Model-driven development

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

Synthesise the disparate strands we've seen so far and talk about the state of the art in model-driven development.

(Definition coming up. For our purposes, MDD includes development that makes serious use of domain-specific modelling languages.)

Initial view: ways of using models

At the beginning we said: UML use varies across projects and organisations, e.g.

- people scrawl UML diagrams on napkins and whiteboards
  – ephemeral models
  you know how to do this
- UML diagrams appear in documents (sometimes after the code has been written)
  – models as static documentation
  you know how to do this
- UML diagrams are developed in tools before the code, and code is generated from/in parallel with them
  – model-driven development (MDD)
  you've started to see this in labs

Currently these are alternatives...

… so let's explore the tensions

I claimed: "Good modelling, design and testing should let you change the software quickly and without breaking it, when things change."

… so let's explore the tensions

I claimed: "Good modelling, design and testing should let you change the software quickly and without breaking it, when things change."

It was disingenuous of me to lump modelling, design and testing together: having done so, I can defend the statement.

- Testing… yes, essential for changing things without breaking them. (There's still a cost-benefit question, though, and lots of issues about which tests you write, and even which you run.)
- Design… yes, by definition (good design *is* the decisions that let you do that…)
- Modelling… helps you think about design, but is it important otherwise? That depends on the lifespan of the model, and the speed of change of the environment, among other things.

Speed of change and lifespan of models

Currently the way in which modelling can sensibly be used is strongly influenced by how fast the design is expected to change.

We'll consider separately two cost-benefit equilibria easily available today:

- long-lived models for slowly-changing design
- short-lived models for fast-changing design.

Long-lived models for fast-changing design: often unattractive because too much work to keep the model up to date, for too little benefit. Ongoing challenge: make this work, by decreasing cost and increasing benefit.

Short-lived models for slowly-changing design: likely to be suboptimal, as if the model is useful at all, you can probably get benefit from saving it.

Slowly-changing design

The waterfall idea – requirements, analysis, design, implement, test – is generally fiction (and, NB, regarded as such by W. Royce who coined the term "waterfall"!)

However, rarely, "big design up front" (BDUF) really is necessary:

- when implementation is not easy to change, e.g., involves building hardware
- when very costly verification processes need to be done – safety-critical software.

Then we typically need detailed, carefully checked, tool-supported modelling.

Fast-changing design

If change to the software is frequent, and done at code and test level, modelling has to be as quick and easy as possible.

Hence agile modelling usually done on whiteboards or scrap paper.

Do *not* want to spend hours with a UML tool or, worse, drawing tool to update a beautiful model of the current design if this will change again tomorrow anyway and the work will be wasted.

This is why organisations that mandate UML diagrams in design documents, but also need designs to change, often get models written after the code is complete!

Vision of future: MDD as easy as modelling on a whiteboard, *and* bringing extra benefits e.g. most code generated/updated automatically.

Agile software development

began as a reaction against slow, high-ceremony (document-intensive), expensive software development methodologies.

Aimed to counteract criticism that the only alternative was hacking by making principles and practices explicit. E.g. Manifesto for Agile Software Development (2001) signatories declared that they value:

- Individuals and Interactions over processes and tools
- Working Software over comprehensive documentation
- Customer Collaboration over contract negotiation
- Responding to Change over following a plan

NB this does not mean they don't value the things on the right!

Many processes under this umbrella, e.g. Extreme Programming, Scrum, Kanban.

Modelling, design and testing

support each other:

- good modelling lets you pick a design that will work
- good testing helps you refactor a design when necessary
- good design lets you test effectively (via well-chosen APIs)

Key idea of agile development (though not unique to it):
simple design is easier to change and less error prone.

As simple as possible, but no simpler... what counts as simple will depend what functionality has to be provided, which will change.

Modelling helps you find a simple design, testing helps you get to it.

Testing

Tests are important anyway, but *essential* for changing software quickly and without breaking it.

Recall the refactoring approach. When you need to make a change (add functionality, fix bug):

1. Refactor the system into the state you wish you were starting from (now you are!)
2. Make the change.

Refactoring is a key technique for controlling technical debt.

1. The refactoring step

In small increments – your aim is to make the change steps just small enough that you never make a mistake, so all these tests always pass (ha!):

1. Run (at least the relevant) existing tests – presumably they pass, but this is a good sanity check
2. Do a refactoring step (recall, a small redesign step, not altering the functionality: e.g., eliminate some code duplication)
3. Rerun the tests to check they still pass.

Repeat until the codebase is how you want it to be, i.e., ideally designed to make the change you're about to make easy.

2. The change step

Now that your codebase is in good shape, with a clean simple, tested design that will support your change easily:

1. Write new tests that should eventually pass, but will currently fail (i.e. tests that demonstrate the bug you're about to fix, or that the new functionality you're about to add should pass)
2. Make your change
3. Rerun all the (relevant) tests, new and old.

It's highly likely that some test (new or old!) will fail, but because the design was so clean, it's easy to fix.

Summarising so far...

If models are not automatically kept in sync with code, the work involved in changing a model duplicates the work in changing the code.

So either:

- minimise number of changes to the model (BDUF, or write the model afterwards); or
- minimise the need to update models when things change (ephemeral modelling, or none at all)

Both have severe disadvantages.

Disruptive idea

What if we could change code and model together, for no more cost – maybe even less – than the current cost of changing code? This, simplifying wildly, is the idea of MDD.

Model-driven development

Means different things to different people, but roughly:

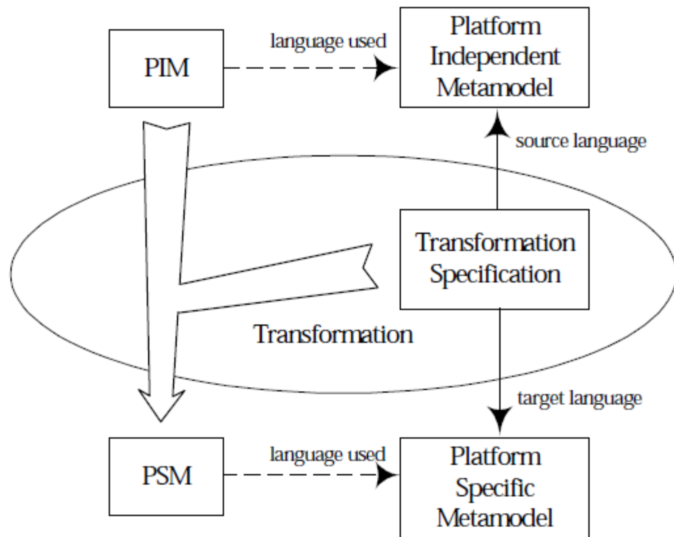
- treat models as important, first-class artefacts in development
- large development may include many models, each adapted to the needs of its users (UML design model, database model, architecture model; can also regard e.g. code and documentation as models)
- use tools to avoid duplicating work (WRITE ONCE), so
- decisions recorded in one model can be automatically rolled through to any other models, including code, using
- model transformations.

Model transformations

A model transformation is a program that can create or modify a model, typically using information from one or more other models. E.g.

- code generator:
  – input a UML model, output (skeleton?) Java code.
- documentation generator, e.g. JavaDoc:
  – input a UML model, or Java code, or whatever, and generate pretty documentation.
- A more sophisticated task is roundtrip engineering:
  – input a UML model and some Java code; change them to be consistent.
  Here be dragons! Tools that do this exist, but they tend to be fragile and unpredictable. Active research area.

OMG's Model-driven architecture



OMG: MDA Guide Version 1.0.1,
http://www.omg.org/docs/omg/03-06-01.pdf, June 2007

OMG MDA essentially doesn't work. Why?

The standard general-purpose languages like UML are far too complex for it to be practical to define fully-automatic transformations.

Two possible responses (and they can be combined):

- Use much simpler languages – DS(M)Ls – so that transformations can be fully automated.
- Accept that the transformations will not be fully automatic. Let people work and make decisions at the most appropriate level: use bidirectional transformations to roll the effects of these decisions through to other models, maintaining consistency.

Neither is a solved problem! We talked about DS(M)Ls; now let's spend some time on bidirectionality.

Why bidirectional transformations?

Separating concerns (Dijkstra) – e.g., into various models – is key to effective software development.

Why bidirectional transformations?

Separating concerns (Dijkstra) – e.g., into various models – is key to effective software development.

But how do we re-integrate those concerns? i.e. maintain consistency between the models, when any of them may be used to record a decision?

Maintaining consistency between models is the job of a bidirectional transformation.

Bidirectionality was optional in OMG's Request For Proposals for a language to express transformations – but high up in users' requirements list.

Different people, different expertise

Typically, on a large project, one team (maybe the “architecture” team) will work on the PIM and another (maybe a “product” team) on (each) PSM.

Expertise in *their* model.

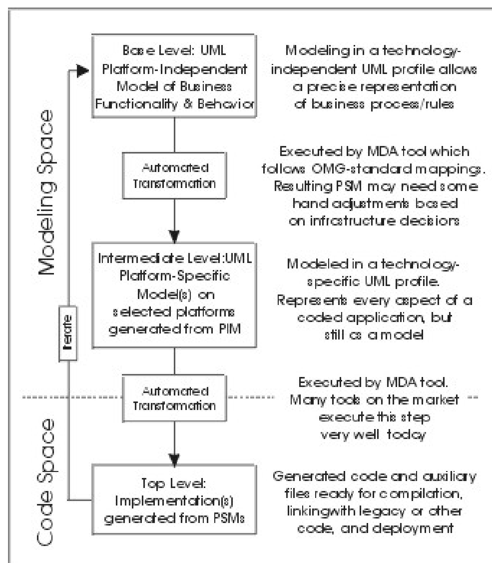
(Maybe even only in *the language of* their model – e.g., RDBMS vs UML, an ADL vs UML...)

If the product team discovers in the course of writing the code that the PSM is not quite right, we want it to be easy for them to make the change and roll it back to the PIM

- in case it has implications for other PSM teams

- for deployment, maintenance etc.

Current view of OMG's MDA



http://www.omg.org/mof/

Concretely: bidirectional transformations

A bidirectional transformation (bx) has two, related, jobs:

1. check whether given models are consistent
2. if not, change one of them to restore consistency, on the assumption that the others are authoritative and must not be changed.

You could have separate programs doing these jobs – but they'd change together and duplicate lots of information.

Motivates bx languages.

Simpler example

Suppose your system is defined by a UML class diagram, and you want its state persisted in a relational database.

You shouldn't have to maintain the RDBMS schema manually, e.g. add a new table when a new persistent class is invented, rename an attribute if a column is renamed...

You'd like to describe the consistency you want between the UML class diagram and the database schema precisely, once, and have it maintained and checked for you.

(This is a famous – indeed infamous – example, but we are eliding many things that would have to be thought about...)

Restricted setting

State-based, relational approach to bidirectional transformations (bx):

- A bx works on a pair of models drawn from a pair of model spaces, typically just sets of models defined by metamodels;
- we have a relation R on pairs of models which specifies what it is for them to be "in synch" or consistent;
- all that matters is the current state of the models (not how they got to that state);
- we store no extra information about how parts of models are related;
- an application of consistency restoration changes only one model.

Restrictive, but enough to model the OMG standard bidirectional language QVT-R.

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

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

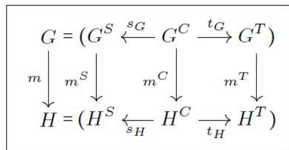
NB QVT-R can be seen as a DSL for expressing bidirectional transformations! (It also has a graphical concrete syntax, which is basically never used.)

Problems with QVT-R

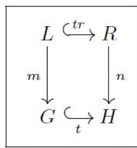
- Lack of available tools.
- Confusion: tools that claim to support QVT-R actually having very different semantics.
- Lack of clarity in the standard (and this matters far more than for, say, UML).
- when- and where- clauses probably not good structuring mechanisms.

Triple Graph Grammars (TGGs)

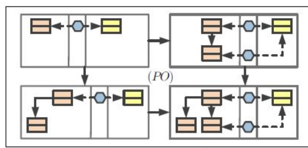
Bx defined by a collection of triple rules, which define a language of integrated triple graphs $\{(s, c, t)\}$. The pairs (s, t) from such triples are said to be consistent (the correspondence graph c helps witness the consistency, taking us slightly beyond the QVT-R setting).



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?

Bidirectional vs bijective

Suppose

Important special case:

$$\forall m \in M \; \exists! n \in N : R(m, n)$$

and vice versa – if you have one model, the other is determined.

If this holds, the bidirectional transformation is said to be bijective.

Then there exist inverse functions $M \longrightarrow N$ and $N \longrightarrow M$ taking a model to the unique related model.

“Easy” – but not sufficiently general.

...

Why aren't bijective transformations enough?

Basically: because some of each model will be "irrelevant".

(E.g., transform a full UML model, including dynamic diagrams, to RDBMS schema)

More interestingly: genuine choice about how to resolve inconsistencies. Bx programmer needs a way to make that choice (perhaps with help from bx user...)

(E.g., every class that has a state diagram is supposed to correspond to a test set. A test set is deleted and we're supposed to update the UML accordingly. Should we delete the class? Delete its state diagram?)

Properties of bidirectional transformations (bx)

We may agree that a bx should be:

- correct: after consistency restoration, the models should be consistent!
- hippocratic: if the models are already consistent, then consistency restoration should do nothing.

Beyond that, agreeing properties is surprisingly hard.

- It is easy to write down properties that would be desirable, but cannot be guaranteed, e.g. history-ignorance (change one model, restore consistency, change the first model back to where it was, restore consistency again: are you back exactly where you started?)
- Some properties are informally desirable, but hard to make precise (Least Surprise).

Implications for language design...

Summary

How models are used is intertwined with how they are related, including related to code, and with use of tools.

OMG's MDA approach aimed to use model transformations to automate much of software development;

but was not very successful as originally envisaged.

Two ongoing developments in the field are promising:

- use of domain specific languages;
- bidirectional transformations.