Models, programs and bidirectional transformations

Perdita Stevens

University of Edinburgh

SPLV, July 2025

Plan

1. Modelling and programming: what a fine mess
2. Model transformations: programs that...
3. State of the art: tools and practicalities
4. Where to next?

Aiming to have a little space for discussion as we go, and more at the end (else we'll finish early!)

Underlying Plan

Not so much: convince you to go and do research on bidirectional transformations, though do feel free...

More: convince you that they provide a useful framework for thinking about separation and reintegration of concerns

especially, about the choices involved.

Also: point at some unsolved problems explaining why, nevertheless, we don't have good bx languages and tools.

Warning: very partial view!

Part 1

Modelling and programming: what a fine mess

What is...

- a model?
- a program?
- the difference?
- why do we care?

First attempt

- if it's executable, it's a program?
- if it's graphical, it's a model?

but there are executable models, and graphical programs...

Purpose

A program's primary reader is a computer

- a program is a complete set of instructions telling a computer what to do

A model's primary reader is a human

- a model is an abstract, usually graphical, representation of some aspect of a system

But of course both may be read by both computers and humans.

"Complete"/"some aspect of" is wrt the set of decisions needing human input.

The fundamental problem of software engineering

Approaches to the problem

Subtly different, yet all fundamentally the same idea:

- abstraction, on every level – individual thinking, languages, verification techniques, better use of interfaces e.g. for unit testing
- separation of concerns – e.g., into models
- sequentialisation – e.g., YAGNI, bounded small releases.

All about humans

Getting chunks of processing done within the limited capacity of an individual human brain.

Controversial?

Sometimes I think the key difference between the programming language and the modelling communities is:

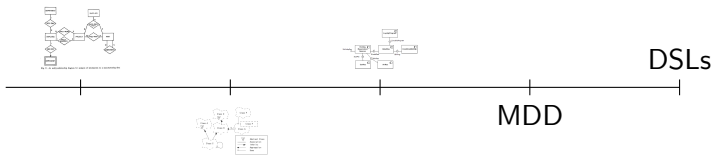
- PL people think programming is easy and fun
- Modelling people know it usually isn't.

This is partly a matter of scale/scope of course, so there's no real contradiction here.

So, modelling is for when programming is too hard

(i.e. for currently available languages/tools/people)

– especially, when multiple people are involved and nobody can understand everything



Build models when you can't afford not to

Improving cost:benefit

- Building and maintaining models – especially graphical ones - can be expensive
- the only thing more expensive is not doing so??
- Obvious need to decrease cost – e.g. need good, standard-enough languages and tools
- More interestingly: need to increase benefit, e.g. by making more use of whatever models need to be built

Aspiration:

Each decision should be recorded just once

Standard-enough languages and tools

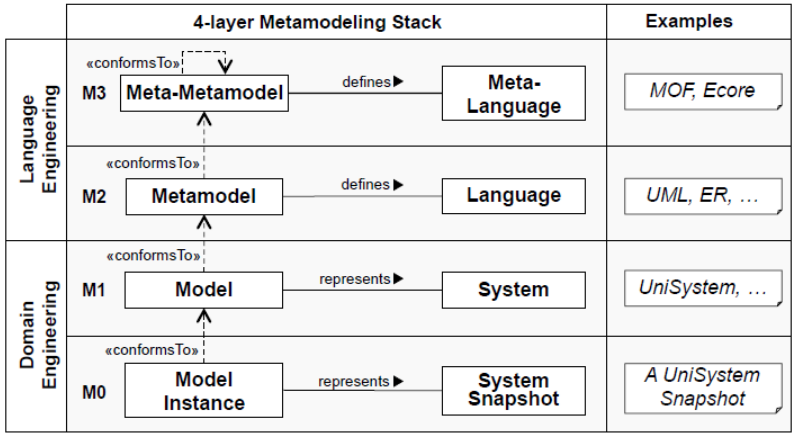
1990s: OO gurus each with their own language

late 1990s-2000s: UML

late 2000s-2010s: MDx, bx, EMF

2010s-2020s: DSLs, bx, xtext

Metamodelling



Picture from Brambilla et al., *MDSE in Practice*

EMF, xtext, diagrams or text?

Early days: diagrams are great, anyone can use them!

Gradual realisation: but they are a real pain to use and maintaining their tools is worse.

Compromise: grammarware (xtext) integrated with metamodelling (EMF).

Increasing understanding that you can, and it is often useful to, have more than one concrete syntax for "the same" language – e.g. textual for experts' efficiency, graphical for outsiders' comprehension.

Models are great for separating concerns

What about when we need to reintegrate concerns?

Great if you don't have to, things are encapsulated and orthogonal. If you can arrange that you should!

Back in the real world, concerns have overlapping or otherwise interconnected data.

These need to be brought into consistency.

Part 2

Model transformations

Recap: What's a model?

Everything's a model!

A model is an abstract, usually graphical, representation of some aspect of a system

For example

- UML model
- database schema
- map of user's navigation between screens
- bunch of Java code
- bunch of JUnit tests.

A model supports the work of a particular group of people. Ideally, it records all and only the information they need to do their work.

So, having multiple models is a consequence of separation of concerns.

Any automation of the reintegration of concerns requires model transformation.

What is a model transformation?

A model transformation is any program that involves models

e.g.

- as one input: UML model to Java
- as multiple inputs: input two models, check consistency, return boolean
- as one output: specification to UML model
- all: take multiple models, return new consistent versions

So what's special about the type "model"?

Not a whole lot really...

- Representation can be tricky...
- Graphs, not just trees! Can represent as trees but cross-references mess everything up

More special: facilitating what we do with them

Essence of bidirectionality

- models that are
- live
- not orthogonal

Models that are live – i.e. not dead yet!

A model is live if it may be needed at some time in the future. This might be because:

- someone will want to record a decision in it in future; or just
- someone will want to read it in future.

Issue is acute if it contains information that isn't elsewhere, i.e., you can't just recreate it if it is needed.

Contrast:

- Draft 1 is completely replaced by Draft 2
- Draft 1 can be thrown away.

Models that are not orthogonal

There's no problem having several live models, if the information they record is completely independent.

If it isn't, we are in a bidirectional situation.

A change in either model may necessitate a consistency-restoring change in the other. Typically there are multiple ways to restore consistency, some better than others– we'll come back to this.

JUnit $\longleftrightarrow$ Java

The two tasks of bidirectionality

1. check whether all is well;
2. if not, fix it.

Choices include

- how much to articulate of "all is well";
- how to get it fixed – by human, by bx, by both? Changing one model, changing both?
- what information to maintain in order to do all this – traces, history, deltas, edits...?

bx = bidirectional transformation = artefact for automating those tasks, maybe partially

What is a bx?

A bidirectional transformation – bx – is a special kind of program that records in one artefact

- what it means for all to be well, i.e. for models to be consistent
- if not, how to restore consistency in either direction.

These tasks are tightly coupled, so it pays to integrate their automation:

- avoid duplication of information
- guarantee sensible (predictable, dependable...) joint behaviour.

But if you don't have a bx language handy, you can write separate programs to do the bx tasks, at the cost of some duplication.

Important very special case

Some bx are bijective, i.e.

- the consistency relation is bijective
- i.e. once you have one model there is a unique consistent other model
- so if you fix one model, there's no choice about what consistency restoration must do.

Such bx are "just" reformatting some information.

Non-bijective $\neq$ non-deterministic!

Choices: JUnit/Java example

What should "all is well" mean for our Java source and JUnit tests?

1. The files compile together without error;
2. 1. holds, and the JUnit file includes a test for every public method;
3. 2. holds, and all the tests pass;
4. 3. holds, and a certain coverage criterion is met?

More stringent $\Rightarrow$

- more informative
- less flexible
- more difficult to restore consistency
- more work potentially saved for the user.

We can separate the tasks, and work with bx that do not fully restore consistency.

S., Bidirectionally Tolerating Inconsistency: Partial Transformations, FASE'14

So, thinking bidirectionally buys us

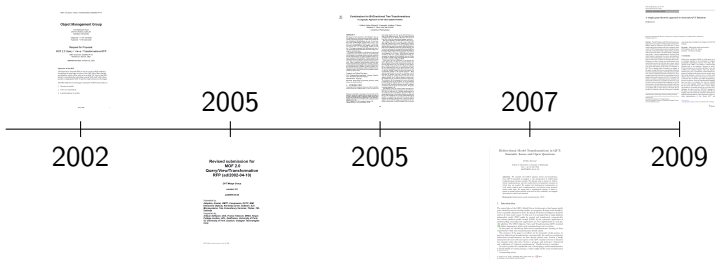
Separation of concerns:

- individual models, each recording all and only the information needed by its group of stakeholders
- reconciliation between the models.

Still beneficial, even without automation...

... especially, because these concerns require different skillsets.

Selective early history



2002: OMG QVT RFP

Object Management Group

First Needham Place
250 First Avenue, Suite 201
Needham, MA 02494

Telephone: +1-781-444-0404
Facsimile: +1-781-444-0320

Request for Proposal:

MOF 2.0 Query / Views / Transformations RFP

OMG Document: ad/2002-04-10
Revised on: April 24, 2002

Submissions due: October 28, 2002

Objective of this RFP

This Request for Proposal (RFP) is one of a series of RFPs related to developing the next major revision of the OMG Meta Object Facility specification, which will be referred to as MOF 2.0. Some of the RFPs pertain to specifying the technology neutral MOF itself, while others pertain to mapping the MOF to specific implementation technologies.

This RFP addresses a technology neutral part of MOF and pertains to:

1. Queries on models.
2. Views on metamodels.
3. Transformations of models.

2005: OMG QVT

Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10)

QVT-Merge Group

version 2.0

ad/2005-03-02

Submitted by:
Adaptive, Alcatel, UMTP, Compuware, DSTC, IBM, Interactive Objects, Kennedy Carter, Softeam, Sun Microsystems, Tata Consultancy Services, Thales, TNI-Valiosys
Supported by:
Artisan Software, CEA, France Telecom, INRIA, King's College London, LIFL, Sodifrance, University of Paris VI, University of York, Xactium, Codagen Technologies Corp

also 2005: seminal Foster/Pierce lens paper

Combinators for Bi-Directional Tree Transformations

A Linguistic Approach to the View Update Problem

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore,
Benjamin C. Pierce, and Alan Schmitt
University of Pennsylvania

ABSTRACT

We propose a novel approach to the well-known *view update problem* for the case of tree-structured data: a domain-specific programming language in which all expressions denote bi-directional transformations on trees. In one direction, these transformations—dubbed *lenses*—map a "concrete" tree into a simplified "abstract view"; in the other, they map a modified abstract view, together with the original concrete tree, to a correspondingly modified concrete tree. Our design emphasizes both robustness and ease of use, guaranteeing strong well-behavedness and totality properties for well-typed lenses.

We identify a natural space of well-behaved bi-directional transformations over arbitrary structures, study definedness and continuity in this setting, and state a precise connection with the classical theory of "update translation under a constant complement" from databases. We then instantiate this semantic framework in the form of a collection of *lens combinators* that can be assembled to describe transformations on trees. These combinators include familiar constructs from functional programming (composition, mapping, projection, conditionals, recursion) together with some novel primitives for manipulating trees (splitting, pruning, merging, etc.). We illustrate the expressiveness of these combinators by developing a number of bi-directional list-processing transformations as derived forms.

Categories and Subject Descriptors.
D.3.3 [Programming Languages]: Language Classifications—*Specialized application languages*

General Terms. Languages

Keywords. Bi-directional programming, Harmony, XML, lenses, view update problem

1. INTRODUCTION

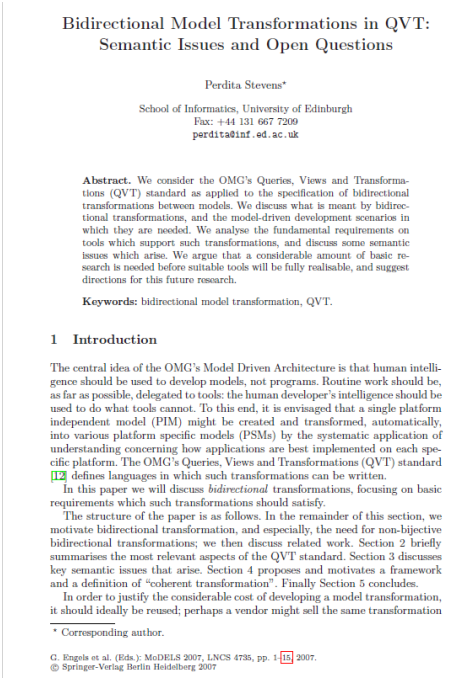
Computing is full of situations where one wants to transform some structure into a different form—a view—in such a way that changes made to the view can be reflected back as updates to the original structure. This *view update problem* is a classical topic in the database literature, but has so far been little studied by programming language researchers.

This paper addresses a specific instance of the view update problem that arises in a larger project called Harmony [26]. Harmony is a generic framework for synchronizing tree-structured data—a tool for propagating updates between different copies of tree-shaped data structures, possibly stored in different formats. For example, Harmony can be used to synchronize the bookmark files of several different web browsers, allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized in any browser and propagated to the others. Other Harmony instances currently in daily use or under development include synchronizers for calendars (Palm DateBook, ical, and iCalendar formats), address books, slide presentations, structured documents, and generic XML and HTML.

Views play a key role in Harmony: to synchronize disparate data formats, we define one common abstract view and a collection of *lenses* that transform each concrete format into this abstract view. For example, we can synchronize a Mozilla bookmark file with an Explorer bookmark file by transforming each into an abstract *bookmark structure* and synchronizing the results. Having done so, we need to take the updated abstract structures and perform the corresponding updates to the concrete structures. Thus, each lens must include not one but *two* functions—one for extracting an abstract view from a concrete one and another for pushing an updated abstract view back into the original concrete view to yield an updated concrete view. We call these the *get* and *putback* components, respectively. The intuition is that the mapping from concrete to abstract is commonly some sort of projection, so the *get* direction involves getting the abstract part out of a larger concrete structure, while the *putback* direction amounts to putting a new abstract part into an old concrete structure. We present a concrete example in §2.

The difficulty of the view update problem springs from a fundamental tension between *expressiveness* and *robustness*. The richer we make the set of possible transformations in the *get* direction, the more difficult it becomes to define corresponding functions in the *putback* direction so that each lens is both well behaved—its *get* and *putback* behaviors fit together in a sensible way—and *total*—its *get* and *putback* functions are defined on all the inputs to which they may be applied. To reconcile this tension, any approach to the view update problem must be carefully designed with a particular

2007: basic foundations of QVT



Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions

Perdita Stevens*

School of Informatics, University of Edinburgh
Fax: +44 131 667 7209
perdita@inf.ed.ac.uk

Abstract. We consider the OMG's Queries, Views and Transformations (QVT) standard as applied to the specification of bidirectional transformations between models. We discuss what is meant by bidirectional transformations, and the model-driven development scenarios in which they are needed. We analyse the fundamental requirements on tools which support such transformations, and discuss some semantic issues which arise. We argue that a considerable amount of basic research is needed before suitable tools will be fully realisable, and suggest directions for this future research.

Keywords: bidirectional model transformation, QVT.

1 Introduction

The central idea of the OMG's Model Driven Architecture is that human intelligence should be used to develop models, not programs. Routine work should be, as far as possible, delegated to tools: the human developer's intelligence should be used to do what tools cannot. To this end, it is envisaged that a single platform independent model (PIM) might be created and transformed, automatically, into various platform specific models (PSMs) by the systematic application of understanding concerning how applications are best implemented on each specific platform. The OMG's Queries, Views and Transformations (QVT) standard [12] defines languages in which such transformations can be written.

In this paper we will discuss *bidirectional* transformations, focusing on basic requirements which such transformations should satisfy.

The structure of the paper is as follows. In the remainder of this section, we motivate bidirectional transformation, and especially, the need for non-bijective bidirectional transformations; we then discuss related work. Section 2 briefly summarises the most relevant aspects of the QVT standard. Section 3 discusses key semantic issues that arise. Section 4 proposes and motivates a framework and a definition of "coherent transformation". Finally Section 5 concludes.

In order to justify the considerable cost of developing a model transformation, it should ideally be reused; perhaps a vendor might sell the same transformation

* Corresponding author.

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 1-15, 2007.

2009: checkonly QVT as a game

Softw Syst Model (2013) 12:175–199
DOI 10.1007/s10270-011-0198-8

SPECIAL SECTION PAPER

A simple game-theoretic approach to checkonly QVT Relations

Perdita Stevens

Received: 2 December 2009 / Revised: 1 February 2011 / Accepted: 15 February 2011 / Published online: 16 March 2011

Abstract The QVT Relations (QVT-R) transformation language allows the definition of bidirectional model transformations, which are required in cases where two (or more) models must be kept consistent in the face of changes to either or both. A QVT-R transformation can be used either in checkonly mode, to determine whether a target model is consistent with a given source model, or in enforce mode, to change the target model. A precise understanding of checkonly mode transformations is prerequisite to a precise understanding of enforce mode transformations, and this is the focus of this paper. In order to give semantics to checkonly QVT-R transformations, we need to consider the overall structure of the transformation as given by when and where clauses, and the role of trace classes. In the standard, the semantics of QVT-R are given both directly, and by means of a translation to QVT Core, a language which is intended to be simpler. In this paper, we argue that there are irreconcilable differences between the intended semantics of QVT-R and those of QVT Core, so that no translation from QVT-R to QVT Core can be semantics-preserving, and hence no such translation can be helpful in defining the semantics of QVT-R. Treating QVT-R directly, we propose a simple game-theoretic semantics. We demonstrate its behaviour on examples and show how it can be used to prove an example result comparing two QVT-R transformations. We demonstrate that consistent models may not possess a single trace model whose objects can be read as traceability links in either direction. We briefly discuss the effect of variations in the rules of the game, to elucidate some design choices available to the designers of the QVT-R language.

Keywords Bidirectional model transformation · QVT Relations · QVT Core · Games · Semantics · Consistency checking

1 Introduction

Model-driven development (MDD) is widely agreed to be an important ingredient in the development of reliable, maintainable multi-platform software. The Object Management Group, OMG, is the industry's consensus-based standards body, so the standards it proposes for model-driven development are necessarily important. In the area of MDD, a key standard is Queries, Views and Transformations (QVT, [15]), a specification of three different languages for defining *transformations* between models, which may include defining a restricted view of a model which abstracts away from aspects of the model not relevant to a particular class of intended user. Rather disappointingly, however, the Queries, Views and Transformations languages have been slow to be adopted. Few tools are available for any of the languages: notably, it sometimes happens that even those tools which use "QVT" in their marketing literature do not actually provide any of the three QVT languages, but rather, provide a "QVT-like" language. In this paper we will consider QVT Relations (QVT-R), the language which best permits the declarative specification of bidirectional transformations. There have been two main candidate implementations of this: Medini QVT[1] and

Communicated by Richard Paige, Jeff Gray, and Dang Van Hung.

P. Stevens (✉)
Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, Edinburgh, Scotland
e-mail: perdita@inf.ed.ac.uk

[1] http://projects.ikv.de/qvt/, version 1.6.0 current at time of writing.

QVT

The OMG's Queries, Views and Transformations (QVT) standard defines three model transformation languages:

- QVT-O (operational): an imperative, unidirectional language
- QVT-R (relations): a declarative, bidirectional language
- QVT-Core: intended as a simpler, lower level bidirectional language to serve as target of translation from QVT-R, but actually not expressive enough for that.[1]

Despite being "standard", none have become very popular.

[1] Stevens 2011, *A simple game-theoretic approach to checkonly QVT Relations*. Software and Systems Modeling 12:175. doi:10.1007/s10270-011-0198-8

plus oops, the official semantic framework can't be right

QVT-R

A QVT-R transformation T is defined in terms of two (usually) metamodels, say M and N.

It comprises relations which are connected by when- and where-clauses. (Example next slide.)

It can be run on a pair of models, in two modes:

- checkonly mode: check whether the models are consistent according to the transformation, return true or false;
- enforce mode: change one of the models, by adding, deleting or modifying its elements, so that afterwards the models are consistent according to the transformation.

QVT-R example (from the spec)

```
relation ClassToTable /* map each persistent class to a table */
  domain uml c:Class {
    namespace = p:Package {},
    kind='Persistent',
    name=cn
  }
  domain rdbms t:Table {
    schema = s:Schema {},
    name=cn,
    column = cl:Column {
      name=cn+'_tid',
      type='NUMBER'},
    primaryKey = k:PrimaryKey {
      name=cn+'_pk',
      column=cl}
  }
  when { PackageToSchema (p, s); }
  where { AttributeToColumn (c, t); }
}
```

Strengths of QVT-R

- Allows you to express what consistency means, and something about how it should be restored, in one text
- Well-adapted to talk about models in languages which are defined using MOF (e.g., UML)
- The basic "relation" construct seems rather natural

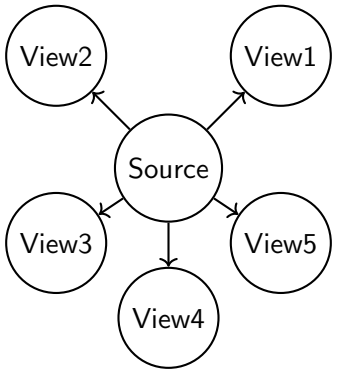
NB QVT-R can be seen as a DSL for expressing bidirectional transformations! (It also has a graphical concrete syntax, which is basically never used.)

Problems with QVT-R

- Lack of available tools.
- Confusion: tools that claim to support QVT-R actually having very different semantics.
- Lack of clarity in the standard (and this matters far more than for, say, UML).
- when- and where- clauses probably not good structuring mechanisms.

Assumptions and properties

Let's start with the simplest setting, corresponding to what we need if we can keep our models in a star configuration



If it is possible this is by far the easiest thing to do.

Set aside model-specific considerations temporarily and consider lenses.

Lenses

Basic assumption of a lens: there is a Source datatype S which contains all the information and a View datatype V which contains some of it.

Foster-Pierce:

$$\begin{aligned} \text{create} &: V \longrightarrow S \\ \text{get} &: S \longrightarrow V \\ \text{put} &: V \times S \longrightarrow S \end{aligned}$$

Basic lens properties

- create-get : $\text{get}(\text{create}(v)) = v$
- get-put : $\text{put}(\text{get}(s), s) = s$
- put-get : $\text{get}(\text{put}(v, s)) = v$

Roughly: changes are actually implemented, and nothing else.

What does that have to do with maintaining consistency?

Consistency between a source s and a view v just means $\text{get}(s) = v$

So:

- check consistency using get;
- restore consistency if the source changes using get;
- restore consistency if the view changes using put.

A (slightly) different symmetric framework

Let M and N be sets of models to be related. A bx between M and N consists of:

- a consistency relation $R \subseteq M \times N$

and two consistency restoration functions:

- $\overrightarrow{R} : M \times N \longrightarrow N$
- $\overleftarrow{R} : M \times N \longrightarrow M$

Distinguished "no information" models:

$\Omega_M \in M$, $\Omega_N \in N$.

Correctness

The "job" of the transformations is to enforce the relation.

So they better had do that:

$$\forall m \in M \; \forall n \in N \qquad R(m, \overrightarrow{R}(m, n))$$

$$\forall m \in M \; \forall n \in N \qquad R(\overleftarrow{R}(m, n), n)$$

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:

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

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

Connections

If M happens to have all the information that N does

we can take

$S = M$

$V = N$

$\text{create}(v) = \overleftarrow{R}(\Omega_M, v)$

$\text{get}(s) = \overrightarrow{R}(s, \Omega_N)$

$\text{put}(v, s) = \overleftarrow{R}(s, v)$

and as you'd expect, the lens laws are equivalent to correct and hippocratic.

Composition

Lenses compose (exercise...)

In general, state-based bx do not.

Composition is less important in MDD than you might think!

Beyond lenses

Lots of things we haven't talked about, especially, many richer settings

- with complementary info to give symmetric bx that compose (Hofmann Pierce Wagner...)
- Triple Graph Grammars will briefly return to these...
- with deltas to capture what change was made, not just the result (Diskin...)
- with effects

And many many properties (e.g., kinds of "least change")

Properties

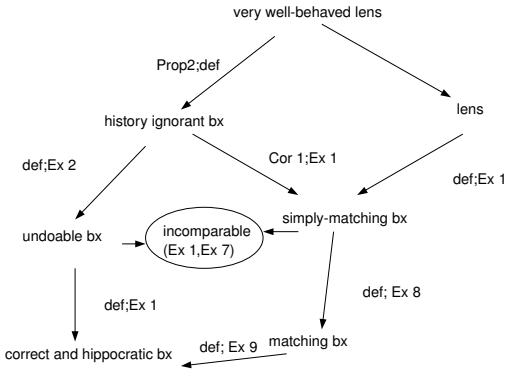
We've seen the basic ones (almost) everyone agrees on:

create-get, put-get, get-put = correct, hippocratic

Beyond that things get hairy:

- strong undoability/history ignorance/put-put : lovely if you can get it but generally you can't (unless you cheat)
- least change: what exactly should this mean except "I know it when I see it"?

Structure-related properties



Observations relating to the equivalences induced on model sets by bidirectional transformations, S.

Least change properties

Hippocraticness is a very basic form of Least Change property: if nothing needs to be changed, change nothing.

Going further is surprisingly hard. We can consider:

- metrics, e.g. edit distance – but this may be tool-dependent and even so it may not give sensible results
- witness structures, e.g. to mitigate those problems
- people! suggesting move to least Surprise
- continuity...
- variants such as Hölder continuity
- category theory...

Principles of Least Change and Surprise for Bidirectional Transformations, Cheney, Gibbons, McKinna, S.

Part 3

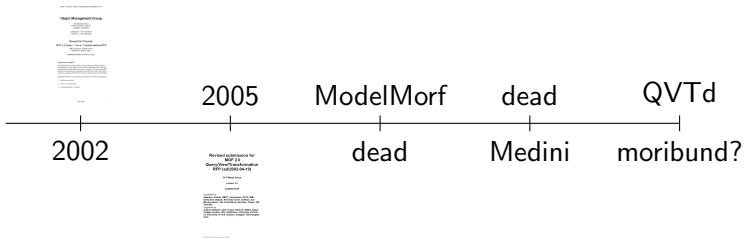
State of the art: Tools and practicalities

Theory's all very well, but

what about tools? And bx languages?

If I want to build a system with bx to integrate concerns, what do I use?

Timeline: OMG bx



Triple graph grammars

Beautiful category-theory-supported setup...

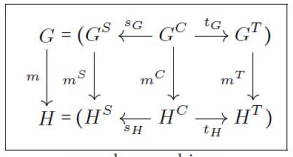
...with lots of complications for practical use.

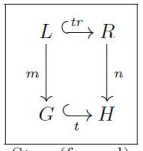
Best bet for MDD bx tooling (not saying much!) e.g.

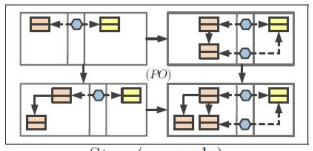
eMoflon::neo `https://emoflon.org/neo/`

Triple Graph Grammars (TGGs)

Bx defined by a collection of triple rules, which define a language of integrated triple graphs $\{(s, c, t)\}$. The pairs (s, t) from such triples are said to be consistent (the correspondence graph c helps witness the consistency, taking us slightly beyond the QVT-R setting).


graph morphism


Step (formal)



Step (example)

Strengths of TGGs

- Good tool support.
- Well-developed underlying theory.
- Graphical notations that are good for expressing consistency, if this is a pretty close structural similarity.

Problems with TGGs

- Very difficult to write a TGG when the consistency relation must relate graphs that are not very similar structurally.
- Can't straightforwardly handle deleting elements.
- Problems specifying when a rule should *not* be allowed to apply (negative application conditions).
- Perhaps a well-explored dead end?

TGG rule example

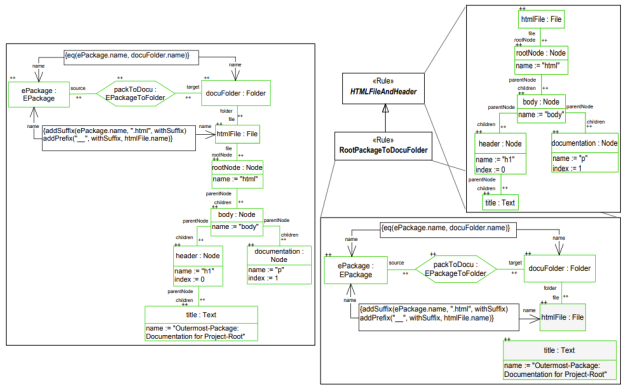
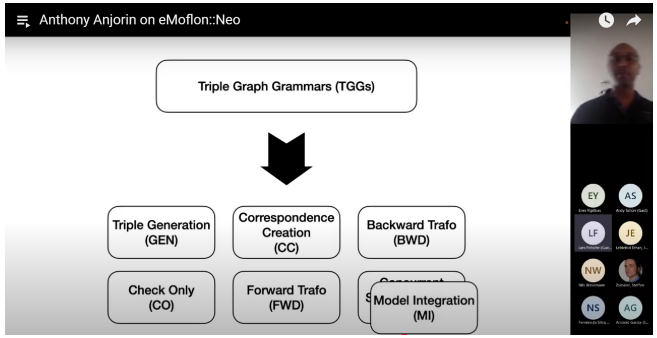


Figure 3: Handling root packages: without (left) and with (right) rule refinement

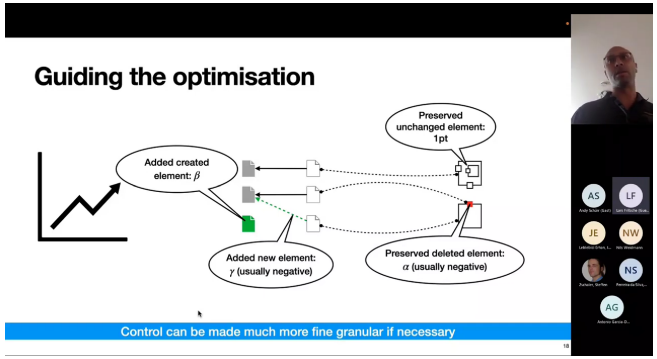
fig from *A Systematic Approach and Guidelines to Developing a Triple Graph Grammar* Anjorin, Leblebici, Kluge, Schürr, S.

Eneo



still from MDEnet training session, link athttps://emoflon.org/neo/

Consistency restoration as optimisation



still from MDEnet training session, link athttps://emoflon.org/neo/

BXtend

Looks promising[2] : pragmatic combination of declarative and imperative DSLs building on Java dialect xtend.

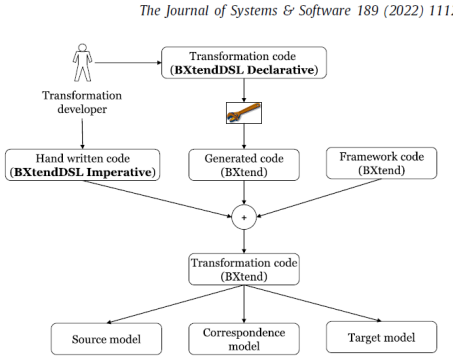


Fig. 1. Layered approach to bidirectional transformations.

BXtendDSL: A layered framework for bidirectional model transformations combining a declarative and an imperative language Buchmann, Bank, Westfechtel

[2] on a brief look; but caveat, update site last updated 3 years ago

A few other important things...

...that I'm not talking about.

- Putback-based programming, e.g. BiGUL `https://hackage.haskell.org/package/BiGUL` (Hu et al.)
- Bidirectionalisation (Voigtländer et al.)
- Answer-set programming approaches e.g. JTL (Cicchetti et al.)
- ...

Bx in (functional) programming

Cottage industry in lens libraries, even optics libraries.

Lots of impressive examples.

- Haskell `https://hackage.haskell.org/packages/tag/lenses` e.g. `https://hackage.haskell.org/package/optics-0.4.2.1`
- Ocaml `https://ocamlverse.net/content/lenses.html`
- Even python! `https://python-lenses.readthedocs.io/`

Philosophy

"An optic is a first-class, composable notion of substructure." [3]

"Lenses are a modular, functional method to update and read immutable record fields." [4]

Different emphasis to MDD!

But "A is consistent with B iff A embeds in B thusly" is a fine notion of consistency.

Using these libraries

These libraries enable you to write code that is

- short
- elegant
- exquisitely dependent on your choice of data structure
- devoid of meaningful names.

[3] `https://hackage.haskell.org/package/optics-0.4.2.1/docs/Optics.html`

[4] https://ocamlverse.net/content/lenses.html

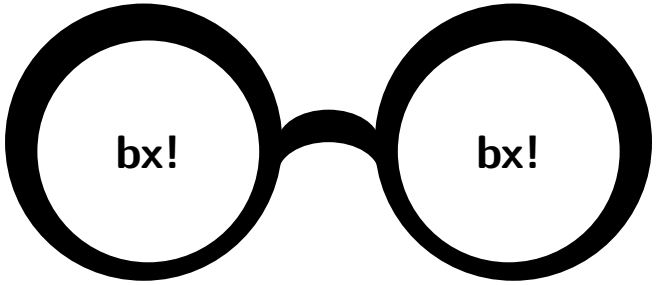
So where do we go from here?

- lessons for practical consistency maintenance now
- challenges to be addressed in future
- relevance of AI??

Part 4

Where to next?

Practical consistency maintenance now



- separation of concerns
- explicit definition of consistency
- checking consistency distinct from restoring consistency
- choice of how much to automate

Define what consistency is required

This is perhaps the biggest mindset shift.

Don't focus on what made the data – as soon as it may have been altered by people or processes beyond your control that's a lose.

Focus on what should be true about it.

Even if the checking of consistency is all that is automated, this can be a big win.

Golden Copy

"If it isn't in the family calendar it hasn't been agreed"

Bx are interesting when there is no data source that has all the information.

"Interesting" is very bad.

If at all possible, ensure that there is one reliable source: have a protocol that it is updated when anything changes, and if that update hasn't happened, the change has in effect not happened.

Understanding the setting

Suppose you have multiple live, non-orthogonal models. Choosing a solution requires understanding:

- Who (with what job, skills, info) is working on each?
- Are they always online?
- Are they in communication with one another?
- What is the surrounding bureaucracy? Who has what authority?
- What properties are essential/desirable/achievable?

Etc. etc.

How to Regain Equilibrium without Losing your Balance? Scenarios for Bx Deployment (Discussion Paper), McKinna, S.

Challenges to be addressed in future

- Languages and tools that offer better support for consistency maintenance – putting consistency front and centre.
- Bx in the large – when there are many models and many consistency relations, maybe many bx languages.
- Managing the division of labour – make bx use work for someone expert in only one of the concerns.
- Beyond "all or nothing" consistency

Bx languages

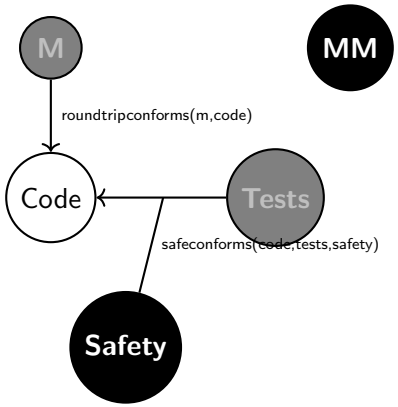
What would the ideal consistency-oriented bx language look like?

QVT-R has a lot going for it... but turns out to have semantics closely analogous to μ-calculus... which is not a good thing.

QVT-R : μ-calculus :: ? : MSOL

A simple game-theoretic approach to checkonly QVT Relations, S.;
Enforcing QVT-R with mu-calculus and games, Bradfield, S.

Bx in the large



Bidirectional transformations in the large, S.;
Towards sound, optimal, and flexible building from megamodels, S.

Relevance of AI??

Anyone's guess... what's yours?

Maybe: we see a split between

- vibe coding/modelling for most decisions, made by stakeholders and/or AI
- and a thin layer of reliable checking, done by programmers?

In bx: perhaps this means we want reliable checking that

- a model is really in the DSL it's supposed to be in
- models that are supposed to be consistent really are

Interesting read:

Prompting Bidirectional Model Transformations - The Good, The Bad and The Ugly, Buchmann

Languages as types

Raise abstraction level: the construction of the program/model is inscrutable, but once constructed it must conform to the right DS(M)L.

At the whole-system level: consistency is well-typedness.

This should be a slogan already but doesn't seem to be: closest I found was *FLAT: Formal Languages as Types* F Zhu, A Zeller, arXiv:2501.11501

Questions? Comments?