A simple game-theoretic approach to checkonly QVT Relations

Perdita Stevens

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

Thank to ITI Scotland Ltd. for funding my attendance at ICMT

June 2009

Plan

- Background: QVT-R and its definitions
- Inconsistency of the definitions
- Game definition, formalising standard direct definition
- Consequence: non-existence of bidirectional trace objects
- Possible variant semantics and other ongoing work

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 little tool support: nothing industrial-strength and standards-compliant.

Medini QVT (ModelMorf, MOMENT-QVT)

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 little tool support: nothing industrial-strength and standards-compliant.

Medini QVT (ModelMorf, MOMENT-QVT)

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.

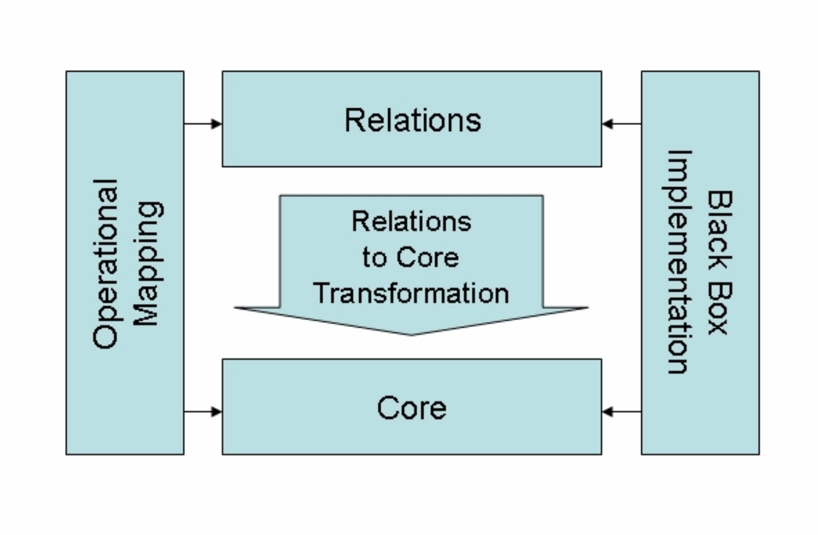
You run the transformation:

- in a direction: examine one model, regarding other(s) as authoritative
- in
  - checkonly mode: is `m2` OK according to authoritative `m2`? 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.

The paper concerns checkonly.

All the QVT languages



How the semantics of QVT-R is defined

The spec attempts to define QVT-R in two ways:

1. By translation into QVT-Core, whose semantics are directly defined
2. Directly

Both direct definitions (of QVT-R and QVT-Core) are informal.

No specification of what happens if they don't agree.

And they don't...

Translation of QVT-R to QVT-Core

To demonstrate inconsistency between the two semantics, we

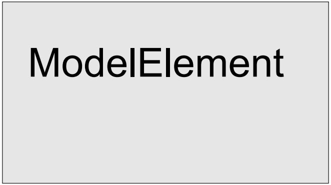
1. give a very simple transformation T whose meaning is absolutely clear under the direct semantics
2. show that no QVT-Core transformation can behave the same way as T, i.e., QVT-Core cannot express T

Then it doesn't matter if I've misunderstood the particular translation given in the spec - no translation to QVT-Core could give semantics consistent with the direct semantics of QVT-R.

SimplestMM

ModelElement

SimplestMM



And, err, that's it.

Models are `zero`, `one`, `two`...

The transformation

```
transformation T (m1 : SimplestMM ; m2 : SimplestMM)
{
  top relation R
  {
    checkonly domain m1 me1:ModelElement {};
    checkonly domain m2 me2:ModelElement {};
  }
}
```

Pick a direction to run it in, for the sake of argument towards `m2`.

T simply implements a function $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{B}$.

Which function?

According to the direct semantics

`T`, evaluated in the direction of `m2`, must return true iff

for every valid binding of some model element from `m1` to variable `me1`,

there exists a valid binding of some model element from `m2` to variable `me2`.

According to the direct semantics

`T`, evaluated in the direction of `m2`, must return true iff

for every valid binding of some model element from `m1` to variable `me1`,

there exists a valid binding of some model element from `m2` to variable `me2`.

The only way this can fail is `m1` $\neq$ `zero` and `m2` $=$ `zero`.

So checking in direction of `m2`:

$$T : (m1, m2) \mapsto \begin{cases} \texttt{false} & \text{if } m1 \neq \texttt{zero} \text{ and } m2 = \texttt{zero} \\ \texttt{true} & \text{otherwise} \end{cases}$$

Overall consistency (conjoin checks in both directions):

$$T : (m1, m2) \mapsto \begin{cases} \texttt{false} & \text{if exactly one of } m1,\ m2 \text{ is } \texttt{zero} \\ \texttt{true} & \text{otherwise} \end{cases}$$

According to the translation to QVT-Core

```
module SimpleTransformation imports SimplestMM {
transformation Translation {...imports...}

class TR {
  theM1element : ModelElement;
  theM2element : ModelElement;
}

map R in Translation {
check m1() {anM1element : ModelElement}
check m2() {anM2element : ModelElement}
where () {
  realize t:TR|
    t.theM1element = anM1element;
    t.theM2element = anM2element;
}
```

Uniqueness of bindings in QVT-Core

`Translation (two, one)` will return False!

Why? QVT-Core, unlike QVT-R:

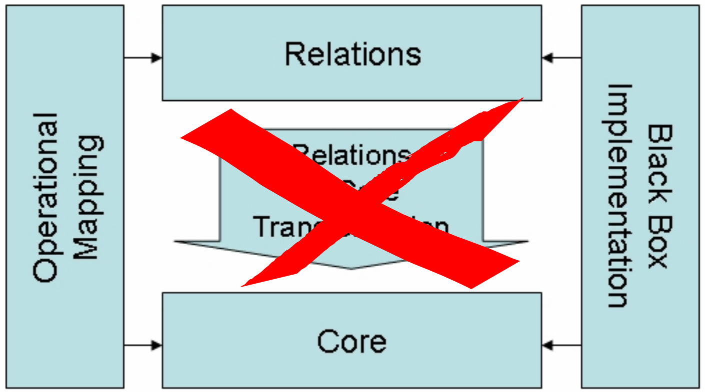
- performs checkonly transformations without specifying a direction
- insists that a valid binding in each domain must be *uniquely* determined by a choice of valid binding in the other.

Some ambiguity (see paper), but certainly, no QVT-Core transformation could return true on both `(one,two)` and `(two,one)` but false on `(one,zero)`.

That is, QVT-Core cannot express our original `T`.

And this is not a curiosity: there are many useful transformations which it cannot express.

So let's forget the translation



Operational Mapping

Relations

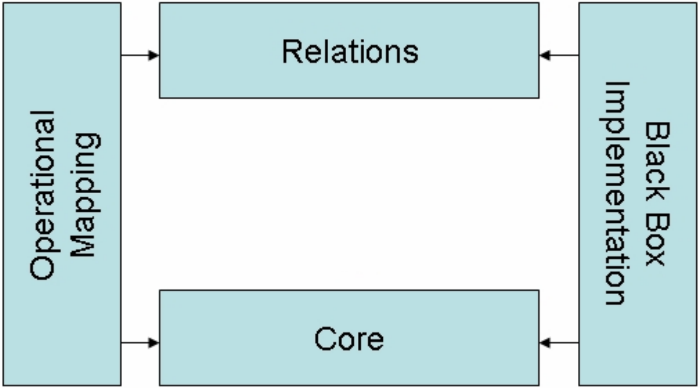
Relations

Tran

Core

Black Box Implementation

So let's forget the translation



QVT-R: the direct semantics

Informal, but fairly clear how to interpret a single relation.

Main problem is interpretation of *when* and *where* clauses.

Potentially ill-founded recursive definition of relationship satisfaction.

To express a top relation logically, we need (at least) arbitrary quantifier alternation depth (possibly fixpoints or equivalent).

Let's learn from logic and concurrency, and use games to explain what's going on.

See paper for details: here will explain using an example.

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: *where* and *when* clauses

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

"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."

Put *when* on hold for a moment...

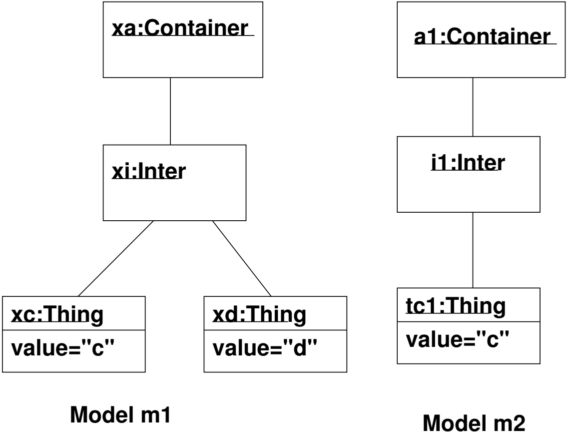
Example transformation

```
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



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 `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.

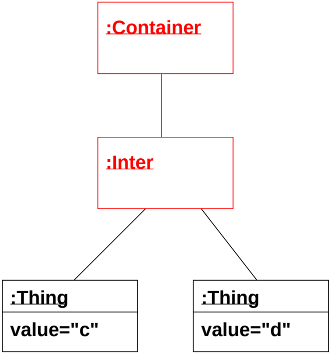
Let’s define game G to check in the direction of model `m2`.

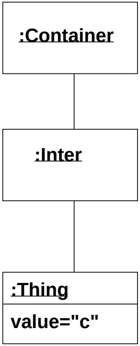
Two players, Verifier who wants the check to succeed, Refuter who wants it to fail.

Semantics: return true if Verifier has a winning strategy, false if Refuter does.

Refuter

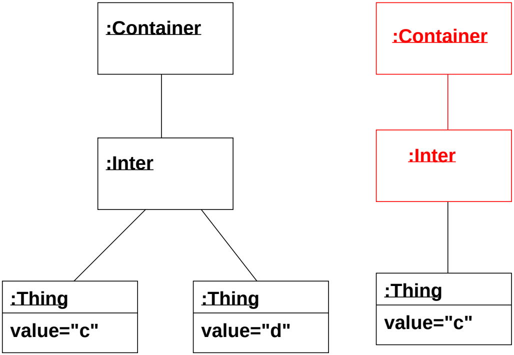
```
top relation ContainersMatch
{
  inter1,inter2 : MM::Inter;
  checkonly domain m1 c1:Container {inter = inter1};
  checkonly domain m2 c2:Container {inter = inter2};
  where {IntersMatch (inter1,inter2);}
}
```





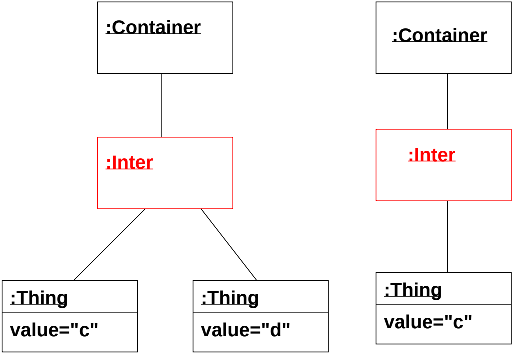
Refuter;Verifier

```
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);}
}
```



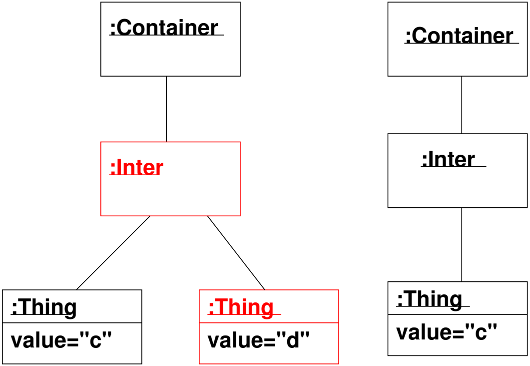
Refuter;Verifier;Refuter

```
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);}
}
```



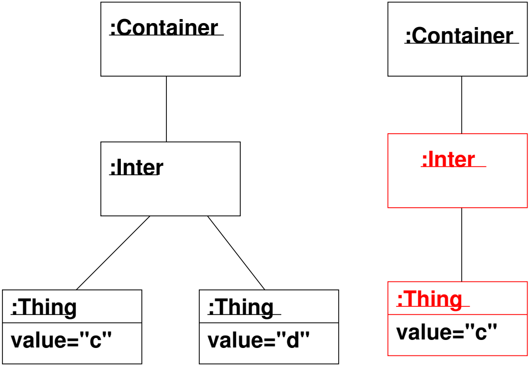
Refuter;Verifier;Refuter

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



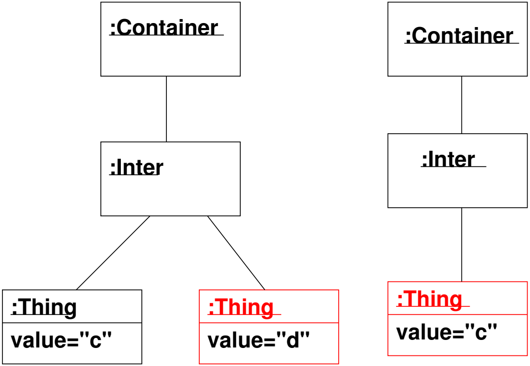
Refuter;Verifier;Refuter;Verifier

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



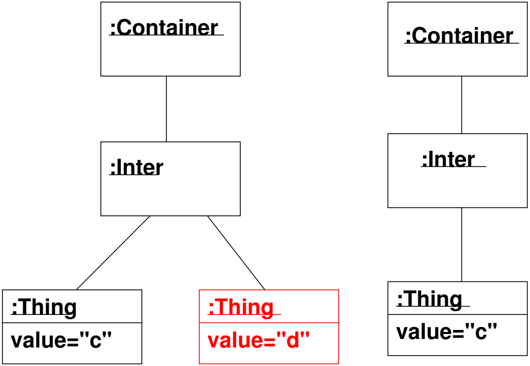
Refuter;Verifier;Refuter;Verifier;Refuter

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



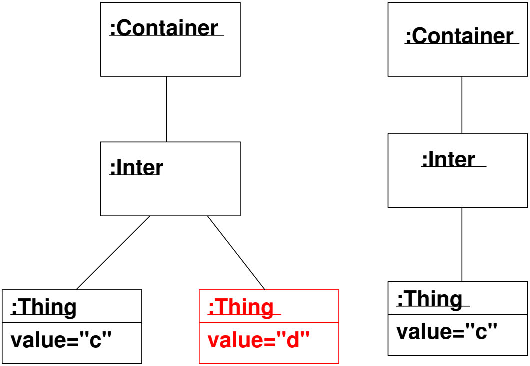
Refuter;Verifier;Refuter;Verifier;Refuter

```
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};
}
```



:Container

:Inter

:Thing

value="c"

:Thing

value="d"

:Container

:Inter

:Thing

value="c"

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 player to "counter-challenge" a *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 *when* clause. To do that I counter-challenge a relation from the *when* clause, and we swap roles and play in that relation.

(see paper for details)

Winning conditions

You win if your opponent can't go.

But what about infinite plays?

Winning conditions

You win if your opponent can't go.

But what about infinite plays?

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

Or, could define winning conditions on infinite plays (cf parity games etc.) - something like, you lose if it's your fault the play's infinite? (Future work!)

Winning conditions

You win if your opponent can't go.

But what about infinite plays?

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

Or, could define winning conditions on infinite plays (cf parity games etc.) - something like, you lose if it's your fault the play's infinite? (Future work!)

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

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.

In direction of `m1`

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

←	(T,T)	(T,F)	(F,T)	(F,F)
PwhereQ	V	R	R	R
PwhenQ	V	R	V	V
QwhereP	V	R	R	R
QwhenP	V	V	V	R

Our semantics and ModelMorf agree, hooray!

In direction of `m2`

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

$\longrightarrow$	(T,T)	(T,F)	(F,T)	(F,F)
PwhereQ	V	R	R	V
PwhenQ	V	R	R	V
QwhereP	V	R	V	V
QwhenP	V	V	V	V

Our semantics and ModelMorf agree except on one point.

`PwhenQ` 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

$$\texttt{FirstIsTrue(s1,s2)} \Rightarrow \texttt{s1.value = s2.value}$$

But what does `FirstIsTrue(s1,s2)` mean? Standard doesn't say. I say: for all valid completions of `s1` ... there exists a valid completion of `s2`...

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

Trace objects and the game

Our semantics decrees the target to be OK, according to the transformation and relative to the authoritative source, iff Verifier has a winning strategy for the game.

What does such a winning strategy look like?

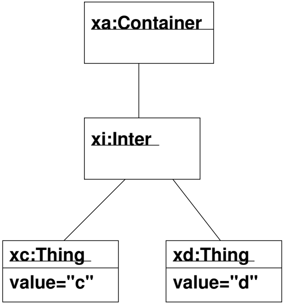
Formally, a* strategy is a sufficiently-defined partial map from {positions where Verifier is to move} to {legal moves}. It's a winning strategy if Verifier wins every play in which she follows it, regardless of what Refuter does.

For transformations without *when* clauses, that's a set of trace objects: given a relation, and a valid binding challenge, Verifier's response is to pick matching bindings.

This is a *bit* like correspondence graph in TGGs, but not the same!

* complete, deterministic, history-free

Consistent: checkonly returns true in both directions



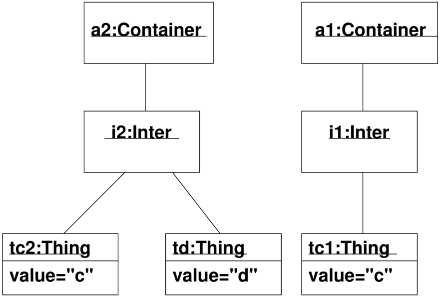
xa:Container

xi:Inter

xc:Thing
value="c"

xd:Thing
value="d"

Model m1



a2:Container

a1:Container

i2:Inter

i1:Inter

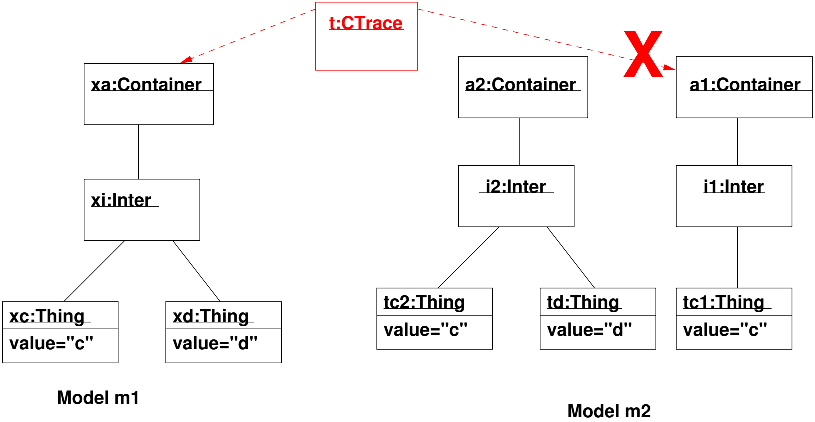
tc2:Thing
value="c"

td:Thing
value="d"

tc1:Thing
value="c"

Model m2

But with no bidirectional trace objects



What if we fiddle the game? (Bisimulation fashion!)

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.

At least for transformations without *when* clauses, a Verifier winning strategy would "be" a set of bidirectional trace objects.

NB this is definitely not an interpretation of the QVT spec: it's new semantics for existing syntax. Q: is it useful? Too strong? (Future work!)

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

Ongoing and future work

- Investigation of bisimulation-based interpretation of QVT-R, especially, is it useful?
- Winning conditions for infinite plays: what's sane, implications?
- What about enforce mode? Given a Refuter winning strategy, change one model so that Verifier has a winning strategy instead... restrictions needed?
- Implementation – on Medini as basis, using Java version of generic game engine from the Edinburgh Concurrency Workbench?