Issues surrounding model consistency and QVT

Laurence Tratt, Tony Clark
laurie@tratt.net, anclark@dcs.kcl.ac.uk

December 16, 2003

1. Introduction

This document is intended to outline some of the issues surrounding model consistency in the context of QVT. As far as possible this document is intended to be independent of any particular QVT submission, although we do borrow some terminology from the Q^{V}T-*Partners* submission.

2. The issue

The problem at hand can be succinctly phrased thus:

> If one has two or more models that are related to each other via a transformation, and then changes one model, what notification is given to the user what happens to the other models?

To elaborate on this, we are interested in the situation where one model is the result of a transformation of another, and to determine what effects changes upon either model have upon the other (directly or indirectly). Possible outcomes include no action, interaction with the user, and automatic action by the transformation system.

In the rest of this section we outline some specific real-life scenarios where consistency between models is an important issue, and where one would expect to see tool support to detect inconsistencies, or to enforce consistency.

2.1. Loading and saving XMI files

This scenario is one that, in various guises, most of us find ourselves doing very frequently – in this instance we have cast the scenario in terms of a modelling tool that loads and saves XMI files. In such a situation, a user loads an XMI file into a modelling tool, at which point the XMI input will be transformed into the tools internal representation of models. It is reasonable to assume that at the point of loading, the tools internal representation of the model and the XMI representation will be consistent with each other.

After loading, assume that the user edits the model within the tool – at this point the tools internal representation of the model and the XMI representation will be inconsistent with each other. This inconsistency is resolved in one of two ways: the user requests the tool to transform its internal representation of the model into XMI, and then overwrite the version on disk; the user requests the tool to reload the model from the XMI representation on disk.

Upon first sight this scenario might seem painfully obvious, but it does in fact demonstrate two important points: that users are used to manually enforcing consistency between two representations of the same underlying model, and that the method of enforcing consistency (where one or the other representation is destroyed) is typically very crude.

2.2. Legacy system conversion

One of the goals of transformations is to allow the conversion of systems running on old or outdated platforms[1] into systems running on new platforms. Imagine a scenario where one has a large Ada system which one wishes to transform into a Java system. Once the initial transformation has been performed, there will be a potentially lengthy stage of testing and modification of the Java system; during this time, modifications to the Ada system are likely, and one would expect to see some changes in the Ada system being reflected in the Java system, either via an automatic or manual process. However, the opposite is most certainly not true: one does not wish to see changes in the new system being reflected in the legacy system, as this would destabilize the running legacy system.

As the new system is likely to increasingly diverge from its original transformation from the legacy system over time, although one will expect the transformation system to detect such inconsistencies, expecting that a tool can automatically correct them is extremely ambitious.

2.3. Code generation

The transformation of PIMs to PSMs is a fundamental part of MDA. The scenario we choose to illustrate this is hopefully familiar to most people: the transformation of a UML model to Java. Initially this may seem similar to the legacy system conversion scenario (section 2.2). However the fundamental difference here is that the user expects changes made to either the PIM or PSM to be reflected in the other.

2.4. Scenarios summary

The preceding scenarios have been intended to show the following points:

- Users are already used to automatic, and destructive, consistency enforcement in their everyday use.

[1]Without wishing to delve into the specifi cs of ‘what is a platform’, for our purposes a platform can be assumed to be a combination of programming language, libraries, development tools and operating system.

- Some transformations require consistency checking/enforcement in only one direction.
- Transformations might be expected to check/enforce consistency between two models over a long period of time during which an arbitrary number of changes to the models may occur.

2.5. When inconsistency is acceptable

It is important to realise that not all inconsistencies need to be resolved. To explain the need for this, we use an analogy from source code revision systems. In such systems users will typically be working upon a local check out of a remote repository, refreshing it at regular intervals, but only relatively infrequently sending their updates to the repository. At best, their local copy will only be fully in sync with the repository in the moments after they have checked in all of their updates; in practise it is very common for users to only ever check in a subset of their updates at any given point, meaning that they are never fully in sync with the repository.

In our context, we think that as with source code revision systems, users will often wish to be able to keep certain parts of their transformed models out of sync. For example in the legacy system conversion scenario (section 2.2), the user may edit the Java system in such a way that it no longer bears any resemblance to the original Ada system, in which case they not only do not wish the transformation system to attempt an automatic reconciliation, but furthermore do not wish to be continuously reminded upon every check for inconsistencies that that particular part of the system is inconsistent. A good consistency checking scheme will provide for such cases by allowing the user to inform the system that they do not wish to be informed of such inconsistencies[2].

3. Potential solutions

One can imagine a number of different ways in which implementations could provide support for some or all of the situations outlined in the preceding parts of this document. In this section we aim to give a high level categorization that encapsulates the important aspects of different solutions. It is our contention that all solutions can be viewed in terms of model and model change deltas (see section 3.2).

In order to explain different aspects, we will use the simple example shown in figure 1 – a transformation involving classes and relational databases. We assume an intuitive understanding of the underlying transformation this figure represents – the precise details of this transformation are not important for the points we are trying to make in this document.

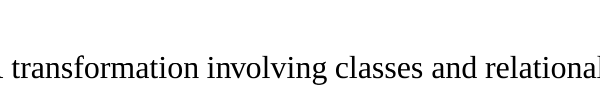

Figure 1: A transformation involving classes and relational databases

[2]A simple analogy from CVS is its `.cvsignore` fi les.

3.1. The QVT engine

For the purposes of this document, we assume that that a black box QVT implementation exists, henceforth referred to as the *QVT engine*. What the QVT engine contains depends on what functionality one desires, and where one wishes that functionality to be implemented. In this document we are dealing with two distinct, but related activities: the transformation of models, and the detection and resolution of inconsistencies between models that are related by a transformation. In some of the potential solutions we discuss, the QVT engine need only consist of the transformation aspect; in some cases the QVT engine would need to have some support for consistency aspects; in other cases the QVT engine will need to contain a sophisticated consistency aspect which will have a tight coupling with the transformation aspect.

3.2. Change deltas

Change deltas denote the change between two different things; another way of thinking of them is as the steps necessary to change one thing into another. Note that this second definition does imply a sense of direction with change deltas, although it is generally possibly to use deltas in the opposite direction to which they were originally intended. Change deltas are fundamental to the operation of source-code revision systems, and are useful in their own right in many ways – the canonical example of a delta-producing program is the Unix `diff` tool. See section A.1 for information on `diff` and exploration of some more advanced issues with change deltas.

3.2.1. Textual change deltas

As a simple example of change deltas, imagine that one has two files containing plain text data. The first file contains:

```
1
3
```

and the second contains:

```
1
2
3
```

If we were to generate a change delta from the first to the second file, it would look along the lines of ‘place a line containing the character `2` between the lines containing `1` and `3`.’ One obvious use for this, is for a human to analyse the changes from one file to another. Another use is that this change delta can then be passed onto someone else who has a copy of the first file, and the application of the delta to the first file will cause it to be changed to have the same contents as the original second file. In our simple example, where each file contains very little data, the change delta would probably be bigger than either file; in general, however, change deltas will tend to be a fraction the size of the files they were generated from. One other significant advantage of change deltas, and the reason they are used in source-code revision systems, is that they can often be applied to modified versions of files – see section A.1 for more details.

Note also that we could equally have wished to generate a change delta from the second to the first file, in which case the resulting change delta would look along the lines of ‘remove the line containing `2` between the lines containing `1` and `3`.’

3.2.2. Model change deltas

In our context, we are interested in model change deltas – in other words, change deltas which describe the difference between one model and another. The particular manner this will take is beyond the scope of this part of the document – conceptually, we assume that model deltas come in an easily parse-able standard form which are interchangeable between different tools. We anticipate that the model delta language will be fairly low-level to allow it complete flexibility.

4. Scenarios

We now outline a number of transformation transformation, and consistency related scenarios. To set the scene we first of all outline the ‘base case’ transformation scenario, where a model is taken as input by the QVT engine and a new model is output as in the informal figure 2. In such a scenario, the QVT engine need consist of only the transformation aspect.

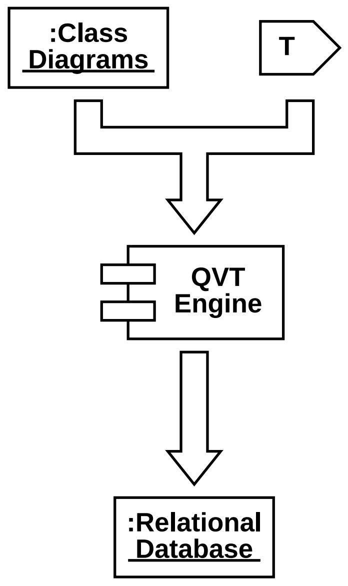

Figure 2: Transformation only

In the rest of this section we are interested in scenarios where both models already exist, and where separate tools manipulate the models, as in figure 3. For subsequent scenarios, we therefore start with an assumption that the system is in a state similar to that of figure 3 – in other words, that a central repository holds copies of models of class diagrams and relational databases which are accessed by tools A and B respectively, and that the QVT engine has access to both models within the repository.

Tools A and B are assumed to be model editors, that is tools which edit the models which they have access to in the repository. Although for simplicity's sake we work under the assumption that models are held by a central repository, it is quite possible to generalise these scenarios for situations where tools hold local copies of models.

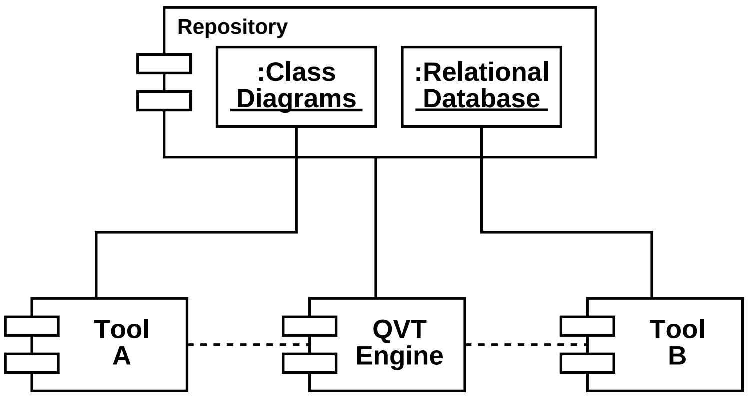

Figure 3: Tools with models involved with transformations

For completeness, using similar notation to figure 3 to re-express figure 2, we arrive at figure 4. Although it is difficult to satisfactorily express a mixture of static and behavioural concepts within a single figure, we hope that the intention behind figure 4 is relatively transparent. This figure shows tool A communicating to the QVT engine to invoke the mapping T to produce the relational database model in the repository. It is important to realise that the mapping T on its own can do nothing – it is only when it is run within the QVT engine that it can perform a transformation.

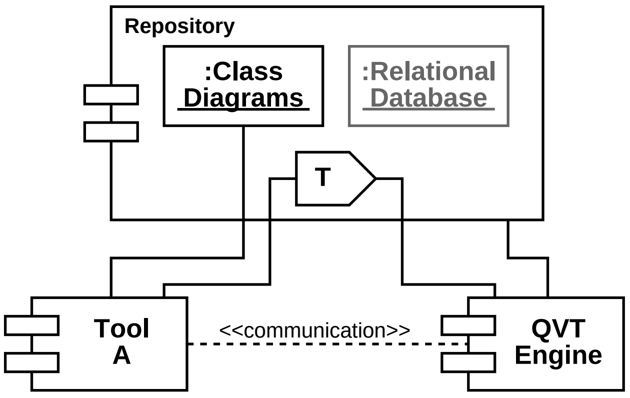

Figure 4: Re-expressing figure 2

4.1. Consistency check and report

Having in the previous scenario generated one model from another, the obvious thing to do at this point is to check that the new model is correct with respect to that which it was transformed from by ensuring that any transformation specifications are satisfied, as in figure 5. The intention here is that all the relevant transformation specifications in T are checked by the QVT engine, from which a consistency report is generated which is passed to the user. In the best situation, the consistency report will simply say ‘all specifications satisfied’. Otherwise, the consistency report should contain a list of all specifications which failed, and any relevant debugging output which may help the user to resolve the problems which have occurred. This extra output may take the form of narrowing down which part of a potentially large specification failed, giving the parts of either or both models which caused the specification to fail etc.

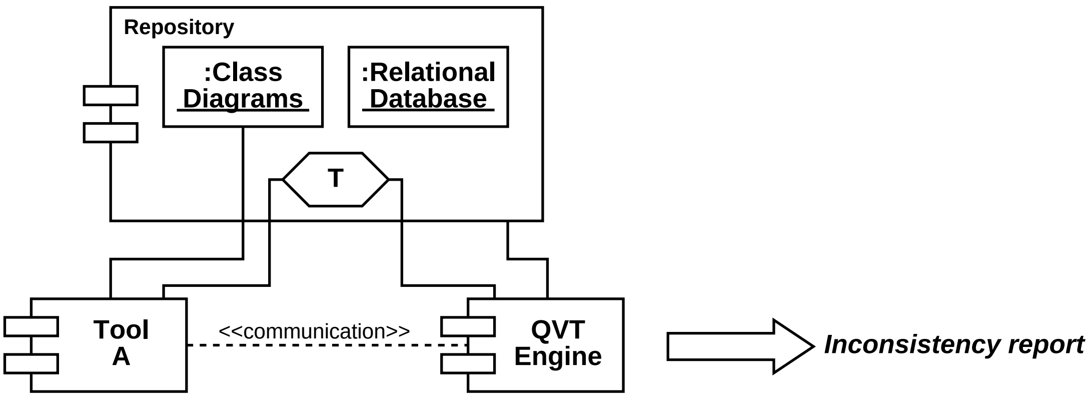

Figure 5: Checking two models for consistency

In order to bring out some of the dynamic aspects that are not well expressed in a static diagram such as that of figure 5, figure 6 shows a sequence diagram of the steps involved in consistency checking. Note that the accesses of the various models within the repository are drawn with grey lines to denote the fact that we are not particularly interested in exactly *how* they are accessed, but merely that they are accessed in one fashion or another.

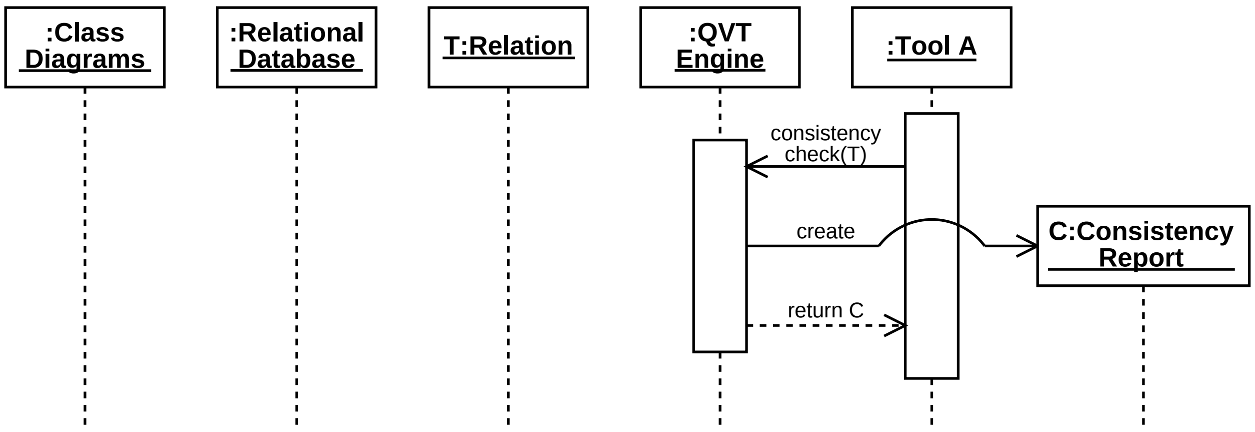

Figure 6: Sequence diagram for consistency checking

Notice that although we set up this scenario as having happened after the creation from scratch of the relational database model, consistency checking and reporting can happen at any point.

4.2. Changing a model

This scenario assumes that the system is in the state shown in figure 3 – that is, that both the models for class diagrams and relational models exist in the system, and their respective tools have links with them. The scenario involves tool A making a change to the class diagrams model. There are various ways that one can imagine this scenario being played out in practice.

4.2.1. Direct manipulation of model

The most direct way for tool A to change the model is to directly access the model within the repository. The QVT engine then has two main choices as to the action it can take:

1. It can use the repository's model history facilities to calculate an appropriate change delta between the changed model and the model before it was changed, and then carry on as for the scenarios in sections 4.2.2 and 4.2.3.

   Although this makes life easy for tool A, it raises the following issues:

   - For large models, the issue of scalability is likely to raise its head due to forcing both the QVT engine and repository to access and fully traverse two copies of a large model.
   - Because tool A has no knowledge of the delta language, it will only be able to indirectly request changes in the other model; it will not be able to update its own model if requested.

2. It can run a consistency check and report as in section 4.1.

4.2.2. Change delta generation and delta transformation invocation

Instead of tool A directly manipulating models within the repository, and expecting the QVT engine to automatically detect what changes have occurred, it can produce model change deltas (see section 3.2.2) which are then passed to the QVT engine. The QVT engine then attempts to match up the deltas with *delta transformations* which are mappings which take in deltas for one model and produce deltas for a second model. Figure 7 shows the sequence diagram for this. Since it makes no practical difference, we are deliberately vague as to whether tool A also modifies the class diagrams model or whether the QVT engine applies the delta it is given by tool A as well as using them to fire off the relevant delta transformations.

Note that there is nothing inherently special about delta transformations – they are simply transformations which, as a matter of convention, take in deltas and generate deltas. There is nothing to stop the delta transformations directly manipulating the other model rather than producing deltas which will update it. However to do so would be to miss out on some of the advantages of this scheme.

There are several advantages to delta transformations:

- They are efficient in that they only deal with relatively small change deltas which, by definition, have already performed most of the necessary analysis of one model.
- They allow efficient implementation of change propagation.

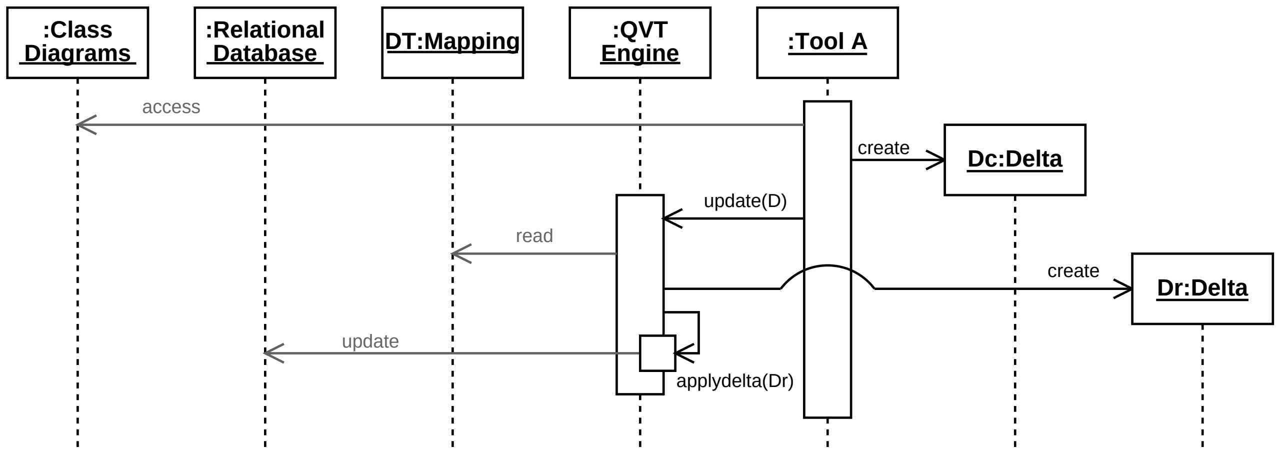

Figure 7: Sequence diagram for change delta generation and delta transformation invocation

- They allow change deltas to be easily passed around the system. In terms of our running example, this means that delta transformations can not only update the relational database model but can also be transmitted to tool B to inform it of what particular changes have been made (see figure 8), allowing tool B to perform various actions such as flagging changes to the user.

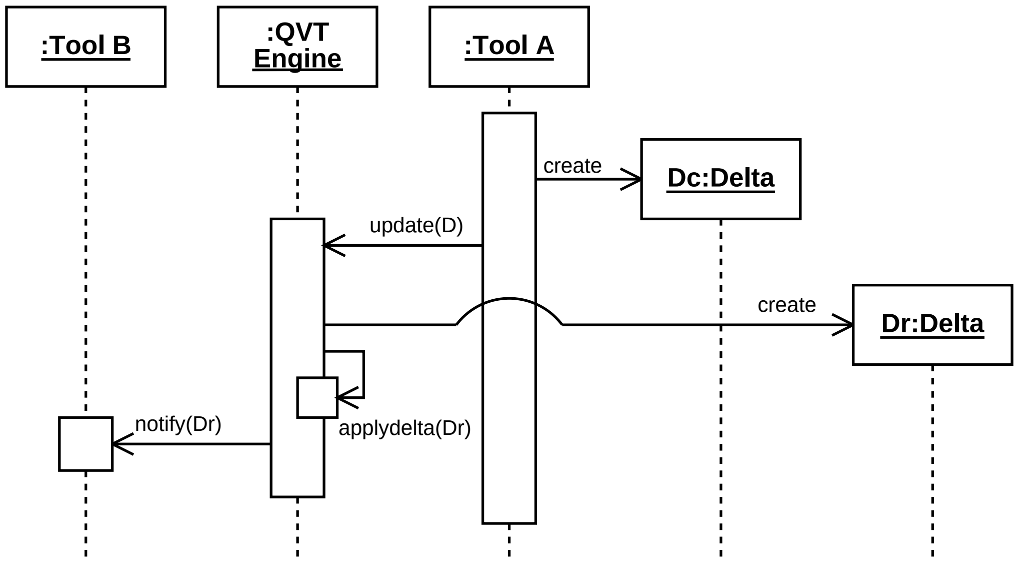

Figure 8: Using delta transformations to provide notification of changes

4.2.3. Change delta generation and automatic update

The idea behind this method is ambitious, and not applicable in the general case, but may have practical uses. Essentially rather than handing over change deltas to delta transformations, the QVT engine would attempt to reason about how a particular delta needs to cause a change in the corresponding model. The most obvious way this can be achieved is by the analysis of transformation specifications.

5. Conclusion

Although this document is not meant to be prescriptive, we nevertheless have the following recommendations to make:

- We believe the architecture of change delta generation and delta transformation invocation described in section 4.2.2 holds the promise of being a practical solution to the model consistency problem.
- An underlying assumption for most of the scenarios presented is that there is an agreed mechanism for inter-tool communication. This needs to be explored further, particularly to see if any existing work can be utilised.

A. Deltas

A.1. Change deltas

A.1.1. Textual change deltas

Change deltas denote the change between two different things. Directly or indirectly, most readers of this document are likely to be fairly familiar with change deltas produced by the Unix `diff` program which when given two text files produces output which shows how to change the first file into the second file. So for example given the following two files:

```
1
3
```

and:

```
1
2
3
```

`diff` will produce output along the following lines:

```
--- a   Wed Sep 24 12:10:28 2003
+++ b   Wed Sep 24 12:10:35 2003
@@ -1,2 +1,3 @@
 1
+2
 3
```

which effectively says ‘place a line containing the character `2` between the lines containing `1` and `3`.’

A.1.2. Context information

The change deltas produced by `diff` contain more information than might at first appear to be necessary – the extra information is context information, which allows the change delta to be applied to modified versions of the first file and still produce the expected result. It is unclear as to whether such a facility is required by any of the solutions we outline in this document, although such a facility would undoubtedly facilitate more complex scenarios such as two tools simultaneously updating different parts of the same model.

A.1.3. Change delta styles

Although there is often an intuitive change delta from one object into another, there are in general an infinite number of change deltas which will produce the expected result. For example[3] imagine the strings `ab` and `acab` – human intuition says that to get from the first to the second string one would simply prepend `ac`. The `diff` algorithm on the other hand internally produces a change delta which says that one should add the string `ca` between the `a` and `b` characters. Both change deltas are perfectly valid, but it may well be that one style of change delta is preferred over the other – indeed, some `diff`-like tools go out of their way to produce change deltas which are more comprehensible to humans than those produced by `diff`. Conversely, many `diff` implementations have a switch to produce a minimal change delta; whilst one might expect the lack of extraneous information to be beneficial, such a change delta is rarely comprehensible to humans. We believe that the algorithms used to produce model change deltas may have a significant usability impact in many of the solutions we outline in this document.

[3]This example is taken from a comment within Tim Peter's `ndiff` program.