sort of. It may at least point you at the right part of the source model to study.
- ▶ analyse the impact of (proposed) changes to an artefact (“what might break if we change that?”)
  - sort of, provided it’s the source model that changes, and the change is simple.
- ▶ debug (“what could be causing THAT?!”)
  - yes, to a limited extent: the source side of the trace object gives proximate cause
- ▶ communicate (“who should know about that?”)
  - not really within scope

# The trace model and the traceability questions

Suppose we have a source model, a trace model, and a target model with a problem. Will the trace model help trace the source of the problem?

- ▶ understand the system (“why’s that like that?”)
  - sort of. It may at least point you at the right part of the source model to study.
- ▶ analyse the impact of (proposed) changes to an artefact (“what might break if we change that?”)
  - sort of, provided it’s the source model that changes, and the change is simple.
- ▶ debug (“what could be causing THAT?!”)
  - yes, to a limited extent: the source side of the trace object gives proximate cause
- ▶ communicate (“who should know about that?”)
  - not really within scope

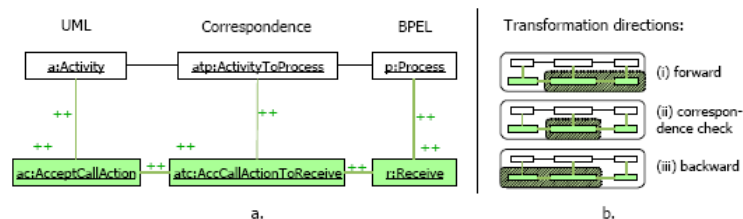
OK, so that’s clear as mud, let’s look at something else.

# Triple graph grammars

Development from earlier graph grammar work, by Andy Schürr.  
Unlike QVT, TGGs have solid formal basis.

A graph grammar consists of rules.

E.g.



Picture from *Applying Triple Graph Grammars For Pattern-Based Workflow Model Transformations*, Lohmann, Greenyer, Jiang and Systä, JOT vol 6 no 9

A pair of graphs, and the correspondence graph, are built up in parallel.

Ambiguity is to be avoided.

## TGG correspondence graph and provenance

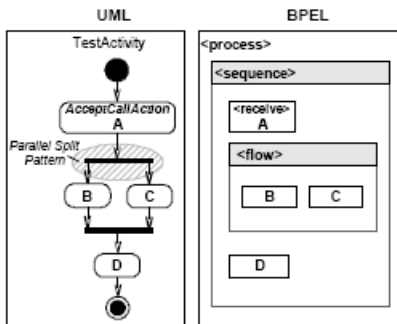
By construction, a correspondence node can be read in both directions: the linked nodes in the LH graph are in some sense related by the TGG to the nodes in the RH graph.

In what sense exactly? *Given the full context*, the existence of either the LH or the RH nodes is enough to justify the creation of the others, by applying the rule.

But reasoning locally about TGGs is hard.

## Typical TGG application scenario

In practice, examples where people use TGGs involve very close structural correspondence between the models.



E.g.

Picture *ibid.*

Such situations are common and important – but not the source of most traceability problems.

How broadly are TGGs applicable?

## Back to QVT-R: a new approach

Let us reformulate what trace objects do, in the hope of getting a better handle on how they do, and do not, contribute to traceability.

We have to delve a little deeper into the QVT-R language...

More detail on the next section in *A simple game-theoretic approach to checkonly QVT Relations*, S., accepted for ICMT'09

## Invoking relations: where clauses (omitting *when* clauses)

“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.”

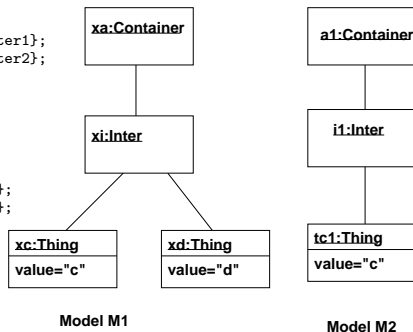
```
relation ClassToTable
{
  domain uml c:Class { ... stuff ...}
  domain rdbms t:Table { ... stuff ... }
  where { AttributeToColumn(c, t); }
}
```

# 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};
  }
}
```



## Same thing 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 a way of checking conformance to metamodel and “local” checking inside relations.

## Same thing as a game

Take:

- ▶ a pair of metamodels
- ▶ a QVT-R transformation;
- ▶ models  $m_1$  and  $m_2$  conforming to the metamodels.

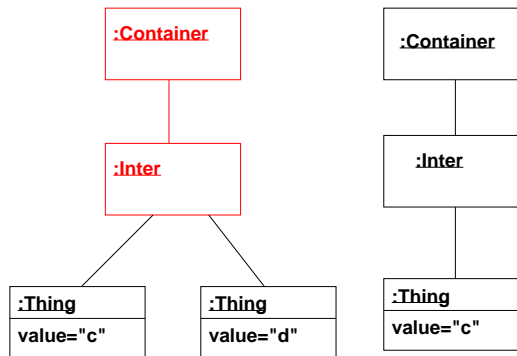
Assume we have a way of checking conformance to metamodel and “local” checking inside relations.

Let’s define game  $G$  to check in the direction of model  $m_2$ .

Two players, Verifier who wants the check to succeed, Refuter who wants it to fail.

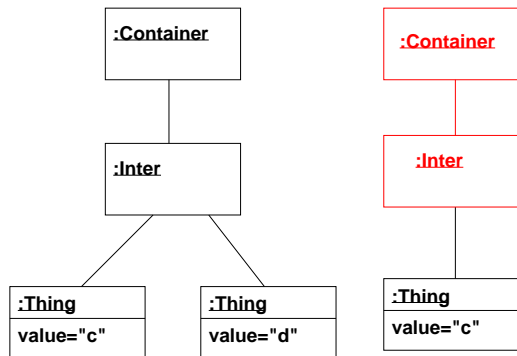
# Refuter

```
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);}
}
```



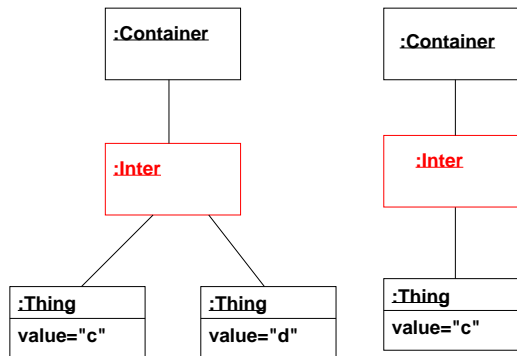
# Refuter; Verifier

```
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);}
}
```



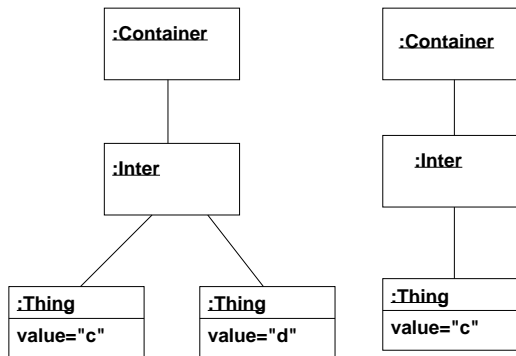
# Refuter;Verifier;Refuter

```
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);}
}
```



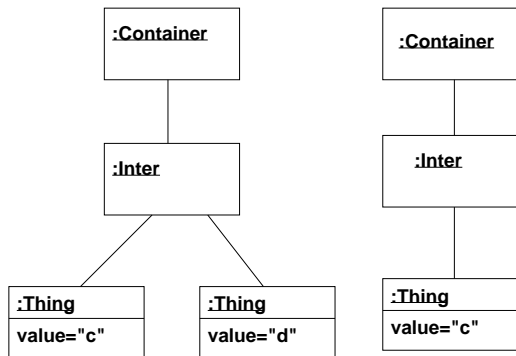
# Refuter;Verifier;Refuter

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



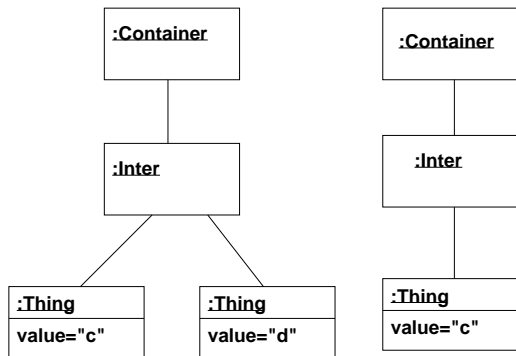
# Refuter;Verifier;Refuter;Verifier

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



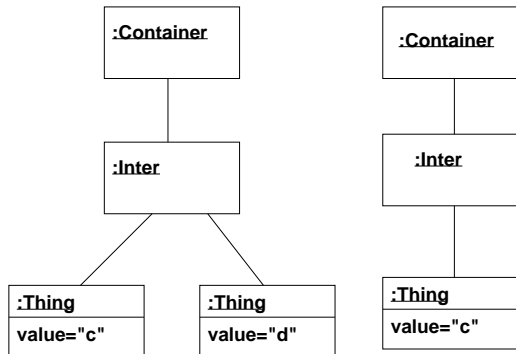
# Refuter;Verifier;Refuter;Verifier ;Refuter

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



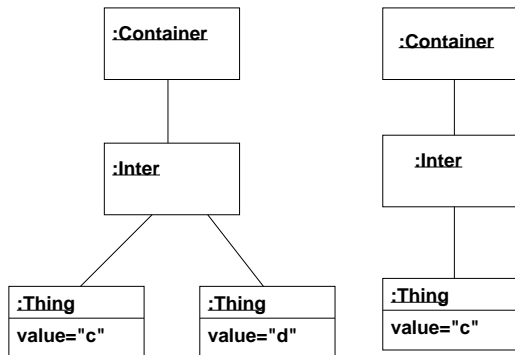
# Refuter;Verifier;Refuter;Verifier;Refuter

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



# Refuter; Verifier; Refuter; Verifier; Refuter; VERIFIER LOSES!

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



## Summary of moves (missing out *when*)

Position	Next position	Notes
Initial (Refuter to move)	(Verifier, $R, B$ )	$R$ is any top relation; $B$ comprises valid bindings for all variables from m1 domain
(Verifier, $R, B$ )	(Refuter, $R, B'$ )	$B'$ comprises $B$ together with bindings for any unbound m2 variables.
(Refuter, $R, B$ )	(Verifier, $T, D$ )	$T$ is any relation invocation from the where clause of $R$ ; $D$ comprises $B$ 's bindings for the root variables of patterns in $T$ , together with valid bindings for all m1 variables in $T$ .

You win if your opponent can't go. Let's not talk about infinite plays today

## Trace objects and the game

Verifier has a winning strategy iff the two models are consistent in the direction considered (e.g., the target was correctly produced from the source).

What does such a winning strategy look like?

## Trace objects and the game

Verifier has a winning strategy iff the two models are consistent in the direction considered (e.g., the target was correctly produced from the source).

What does such a winning strategy look like?

Like a set of trace objects.

Issue: how close is the connection between the “patchwork” structure of the transformation, the strategy and the models?

# Bidirectionality

So far we've only hinted at the issues raised by bidirectionality.

What do we mean by it, anyway?

A bidirectional model transformation must be able to propagate changes to *either* model.

Examples:

- ▶ requirements  $\longleftrightarrow$  tests
- ▶ platform independent model  $\longleftrightarrow$  platform specific model  $\longleftrightarrow$  source code
- ▶ database instance  $\longleftrightarrow$  modifiable view

Non-examples:

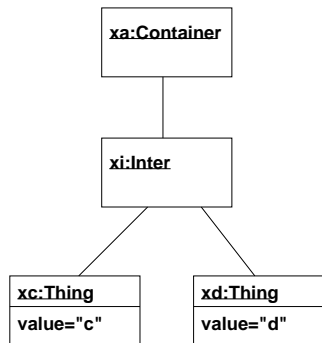
- ▶ source code  $\longrightarrow$  binary
- ▶ database  $\longrightarrow$  read-only view

## Bidirectionality and TGGs and QVT

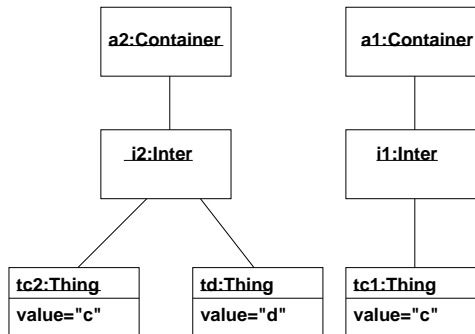
TGGs really are inherently symmetric: correspondence nodes have no inherent direction.

QVT, even QVT-R, is different. Trace objects trace what a transformation did, and it did it *in one direction*.

# Consistent, both ways

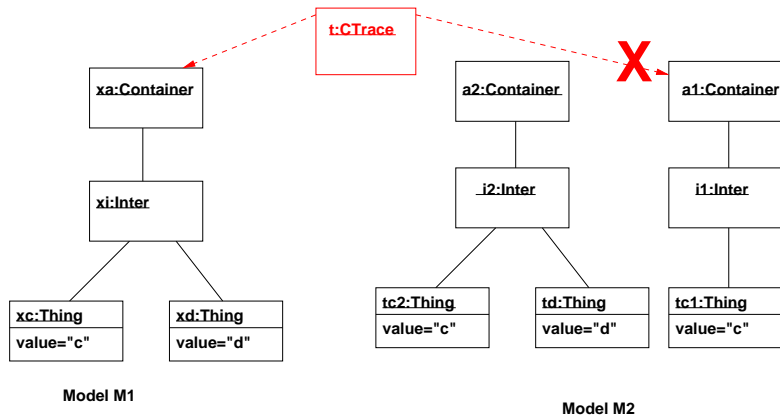


Model M1



Model M2

## But with no bidirectional trace objects



## What if we fiddle the game?

Let the player who's choosing bindings also choose which domain to choose them from. Then the other player has to match from the other domain.

Refuter then has a winning strategy for the previous example...

## What if we fiddle the game?

Let the player who's choosing bindings also choose which domain to choose them from. Then the other player has to match from the other domain.

Refuter then has a winning strategy for the previous example...

Probably there are bidirectional trace objects, now?

NB This is definitely not doing what it says in the QVT spec – but maybe it's DWIM?

## Quis custodiet ipsas transformationes?

It's all very well (maybe) to use formal model transformations to track relationships between models, and their trace models to store provenance information.

But the transformations themselves now become crucial artefacts in the software engineering process. How do we make decisions concerning *them* properly traceable?

This is probably an issue with no equivalent in the eScience provenance world: data provenance of your terabytes of data seems incomparable with workflow provenance of what was done with it.

But in software engineering, there is no such separation of data and process. The transformations can easily be as complex, and as prone to change, as the models they transform.

# Conclusion

Traceability is a hoary old problem.

Provenance is a fashionable new solution.

In model transformations specifically, there may be benefit in combining ideas.

However, it won't be trivial!