Model-driven development, traceability and games

Perdita Stevens

May 2009

Welcome!

Relevant courses (not pre-requisites!) include:

- Software engineering with objects and components
- Programming Methodology
- Software System Design
- Communication and Concurrency

Models

For purposes of this talk:

A model is any artefact relating to a software system.

E.g.

- any diagrammatic abstraction of the design, e.g. in UML
- a set of requirements in stylised "numbered sentence" form
- the source code
- a test suite

Models

For purposes of this talk:

A model is any formalisable artefact relating to a software system.

E.g.

- any diagrammatic abstraction of the design, e.g. in UML
- a set of requirements in stylised "numbered sentence" form
- the source code
- a test suite

A model can often be formalised as a graph of connected model elements, which conforms to a set of rules embodied in a metamodel.

How are models used?

(Thinking principally about diagrammatic abstractions for now)

Principally: for communication between humans.

The process of developing a model helps clarify thought about the system.

Giving someone a model can help them understand the system or its development.

If a model is still up to date, it can help maintainers.

The trouble with models

They are necessary, but expensive to develop and keep up to date.

Two reactions are possible:

1. reduce cost: the agile approach. Model on a whiteboard, and only for a present purpose.
2. increase benefit: the modelling approach. Use well-defined modelling languages; use models throughout development; generate code and documentation from them; verify them; etc.

Ideally, reducing cost and increasing benefit are not mutually exclusive.

As soon as you have any long-lived model, traceability becomes important.

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?")

Classic traceability

requirements $\longleftrightarrow$ code $\longleftrightarrow$ tests

Traditional approach: traceability matrix, e.g.

contributes to	n1	n2	n3	n4	n5
m1	X				
m2	X				
m3	X		X	X	X
m4		X			

Mature tool support, e.g. DOORS.

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.

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.

Imagine...

You could take the source code and functional requirements specification of an implemented software system,

delete and change (and add?!) some requirements

and have the source code automatically modified to suit.

Imagine...

You could take the source code and functional requirements specification of an implemented software system,

delete and change (and add?!) some requirements

and have the source code automatically modified to suit.

You can't, of course, but with intermediate, more closely related models, e.g. platform independent model and platform specific model, this is the idea of MDD.

Model-driven development

Model-driven development (aka model-driven architecture), very fashionable, pushed by Object Management Group members.

Idea is to get much, much more benefit from modelling: ultimately, eliminate some activities, e.g. coding, completely.

MDD and traceability

MDD is often sold by the "no more coding" side, but maybe the traceability benefit is actually more important/more realistic.

That is, even if we still have to do hard work on each model, it's possible MDD would be worthwhile just for the traceability win.

Even if your tool could only check consistency, this might already be a big win if it did it well enough.

Model transformations specify the connections between models.

Model transformations may

Generate a model:

– take one model, return another

Check consistency of two models:

– take two (or more) models, return true/false (and trace information?)

Update a model:

– take a human-modified model and another it's supposed to be consistent with, (and ???) and return a modified form of the second.

(Also e.g. diffing, merging, synchronisation of several models.)

Need languages to express the transformations.

Model transformation languages

Of course you *can* write a model transformation in any language, e.g. Java.

Problems include:

- hard to write concise, readable transformations in a general-purpose language;
- for a bidirectional transformations, you'd have to write two programs and keep them consistent by hand – nightmare.

So many special-purpose transformation languages have been developed.

QVT Relations

OMG standard language, pretty much finalised in 2005.

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

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

QVT Relations

OMG standard language, pretty much finalised in 2005.

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

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

Sufficient problem:
this kind of language really needs clear semantics!

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

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

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."

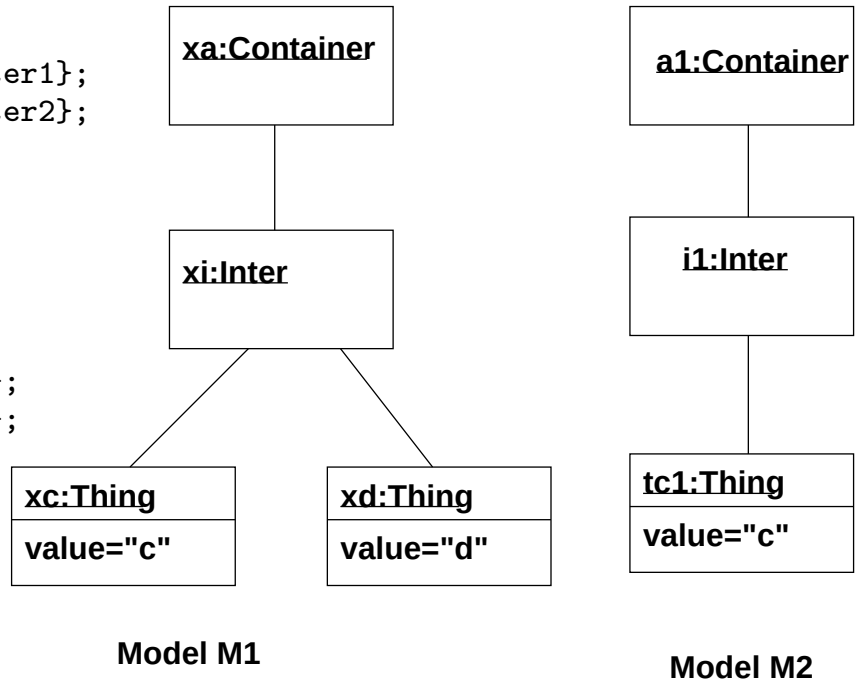
```
relation ClassToTable
{
  domain uml c:Class { ... stuff ...}
  domain rdbms t:Table { ... stuff ... }
  where { AttributeToColumn(c, t); }
}
```

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};
  }
}
```



What does this remind us of?

A process $B = (S_B, i_B, L_B, \rightarrow_B)$ is said to simulate a process $A = (S_A, i_A, L_A, \rightarrow_A)$ if there exists a simulation relation $\mathcal{S} \subseteq S_A \times S_B$ containing (i_A, i_B).

The condition for the relation to be a simulation relation is the following:

$$(s,t) \in \mathcal{S} \Rightarrow (\forall a, s' \ .(s \xrightarrow{a} s' \Rightarrow \exists t' \ . \ t \xrightarrow{a} t' \wedge (s', t') \in \mathcal{S}))$$

and this is easily reformulated as a two-player game.

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 a way of checking conformance to metamodel and "local" checking inside relations.

QVT Relations checking as a game

Take:

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

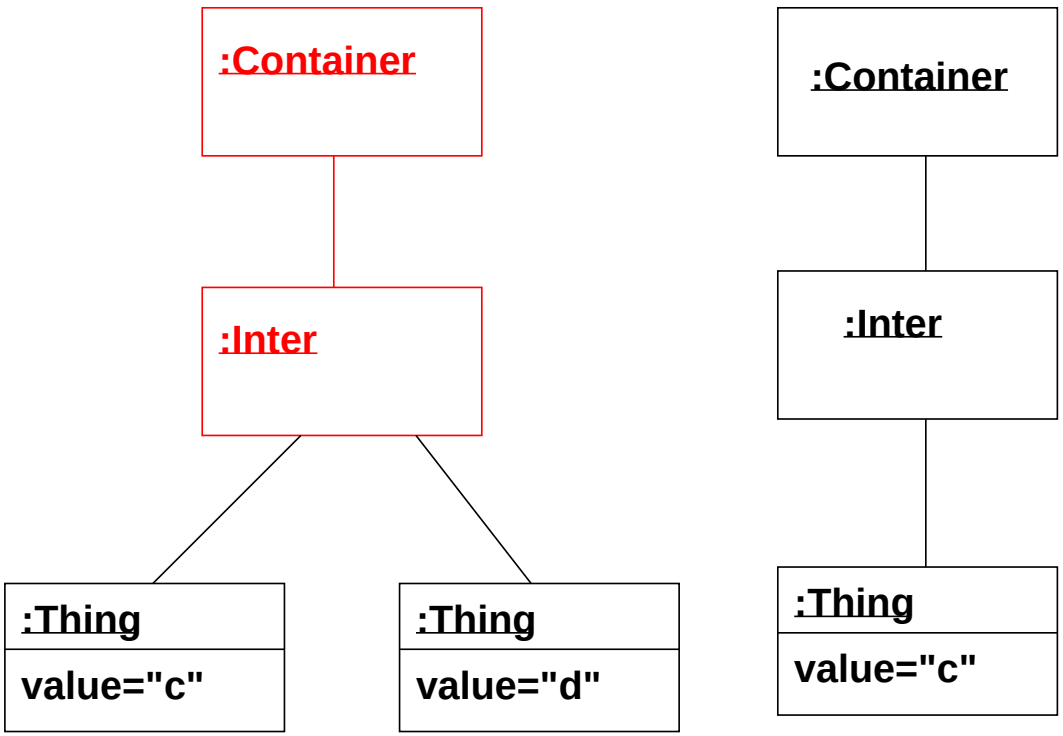
Assume we have a way of checking conformance to metamodel and "local" checking inside relations.

Let's define game G to check in the direction of model m2.

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

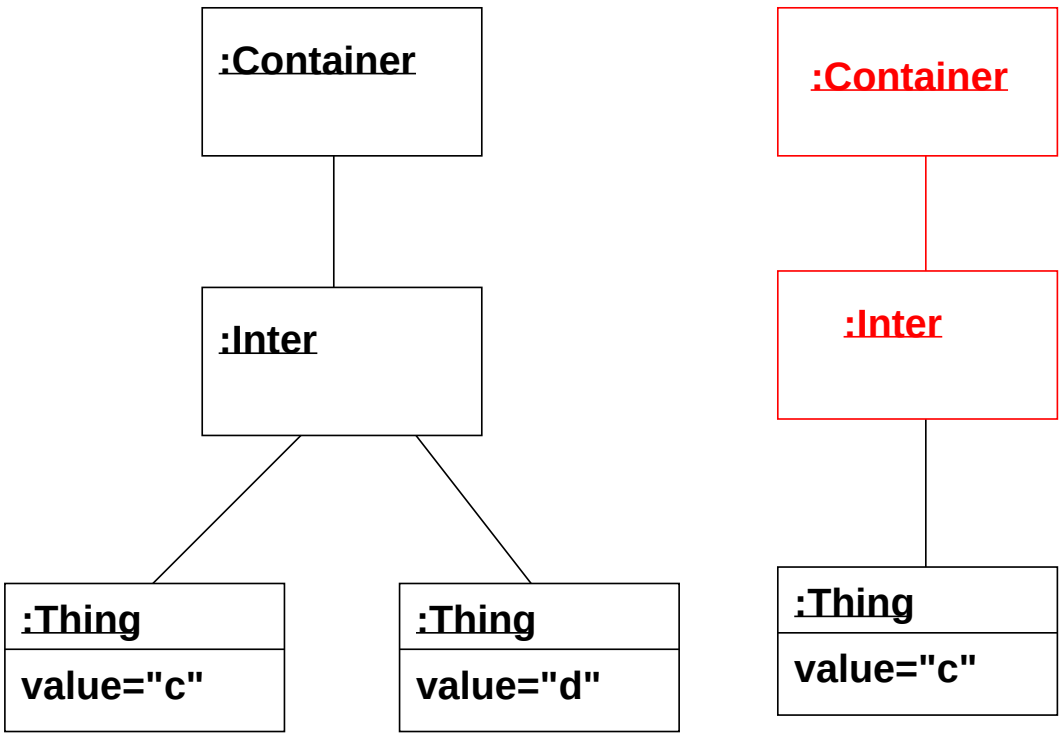
Refuter

```
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);}
}
```



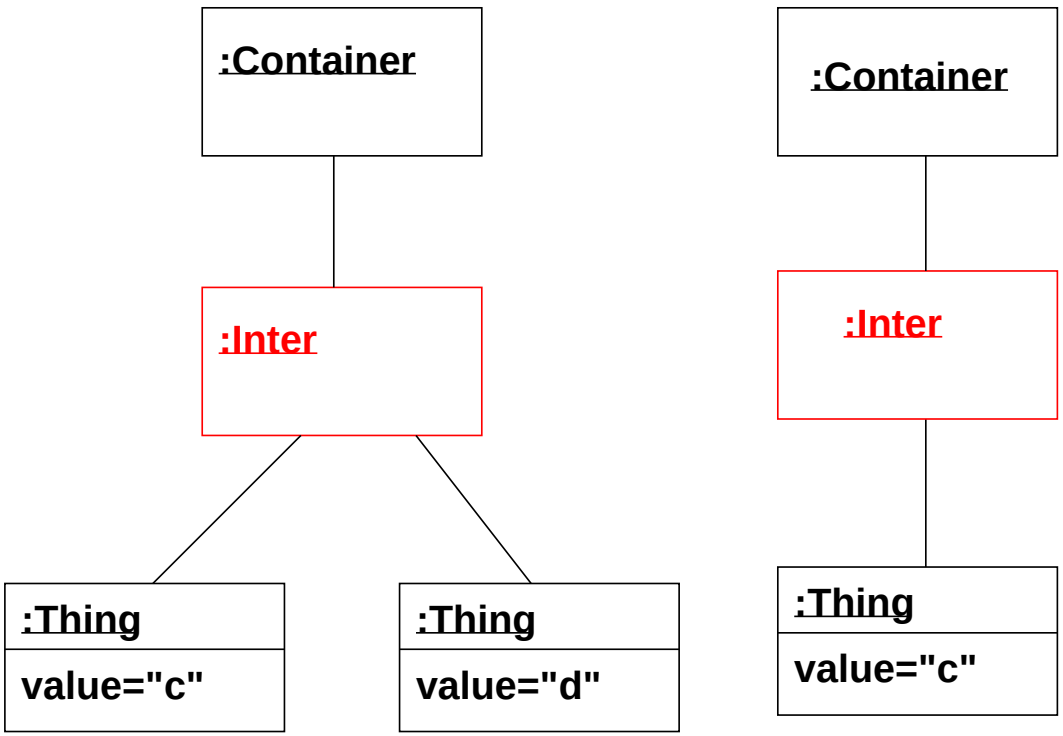
Refuter;Verifier

```
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);}
}
```



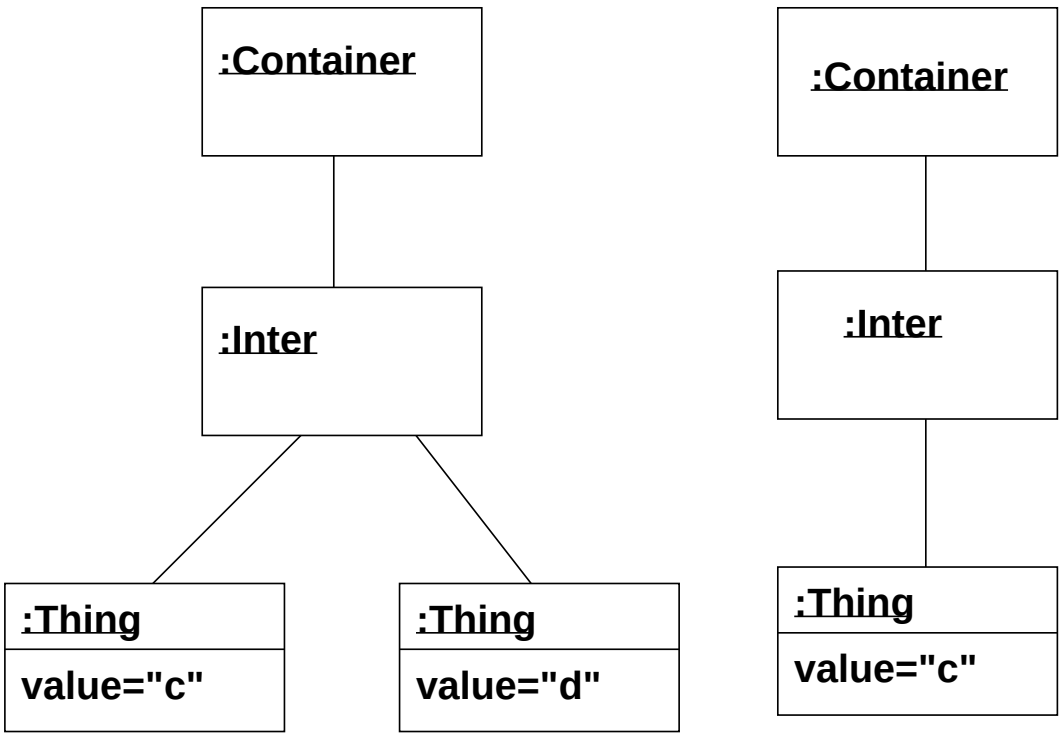
Refuter;Verifier;Refuter

```
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);}
}
```



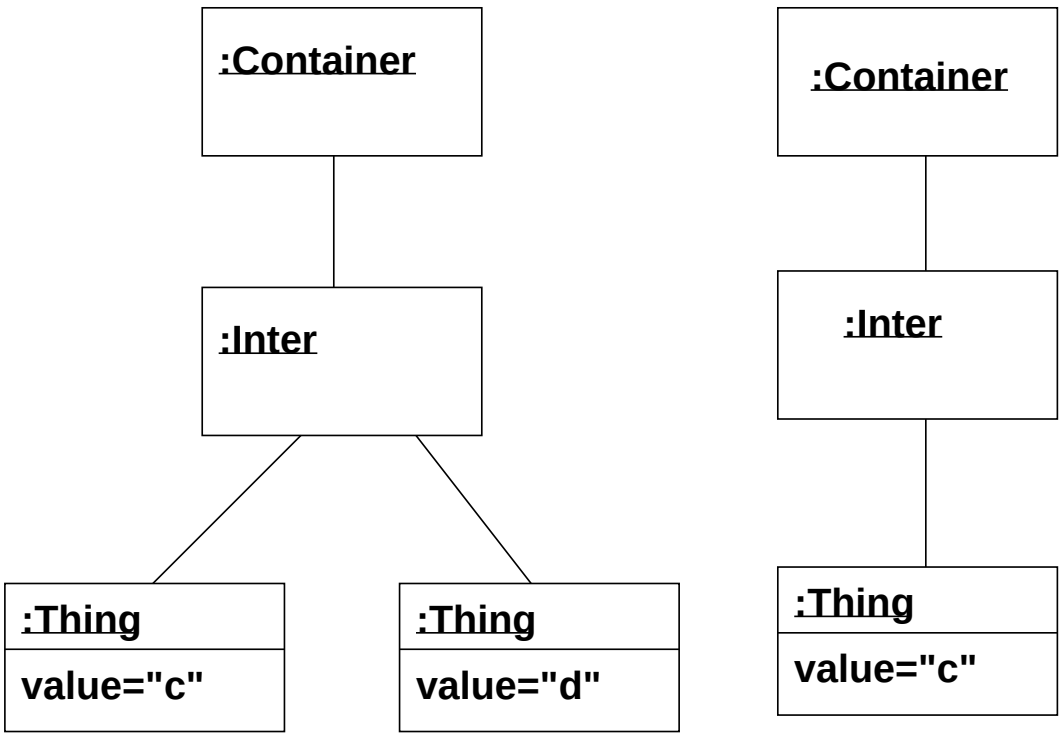
Refuter;Verifier;Refuter

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



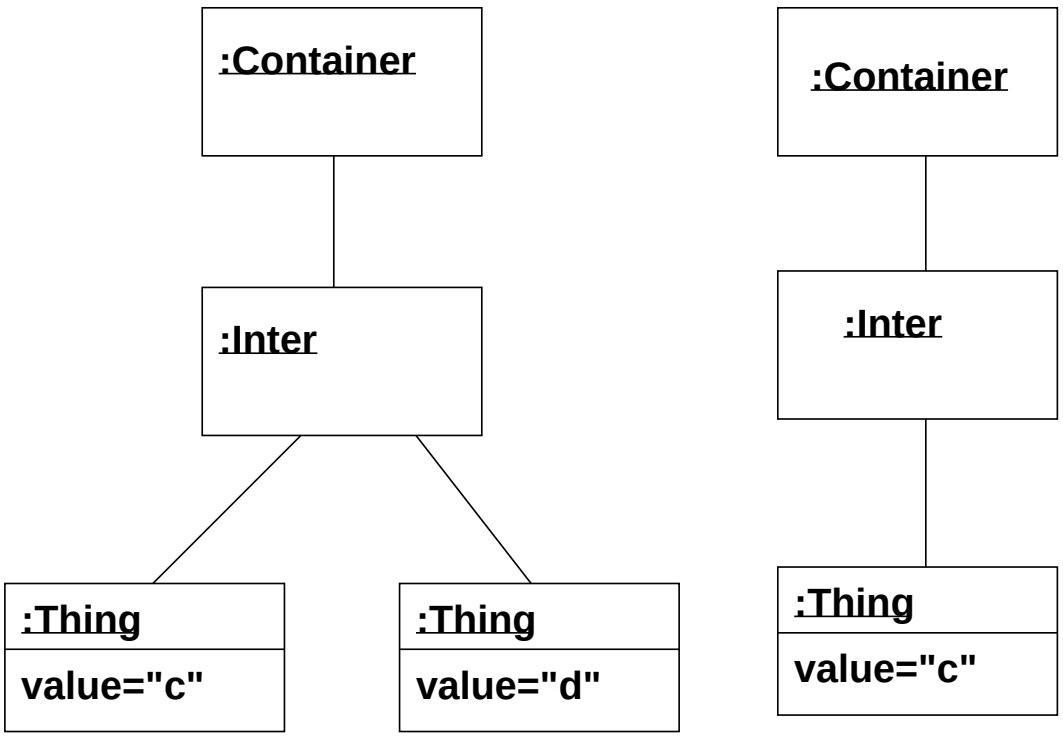
Refuter;Verifier;Refuter;Verifier

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



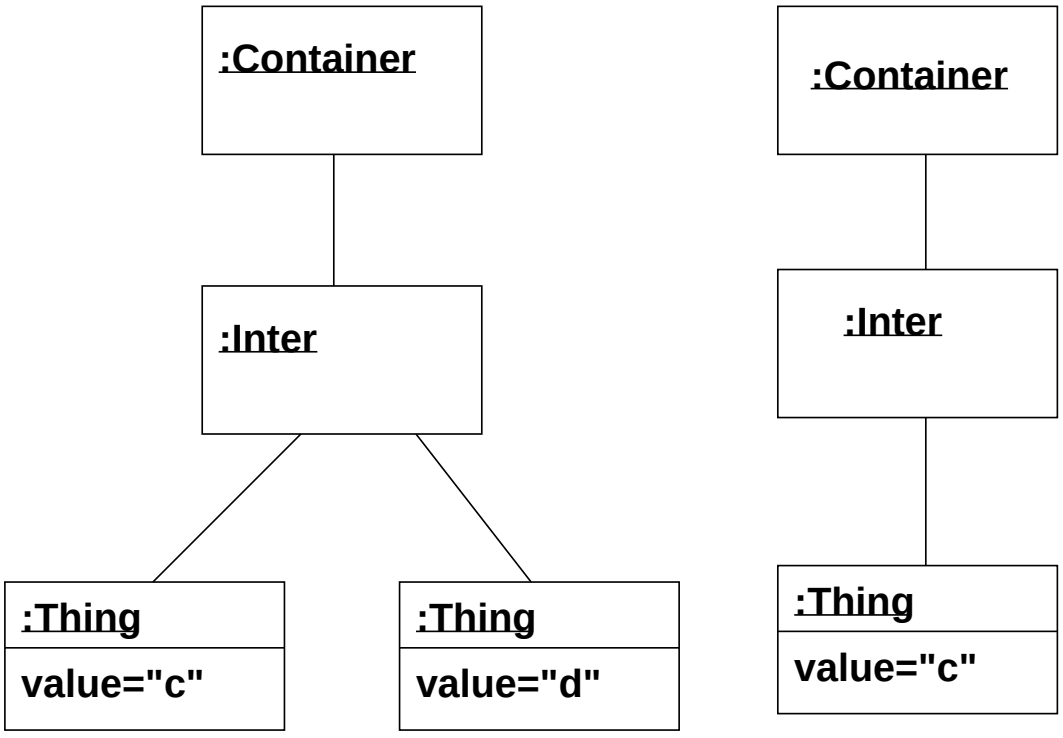
Refuter;Verifier;Refuter;Verifier ;Refuter

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



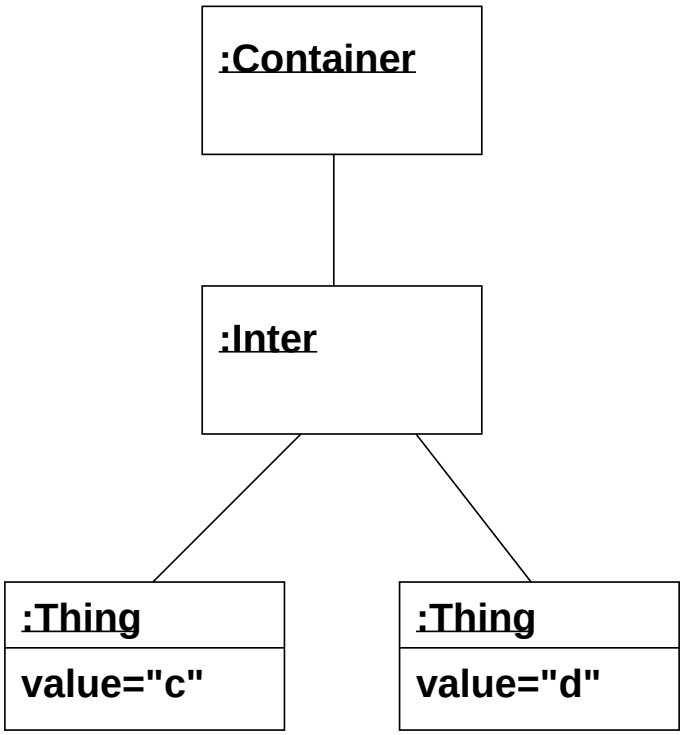
Refuter;Verifier;Refuter;Verifier;Refuter

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

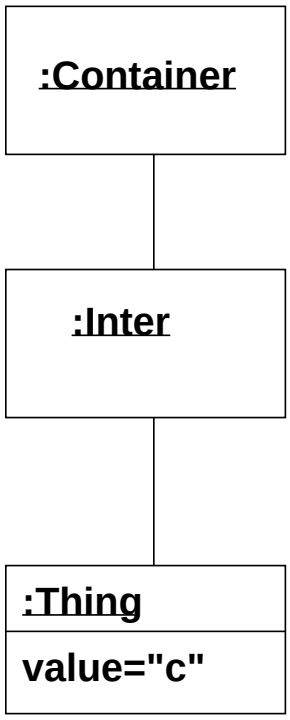


Refuter;Verifier;Refuter;Verifier;Refuter;VERIFIER LOSES!

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



Model M1



Model M2

Summary of moves (missing out *when*)

Position	Next position	Notes
Initial (Ref.)	(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.

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 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 Verifier to "counter-challenge" a when-clause... players swap roles...

(see paper for details)

Winning conditions

As always, you win if your opponent can't go.

We saw Verifier unable to pick a matching valid binding.

In another play, Refuter might have had no `where` relation to pick.

Winning conditions

As always, you win if your opponent can't go.

We saw Verifier unable to pick a matching valid binding.

In another play, Refuter might have had no `where` relation to pick.

But what about infinite plays?

Winning conditions

As always, you win if your opponent can't go.

We saw Verifier unable to pick a matching valid binding.

In another play, Refuter might have had no `where` relation to pick.

But what about infinite plays?

Could just forbid: insist graph of relations with when-where edges be a DAG.

Or should Verifier (rsp. Refuter) lose a play that goes infinitely often through when (rsp. where) clauses (only)? Etc.?

Winning conditions

As always, you win if your opponent can't go.

We saw Verifier unable to pick a matching valid binding.

In another play, Refuter might have had no `where` relation to pick.

But what about infinite plays?

Could just forbid: insist graph of relations with when-where edges be a DAG.

Or should Verifier (rsp. Refuter) lose a play that goes infinitely often through when (rsp. where) clauses (only)? Etc.?

NB QVT spec doesn't address the issue at all – corresponds to infinite regress of its definitions.

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.

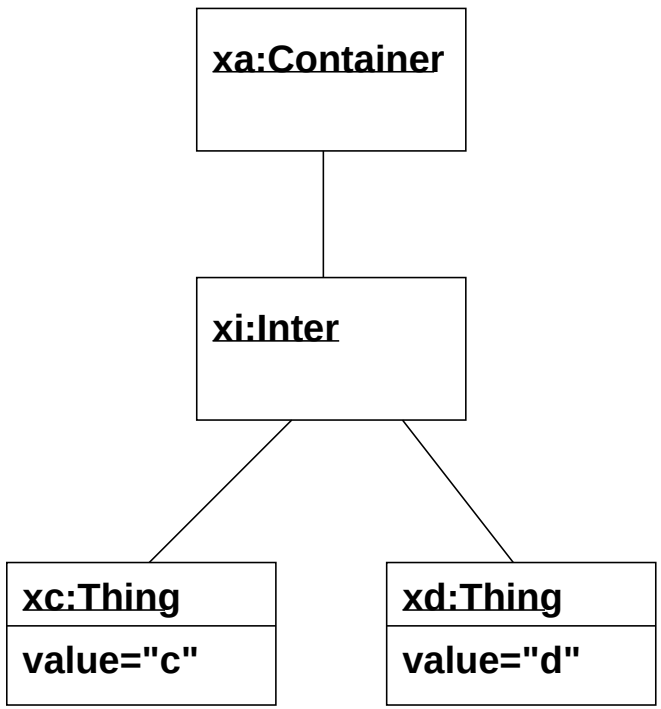
Examples:

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

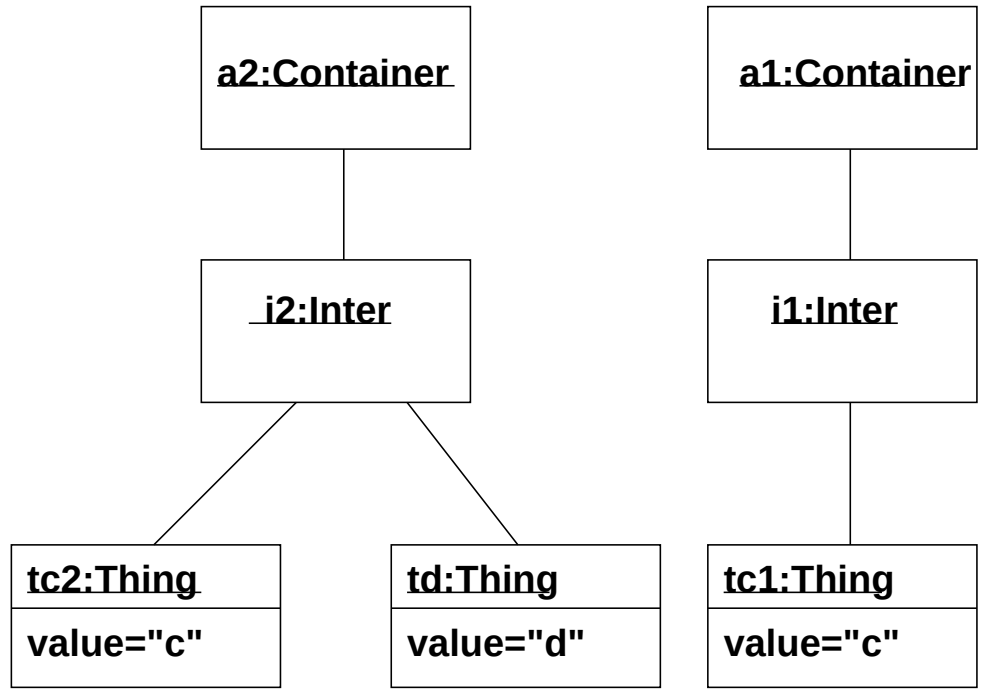
Non-examples:

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

Consistent, both ways

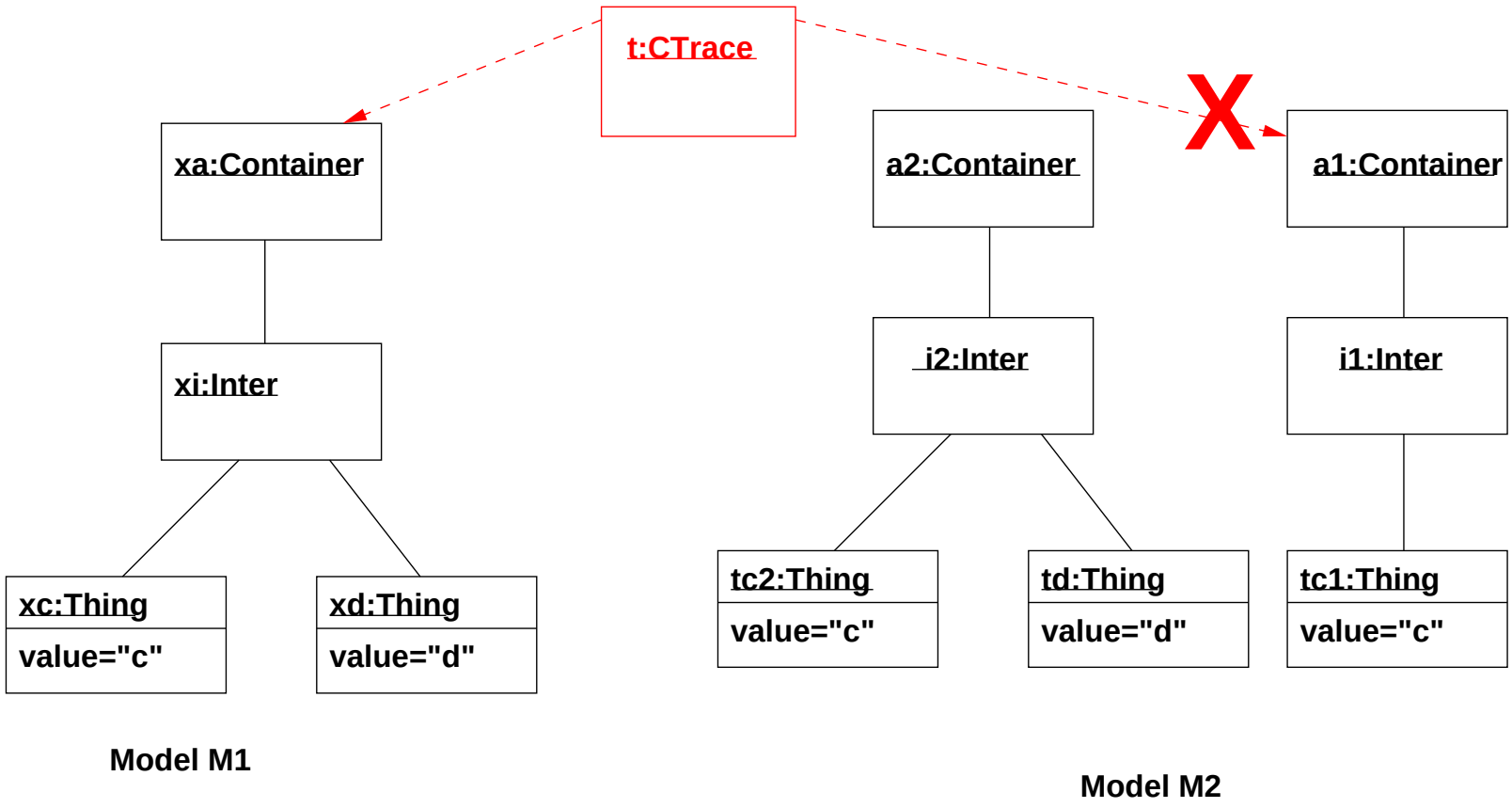


Model M1



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?

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

Bisimulation

A process $B = (S_B, i_B, L_B, \rightarrow_B)$ is said to bisimulate a process $A = (S_A, i_A, L_A, \rightarrow_A)$ if there exists a bisimulation relation $\mathcal{S} \subseteq S_A \times S_B$ containing (i_A, i_B).

The condition for the relation to be a bisimulation relation is the following:

$(s, t) \in \mathcal{S} \Rightarrow$

$((\forall a, s' \,.(s \xrightarrow{a} s' \Rightarrow \exists t' \,.\, t \xrightarrow{a} t' \wedge (s', t') \in \mathcal{S})) \qquad \wedge$

$(\forall a, t' \,.(t \xrightarrow{a} t' \Rightarrow \exists s' \,.\, s \xrightarrow{a} s' \wedge (s', t') \in \mathcal{S})))$

Exactly analogous difference in the game: allow Refuter to choose which process to challenge from, making a new choice each time.

Conclusion and ongoing work

A simple game, can help explain what QVT-R means...

... but also exposes false assumptions.

How does the community react to the bisimulation-based interpretation of QVT-R?

Can this also help explain enforcing transformations, i.e., how inconsistencies between models should be corrected?