# GMT Framework

Edward D. Willink, EdWillink@iee.org
28 November 2003

1 Introduction

The following discussion document is prompted by encouragement from discussions with a number of people at OOPSLA 2003 in Anaheim, and at Metamodelling for MDA 2003 in York. Comments are invited via https://dev.eclipse.org/mailman/listinfo/gmt-dev.

It is clear that the Eclipse Generative Model Transformer project should not favour any particular transformation technology; rather it should provide a framework within which any transformation, transformation technology, and as importantly any inter-transformation model representation can coexist.

GMT can therefore align very naturally with the Eclipse plug-in philosophy by providing the GMT framework as one or more Eclipse plug-ins into which specific transformations or transformation technologies are themselves plugged in, with plugged-in transformations also providing the conversions between model representations.
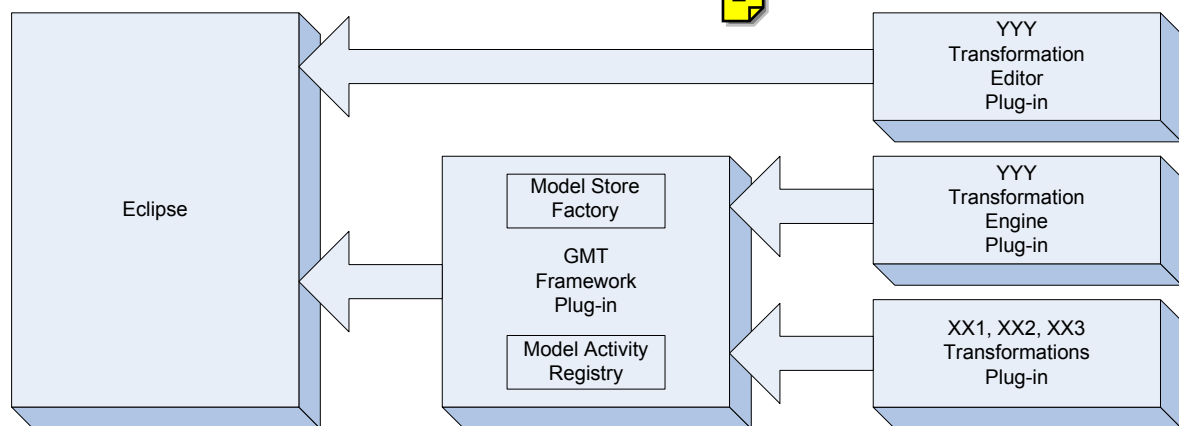
The representation of a model is managed by a model store created by the Model Store Factory that maintains a registry of representations supported by plug-ins.

Transformation of a model is performed by a model activity that a plug-in has installed in the Model Activity Registry.

2 Concepts

Transformations operate between instances of meta-models, which constitute the type definitions of the input and output models. We therefore need a mini-language to define the model type; the way in which the model representation encodes the instance of a meta-model.

2.1 Model Type Name

A model type may be defined by an identifier such as `EjbJava`, or a string such as `"EJB aligned Java"` if a more descriptive name is required. In BNF:

```
ModelTypeName := Identifier
              | String
```

2.2 Model Representation

There are major problems with standardisation of model representations. Even within one standard such as XMI, there are two significantly different versions, and major issues of tool

compliance. It is therefore inappropriate to consider imposing any standardisation within GMT. The lack of standardisation can be resolved by the concept of an encoding.

When the concepts of a particular `EjbJava` meta-model are defined using EMF and then serialised as XMI2, the resulting model may be described as instantiating the `XMI2<EMF<EjbJava>>` model type.

In general a model type is either a meta-model name, or an encoding of a model type, or an alias for a model type. An alias, like a C typedef, allows e.g. `XMI2<EMF<EjbJava>>` to be referred to by a more convenient name such as `XmiBeans`.

```
ModelType := ModelTypeName
           | ModelEncodingName<ModelTypeList>
           | ModelTypeAlias

ModelEncodingName := Identifier

ModelTypeAlias := Identifier  // referencing a ModelType

ModelTypeList := ModelType
               | ModelTypeList ',' ModelType
```

2.3 Model Versioning

A disciplined modelling tool will provide version control of both directly and indirectly referenced meta-models, so a simple string will not suffice as an identity; an instance of an identity meta-model may be required. This may be accommodated by defining a Version encoding.

```
Version<ModelType, VersionModelType>
```

This need not be implemented in the first release, and can co-exist with

```
OmgVersion<ModelType, OmgVersionModelType>
```

if the OMG produce a suitable version control standard.

2.4 Non-functional properties

Properties such as robustness or security are difficult to represent within conventional programming domains. They too can be accommodated by defining an encoding such as:

```
Robust<ModelType, RobustnessModelType>
```

This incorporates the robustness requirements within the model-type system, but does nothing to directly enforce these properties. However with the robustness requirements in the type system, only transformations that support the robustness encoding can be invoked.

2.5 Summary

In a Java environment, we may hope to make consistent use of perhaps `JAXP<MOF2<T>>` model-type. However the generality of the encoding concept allows us to handle a CVS maintained, encrypted, Poseidon XMI serialised, MOF 1 model of an EJB Java system in a consistent fashion:

```
CVS<Encrypt<PoseidonXMI<MOF1<EjbJava>>,EncryptionModel>,CVSIdentityModel>
```

A name such as `CVS` as a model encoding name forms no part of the GMT framework. The name is plug-in-defined and useable provided some plug-in registers consistent transformations that support the additional encoding. Therefore, when it is discovered that AnotherTool has yet another variant on XMI, the problem can be resolved by providing a plug-in to transform from `AnotherToolXMI<T>` to `XMI<T>`, where *T* is an arbitrary model-type. We define such a transformation as having the signature:

```
AnotherToolXMI<T> -> XMI<T>
```

This may be provided by either the vendor of `AnotherTool` or by its more demanding users.

3 Transformation

In principle, a transformation accepts instances of one or more input meta-models and generates instances of one or more output meta-models. In practice, each of these instances must be stored somewhere with a suitable encoding.

A model activity therefore retrieves its input instances from source model stores, and generates its output instances in destination model stores. Until there is broad agreement on the encodings and meta-models, use of transformations from a diverse community will encounter significant incompatibilities. These can be resolved automatically, if the model store can perform a conversion to suit the accessing transformation activity.

For example if the input model store contains an `XMI<EMF<EjbJava>>` but the transformation requires `JMI<MOF2<EjbJava>>`, the required conversion may be performed by successive invocation of transformations with signatures

```
XMI<T1> -> T1                  // File reader
EMF<T2> -> MOF2<T2>            // Meta-model language conversion
T3 -> JMI<T3>                  // JMI formatting
```

or more efficiently by

```
XMI<T1> -> JMI<T1>             // File reader direct to JMI
JMI<EMF<T2>> -> JMI<MOF2<T2>> // Meta-model language conversion
```

In order to accommodate any legacy or hand-optimised transformation, GMT can invoke each transformation via a Java class that implements `IModelActivity` and which accesses its inputs and outputs via Java classes that implement `IModelStore`.
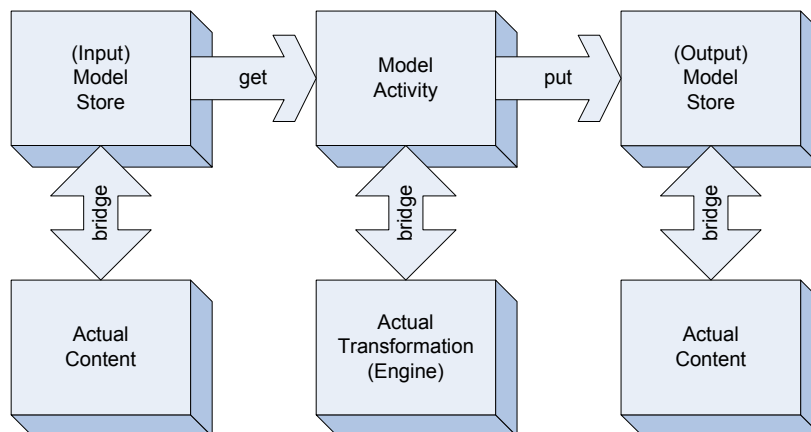
A new transformation may therefore be coded directly by implementing `IModelActivity`. A piece of legacy code, or new code that avoids dependence on the GMT framework, may create a bridge from the GMT framework, by ensuring that the bridge implements `IModelActivity`. Transformations invoked via bridges can be implemented using any programming language, either by using JNI to activate non-Java code, or by starting a separate process for the foreign language program.

As we move towards QVT, we want to use programmed rather than dedicated transformations, so we create a bridge from the GMT framework to a specific transformation engine in just the same way. We create a bridge that implements `IModelActivity`, and which acquires its suitably encoded transformation program from a program input model store. We may look forward to a day when this will always use perhaps a `XMI2<QVT<T>>` encoding. Until then we must accommodate e.g. `XMI1<MyQVT<T>>`.

Once QVT is established we may seek to rescue legacy transformations by plugging-in the

```
MyQVT<T> -> QVT<T>
```

transformation, and continue to use legacy transformation engines by plugging-in

```
QVT<T> -> MyQVT<T>.
```

3.1 Automated Transformations

Conversion of model encodings when accessing model stores has been identified as amenable to automatic invocation of the appropriate one-input, one-output transformation that performs the conversion.

There is scope for taking this automation considerably further. In an MDA context, the PM may impose such strong constraints upon the required output model type, that a transformation sequence that progresses from the model type of the PIM can be computed. In this case, the explicitly invoked transformation may have almost null functionality, with all transformations being performed as part of the conversion from the actual model type of the PIM at input to the target model type defined for the PSM.

4 Interfaces

4.1 IModelType

The `IModelType` interface provides just the re-entrant naming policy by which representations are identified. It does not provide any further modelling functionality, so we may find that some very simple concrete types are adequate, with functionality suggested by the construction signatures:

```
ModelTypeName(String)
ModelTypeAlias(String, IModelType)
ModelTypeVariable(String)              // e.g. T
ModelEncodingName(String)              // e.g. XMI
EncodedModelTypeName(ModelEncodingName, IModelType[])
```

(There is no need for a specific `XmiTypeName`, since these classes define meta-type names. It is the instance of `ModelEncodingName("XMI")` that identifies XMI.)

4.2 IModelValue

This interface is a bridge to a specific model representation. The bridge may take the form of the name of an XMI file, a reference to an actual model such as an `org.w3c.dom.Node`, or equivalent mechanisms for access to EMF models, CVS or Rational Rose repositories.

When an `IModelValue` is used, the `IModelType` should be known so that it is safe to cast and use derived methods

```
MyModel myModel = ((BridgeToMyModel) iModelValue).getModel();
```

4.3 ModelStoreFactory

The `IModelStore` interface defines the protocol for model store bridges to model values. Concrete implementations are registered with the `ModelStoreFactory` by

```
ModelStoreFactory.add(ModelEncodingName, Class);
```

A concrete `ModelStore` is then created by

```
ModelStoreFactory.newStore(IModelType type, IModelValue value);
```

in which the outer `ModelEncodingName` of `IModelType` indexes the `ModelStoreFactory` to identify the concrete `IModelStore`.

4.4 IModelStore

Assignment of a source model store to a target model store

```
IModelStore.put(IModelStore source)
```

may provoke an automatic translation from the representation type of the source to that of the target.

The bridge to the model content may be obtained by

```
IModelValue IModelStore.get();
IModelValue IModelStore.get(IModelType type);
```

the latter form potentially also provoking an automatic translation from the representation type of the source model store to the specified type.

4.5 IModelActivity

This interface comprises the two methods for executing transformations:

```
IModelActivity.run(IModelStore[] inputs,
                   IModelStore[] outputs) throws Exception;
```

is used to invoke a fixed purpose transformation.

```
IModelActivity.run(IModelStore program,
                   IModelStore[] inputs,
                   IModelStore[] outputs) throws Exception;
```

is used to invoke a general purpose transformation engine.

In each case the transformation activity returns on completion.

Exceptions are thrown only for conventional problems, such as class not found, index out of bounds etc.

I have not noticed any discussion of exceptions or errors in the QVT proposals, so the following is a provisional proposal on how to handle them. Problems within the modelled domain are returned (thrown) through one of two implicit `sync` and `async` outputs. The `sync` output is suitable for an instance of an error message meta-model that accompany a complete execution of the transformation activity, other outputs are therefore valid. The `async` output is suitable for an instance of an exception meta-model, in which case transformation execution may have terminated abruptly and inconsistently and so only the `async` output is defined. In a hierarchical transformation context, `sync` and `async` are implicitly merged (caught) and propagated (rethrown), unless explicitly connected (caught). This is very analogous to automatic propagation of a throw in Java or C++. Unhandled `async` outputs are ultimately handled (caught) by the framework.

4.6 ModelActivityRegistry

The model activity registry maintains a mapping from a known transformation signature such as:

```
XMI1<T> -> XMI2<T>
```

to the derived `IModelActivity` that provides (the bridge to) the implementation of the transformation. These implementations will be provided within plug-ins, so a little care is needed to ensure that the signatures can be registered without compromising lazy plug-in loading.

5 Satisfaction of the GMT SRS

5.1 Transformation Component

The ability to perform transformations constitutes the transformation component.

Defining this as a plug-in that supports registration of arbitrary transformations and transformation engines goes further than envisaged.

Requiring that all transformations be plugged in means that the GMT framework alone is unable to perform transformations. This is much less to get going, and does not impose any GMT prejudice as regards good/bad transformation philosophies.

Providing example transformation plug-ins, ensures that the GMT framework is at least useable

5.2 Mapping Component

Mapping unfortunately has an ambiguous meaning.

In the mathematical context of a mapping from one type to another, registration of a single input, single output transformations such as

```
XMI<MOF<MySimplifiedUML>> -> XMI<MOF<YourSimplifiedUML>>
```

provides the support for automatic transformation between at least closely related model types.

In the modelling context of a mapping between PIM and PM under control of a Mark Model to produce a PSM, libraries of transformations with signatures such as

```
XMI<MOF<PimModel>>,XMI<MOF<MarkModel>>,XMI<MOF<PmModel>> ->
                                                   XMI<MOF<PsmModel>>
```

constitute the mapping component.

5.3 Text generation component

Any transformation that supports a signature such as perhaps

```
XMI<MOF<JavaModel>> -> Text<JavaModel>
```

constitutes part of the text generation component, and of course a transformation with a signature such as

```
Text<JavaModel> -> XMI<MOF<JavaModel>>
```

is a Java parser or inverse text generator.

5.4 Workflow component

There is no fundamental difference between a workflow and a transformation, just a matter of perspective. A workflow may involve very different coarse grained activities such as EditJavaCode or CompileJava between dramatically different meta-models, whereas a transformation may involve finer grained activities such as FlattenStateMachine or GenerateGetters where input and output meta-models are almost identical.

Workflows can be supported by a much simpler transformation technology than QVT, but this does not preclude the use of QVT to define a workflow, and so the ability to define any transformation technology as an `IModelActivity` also supports a plug-in for a transformation technology optimised for workflows. Similarly the ability to define the encodings of models stores allows a workflow defined in say XPDL to be accessed as perhaps `XPDL<MyWorkflow>`.

In order to invoke any transformation or workflow, it is necessary to bind the inputs and outputs to suit the users requirements. This is itself just a very simple transformation language that should be installed as yet another transformation plug-in. Its simplicity lends itself to a simple XML/form-like GUI that could have similarities to a simplified PDE plug-in interface. (I have been developing a prototype of this based on the PDE plug-in as a way of becoming more familiar with Eclipse.)

The workflow component is therefore supported by the various transformation technologies, of which those that provide a simple user interface enable friendly activation. Provision of transformations that provide the bridges to major tools such as editors and compilers enable useful workflow applications to be realised.

6 The Model Bus

While model editing, compiling and execution are important, we should support much more, particularly debugging, optimisation and analysis.

How to debug highly optimised declarative transformations is a topic for research, so we should ensure that our support is flexible.

Perhaps an `IModelActivity` or `IModelStore` has an accompanying `IModelListener` interface while an `IModelActivity` also has an `IModelActivityController` interface. The listener should at least enable announcements of progress such as transformation start and finish to be detected. The controller should support operations such as start/stop/step/resume.

I have no experience of implementing such tools, so I suspect that there may be significantly more flexible and powerful solutions.

It would seem that these interfaces could all be added later, however at least a preliminary attempt at `IModelListener` could support intelligent tracing (and automated testing) from the outset

7 Implementation Stages

The framework discussed above is very small and extensible. In addition to the flexibility that it offers to its users, it makes for a very small amount of core code to which highly orthogonal incremental contributions can be made by a community of largely independent developers.

7.1 Basic framework

`IModelType` and five concrete derivations provide just naming functionality, so these are small.

`IModelValue`, `IModelActivity` and `IModelStore` are small interfaces.

`IModelListener` and `IModelActivityController` are again just interfaces.

`ModelActivityRegistry` and `ModelStoreFactory` require real code but they do not amount to much more than dictionaries and a plug-in registration protocol.

7.2 Core plug-ins

A few variants of `XmiModelValue` are essential, with `EmfModelValue`, `FuutjeModelValue`, `UmlxModelValue`, and `WfcModelValue` following close behind. EMF possibly provides the XMI functionality anyway.

7.3 Transformation Component

This is where everyone can add their own transformations and (prototype) transformation languages. Just to get started:

7.3.1 Invocation

A PDE-like plug-in is in progress to support definition of the parameterisation of a workflow.

7.3.2 Execution

A UMLX plug-in should be able to support definition and execution of a workflow without much further development. Significant development is required to make UMLX fully functional.

7.3.3 Wizards

Extensible wizards should assist in creating plug-ins containing bridges to existing transformations.

7.3.4 QVT

We hope that one or more of the competing QVT submissions will be available and integrated into GMT via plug-in bridges, with some of the core technology such as OCL available via direct Eclipse plug-ins.

7.4 Text Component

Traditionally transformations have covered a significant distance between input model and output text. Such transformations can be installed without difficulty in the GMT framework.

Fuutje and EMF have transformations that can be accessed via bridges.

With an ability to sequence model-to-model transformations, the complexity of model-to-text transformations may be reduced. There are a variety of template languages that should be easily installed as simple model-to-text technologies.

7.5 Workflow Component

Some degree of workflow support is provided by the transformation component. More interactive components, such as prompting for user authentication or editing may be installed as transformations.

7.6 Mapping Component

Herein lies the next 20 years of work to support MDA via better and better transformations between yet-to-be-standardised meta-models. Hopefully GMT provides the framework in which this can be a very broad collaboration.

7.7 Modelling Tool

A (UML) modelling tool has always been seen as a tool that GMT exploits and extends, rather than redevelops. As the use of disciplined meta-models underlies QVT approaches, a visual MOF meta-modelling tool will be essential, and preferably one that is well-integrated with Eclipse.

When inadequate Eclipse modelling support becomes a serious issue, this should probably be addressed in a wider context than GMT.

8 FAQ

8.1 How do I define a meta-model?

Meta-models may be defined using any supported language. The basic GMT framework has no built-in meta-modelling languages or meta-models, but plug-ins for widely used languages and representations such as MOF, EMF, XMI will be available. Meta-models for standard languages such as UML should be available in at least one of the widely used formats.

Custom languages may be supported by developing plug-ins.

Custom meta-models are supported by the textual or visual editor for a supported meta-modelling language. Eclipse provides a text editor, which plug-ins may enhance with syntax colouring and checking. Eclipse support for visual editing is limited and so an external tool may be required.

8.2 How do I invoke an installed transform/workflow?

Create a new workflow configuration (via a wizard), or edit an existing configuration. The workflow configuration may use any transformation language able to specify the transformation, its parameterisation, and its input and output resources.

The GMT/WFC (workflow configuration language) plug-in will support editing a *file*.`xmi-wfc` in a similar fashion to the PDE editing of `plugin.xml`.

Then `Run` it.

8.3 How do I support my non-standard file format?

The basic GMT framework has no support for any file format. Each format is defined by a plug-in. You may therefore use one of the supported formats as an example in developing a plug-in for your own format. A wizard may eventually be provided.

8.4 How do I support my non-standard meta-model representation?

If your transformation uses less-standard model representations that GMT does not support, you must either rewrite your transformation to use more-standard model representations, or register further transformations between your less-standard and a more-standard form. These transformations may be installed as fixed or generic transformations.

8.5 How do I install my fixed functionality transformation?

The basic GMT framework has no built-in transformations. All fixed function transformations are registered from a plug-in. You may therefore use one of the transformation plug-ins as an example in developing a plug-in to register bridges to your own transformations. An extensible wizard should be provided for bridges with standard calling conventions.

8.6 How do I install my generic transformation technology?

Minimal support may be added as for fixed functionality, the registered bridge just acquires your transformation program as an additional input.

Progressively improved support may involve additional Eclipse plug-ins emulating the Eclipse JDT support:

A syntax colouring (text) editor plug-in.

Interactive syntax checking and intelligent correction.

Debugging.

In principle, the same support is also required for a visual transformation syntax, however creation of a visual editor within Eclipse is hard. Until more flexible graphical support is available, it may be necessary to support visual editing via a bridge to another tool such as GME.