From the Workfloor: Developing Workflow for the Generative Model Transformer

Ghica van Emde Boas
Bronstee.com
the Netherlands
emdeboas@bronstee.com

ABSTRACT

This document describes an experiment with the purpose of developing a proof-of-concept for a prototype for a workflow component for GMT, the Generative Model Transformer Open Source effort. It discusses the challenges we met in trying to adhere to standards and the problems of using existing "wizards". Despite this, generative development proved to be effective.

Keywords

Generative techniques, Java, UML, XML, XMI, Model-Driven Application development, Workflow.

1. INTRODUCTION

Model Driven Architecture (MDA) is an initiative by the OMG to leverage UML-based modeling techniques. The idea is, that we should be able to define domain specific models and platform specific models separately. By using model-driven generative techniques, or by using executable models, applications will be built.

It is surprising that the word **"Architecture"** is nowhere explained in the MDA literature. It is unlikely that the term refers to *application architecture*: presumably this architecture does not depend on the fact whether the application was developed using MDA. Maybe an architecture that is *model driven* hints to an attempt to define the architecture of the development process itself, or of the tools that are an essential part of MDA.

The developer using an MDA development process will construct, or reuse, a set of models. By a series of transformations applied to these models, a deployable, executable model will be produced or code in a traditional programming language will be generated. The *architecture* of this development process, and the tools that implement it, s of major concern to the MDA community. How can we define an *MDA-component* and how will these components fit together to define the **A** of MDA?

Several tool vendors have started to bring MDA tools to the market. These tools mostly look like a next-generation CASE tool: monolithic and closed world. Although they may be excellent tools, the question is, whether there are other possibilities.

1.1 The Generative Model Transformer

At past year's OOPSLA, a BOF session organized by the author and Jorn Bettin discussed the necessity of tools for the MDA development process. This resulted in an open source initiative that is now part of www.eclipse.org (see ref. [1]).

The goal of the Generative Model Transformer (GMT) tool project is to construct/assemble a set of tools for model-driven software development with fully customizable Platform Independent Models, Platform Description Models, and Refinement Transformations.

GMT's initial requirements document states the following:

At this stage we envisage four main components:

1. *A* **mapping component** *that can combine two XMI-encoded models into one new XMI-encoded model.*
2. *A* **model transformation component** *using XMI as input and output.*
3. *A* **text generation component**, *using XMI as input and text (code) as output.*
4. *A* **workflow component** *that provides the required glue between the three functional components above, any additional user-developed MDA tool components, and popular IDEs/tool platforms such as Eclipse.*

Any component that fulfills the basic requirement of allowing XMI input should be usable as a MDA tool component in GMT.

The requirements document is still under discussion, as is the place of workflow within these requirements. The following description of workflow and how it is intended to be used should be considered as the authors contribution to this debate.

Note that the role of the models themselves is merely data, as their encoding in XMI already suggests.

Theoretically, the mapping- and text generation components are a particular form of model transformation component. Practically, we distinguish between them.

Text, or *code generators* can be found already. The GMT team hopes to reuse one instead of developing a new component. *Model transformators* will be the next components to appear. Our hope for GMT is that UMLX [13] will fullfil this role within the tool, and maybe others later. The shape of a *mapping component* is still to be defined.

The *workflow component* is an entirely different kind of component. It is there to allow the transformations to execute in the right order, with the right input, under the right conditions. The scope of this paper is a discussion of this workflow component.

2. THE WORKFLOW COMPONENT

Let's start with answering the question what workflow is, and why it is important for GMT.

2.1 What is Workflow?

The Workflow Management Coalition [5] defines workflow as:

"The computerized facilitation or automation of a business process, in whole or part."

Workflow Management Systems are already popular for many years within banking and insurance.

A characteristic of an environment that can benefit from workflow, is the presence of many different tasks and activities, where information must be passed between these according to a predefined set of rules.

2.2 Workflow within GMT

The workflow within GMT is intended to help two different sets of users of GMT as a tool:

1. The *developer of an MDA component* that should participate in the GMT tool set.
2. The *developer of an Application* using the GMT as a tool.

Disregarding the developer of MDA components for the time being, let us assume that the GMT has an extensive set of components that can do model transformations and code generation. The application developer who wishes to use model-driven, generative techniques, would choose a modeling tool, a series of transformation components and one or more generation components.

The choice of particular components and order in which the developer applies the use of each of these components determine the shape of the resulting application, *but not its domain contents*. If this developer want to re-iterate his development process, because he want to change or extend his application, or because he wants to develop another application for a similar platform, then he probably wants to use the same *workflow,* using the same MDA components.

Figure 1 below, is taken from the GMT requirements specification [2]. The dark ovals are models, the arrows show transformations. The sequence of the transformations, and the choice of models, define the development process that the developer of MDA applications wants to follow. It is outside the scope of this paper to discuss this picture in more detail.

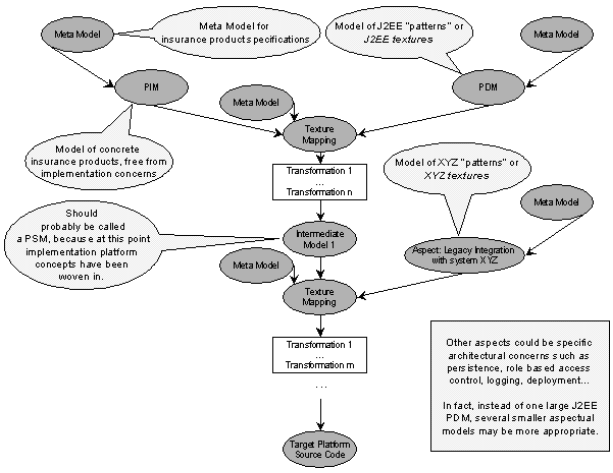

Fig. 1. Informal MDA development flow in GMT

Looking at the process description of GMT, we see that the workflow definition applies precisely to the way we envisage that GMT should be used by application developers.

Let's describe an example that can be done already now with what is available for the GMT tool-set.

Our application developer would like to develop yet another version of our famous *AddressBook* application in a model-driven, generative way. The resulting application should have a simple Java-Swing GUI, and the contents of the addressbook should be saved in a relational database.

Using our experimental, prototypical, GMT workflow implementation, the intention is that the developer can define a workflow that looks roughly as follows:

1. Start the GME tool and define a domain-analysis class model in UML. Export the resulting model as XMI file.
2. Use a UMLX transformation to transform the model to a relational model .
3. Use the FUUT-je tool and a database access code generation template to generate JDBC code and table definitions.
4. Use a UMLX transformation to transform the model to a Swing model .

5. Use the FUUT-je tool and a Swing code generation template to generate Java code.
6. Import all code into Eclipse to make final modifications and add manual code where needed.
7. Build the application using ANT.

The GMT workflow component will orchestrate the cooperation between components that can participate in GMT. These components may be applications that were not designed for cooperation within GMT. The workflow component will make it possible that these components are capable of cooperating effectively and in a loosely coupled way.

By having such a workflow component available, we think that GMT can have a head-start: GMT will be able to utilize quickly other functionality that is already there.

For this reason it is the workflow component that ended on top of the priority list of things to develop [2], although workflow does not seem to be a core subject domain for GMT.

Because the workflow component is intended to be a core part of GMT, there are these major requirements:

- The resulting workflow component must be open-source and fit within the Eclipse project [3].
- Adhere to standards where possible
- Use model driven– generative methods for development where possible.
- Develop a simple first version rapidly, that can bootstrap further GMT development as quickly as possible.

2.3 Choosing the Software to Use for Generative Development

We are trapped in a recursive argument here. GMT would be an ideal tool to help develop the workflow component. However, GMT is misses essential function if we do not have a suitable workflow component. The question is, how we best can develop GMT before GMT is ready.

For now, we will have to do with a piece of "good enough" software: FUUT-je[1]. This is a fairly simple, but effective model driven code generation tool, that was donated to the GMT project by the author of this paper.

Of course we looked around to see whether we could find an open-source, Java, workflow component. Actually, we found one: "Open for Business" (OFBiz) [4].

[1] An alternative would have been to use EMF, the Eclipse Modeling Facility. For now, EMF seems less suitable, because we cannot change it's meta-model and because it seems hard to write your own customized code generation.

We do not have a good excuse for not using it. For the time being, I consider OFBiz to be too large and too ambitious for our limited purpose. We may later interoperate with it, because OFBiz uses the same process definition language called XPDL.

2.4 The WfMC Standards

We did find a set of standards that we would like to comply with: the *Workflow Management Coalition* Workflow Standard [5].

Trying to develop software that complies with a comprehensive standard as defined by the WfMC brings another dilemma: how do we avoid scope creep? If we implement the full WfMC standard, our final workflow product would have much wider applicability than just GMT and it could become such a time consuming activity that the development of other GMT components would suffer.

No easy solutions exist for software development challenges. The other side of the coin of using a comprehensive standard is that it is mature. For example, the WfMC has a well-defined *workflow process definition language*, for which an XML Schema is available. See "XML Process Definition Language" (XPDL) [6]. Because it should be straightforward to interpret an XML Schema, we decided to use it as a starting point.

2.5 Architecture for the GMT Workflow Component

A workflow service that adheres to the WfMC standard will consist of two parts: First the user will define the workflow processes, the activities in each process and the conditions the should be met for the activities to be started.

When the workflow definition is available, an *Enactment Service* will make sure that the activities are executed in the right order under the right conditions. It follows that our development comprises two distinct tasks:

1. An editor for the workflow definition
2. An Enactment Service

Initially, we will support only a subset of the XPDL and write only a simplistic enactment service.

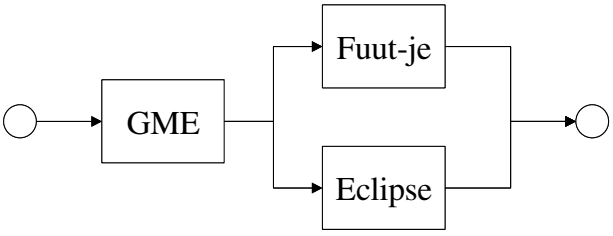

Fig. 2. Sample Workflow Definition

The simple workflow that we should be able to execute in the first demo of our prototype looks as in fig. 2.

The next sections will describe the development of the XPDL Editor and the Workflow Enactment Service.

3. XPDL EDITOR

The messages sent between activities participating in the workflow will be in a format that is defined by the XPDL Schema. Therefore our XPDL editor should be able to read and write XML messages in this format.

Preferably it should have a front-end that will show and interact with pictures like in fig. 2. A crude first implementation could be to just use notepad or a generic XML editing tool. Since the Enactment Service should be able to read the definition also, we chose to develop slightly more usable software, that would be able to read an XPDL message into a suitable Java object structure, and have simple Java Swing entry-forms as a front-end that could be replaced by a fancy one later.

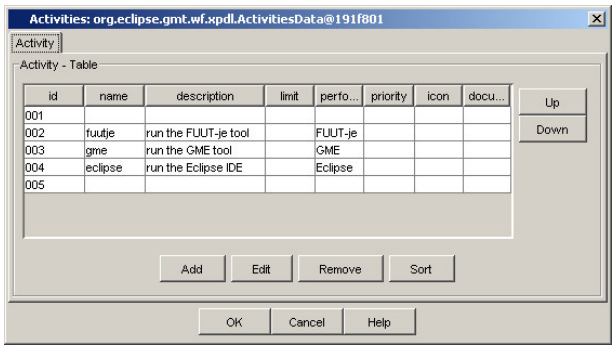

Fig. 3 – Simple Java Swing XPDL data entry form

Our best hope for a standardized way of binding XML schemas to Java representations would be to use JAXB, the Java™ Architecture for XML Binding [8]. Sun has a binding compiler wizard available that, with some work, produces a bag full of Java classes that can be used to marshal and un-marshal the content of a workflow definition specified in XPDL into and from a tree structure of Java objects.

To be able to do *model-driven* development, we should somehow surface the model behind the XML-schema.

3.1 XPDL, it's XML Schema and XMI

An XML Schema is a description of an XML data structure that can be very detailed. The XPDL schema is the definition of the XPDL language structure.

Various approaches exist to bind XML that is defined by an XML Schema to a set of Java objects. One of the better known approaches is JAXB, developed by Sun. Not every XML Schema can be transformed to a set of Java classes, notably those that describe books will not be suitable.

If we have a set of Java classes, we could abstract these to a set of UML model classes. This means that we should be able to express the XPDL language as a UML meta-model and serialize it as XMI.

Fig. 4. Flow of data for transforming XML-Schema

On the other hand, it should be possible to transform XML-schema to XMI directly using XSLT, and then use one of the UML tools to obtain UML. Once we have UML, we can use code generation templates to generate Java classes to interface with the JAXB produced classes.

The flow of transformation needed for our development is shown in fig. 4. Since JAXB is rapidly being adopted as a standard for XML Schema binding to Java classes, *using a standard JAXB compiler and XML Schema -> XMI -> UML -> Java, should be our objective.*

For practical purposes, because we wanted to have a stake in the ground for the GMT workflow as soon as possible, I have taken another approach. There were two reasons:

1. I could not find a tool or XSLT script to transform the XPDL schema to suitable XMI close to JAXB bindings.
2. It was unclear to me how to develop the piece of code that should connect the Java objects as generated by the JAXB wizard with the classes that would come out of a Platform Specific Model (PSM) in UML.

And of course the author is very familiar with the Fuut-je tool ☺. As a result we have decided to use this tool.

Fuut-je has built-in XML facilities, with a very similar purpose as JAXB: easy generation of Java code that can read and write XML files of a particular structure. Fuut-je is certainly not JAXB compliant (it was developed before JAXB became known). It can however generate code that can read XPDL into a suitable Java structure, and write out an XPDL file again. The Java Swing GUI for the XPDL object structure would come for free.

One problem: Fuut-je did not support XML Schemas. If… we would have a DTD as the XPDL definition we could use that instead.

3.2 Developing XML-Schema Support within Fuut-je

As described, XPDL is defined in an XML Schema. Therefore it could be worthwhile to look at providing support for XML Schema's within Fuut-je.

The first option I looked at was the possibility of parsing schema's. The newer Xerces XML parsers have this possibility. After looking at this for a while, I considered it to be too complex for my purpose.

A bit more research, and *the insight that an XML-schema is an XML document that is defined by a schema, where this schema has a … DTD,* led me to a stepwise approach.

Maybe a little more explanation is appropriate. A DTD is a simpler and a less expressive definition of an XML structure. Just like JAXB can bind an XML Schema to a set of Java classes, a DTD can be bound to a set of Java classes. The abstraction of this set of classes into a UML model is what Fuut-je can produce by reading a DTD.

From this Fuut-je model, the tool can generate Java code that can read and write XML that complies with the DTD. A simple Swing editor used to create the XML content data is also generated.

Once the DTD of the XML schema of schema's was read into Fuut-je, we could generate code to read and write XML schema documents into a suitable Java object structure. Remember that XML Schema's look like XML. So does the schema of XML Schema's

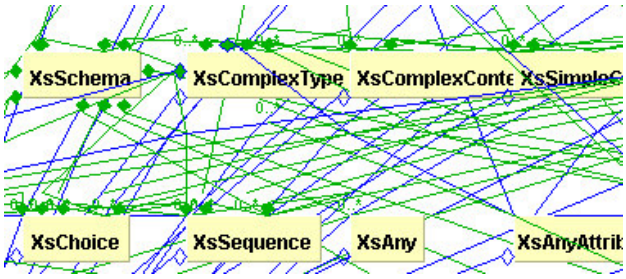

Fig. 5. Part of the XML Schema model

As you can see in fig. 5, the schema model is very ugly. It was not necessary however, to do any editing of the model, therefore the is no need to present a nice picture of it.

Using the generated classes describing the schema of schema's we could write some code that would transform a schema into a Fuut-je model, similar to the bindings of JAXB.

It was now easy and quick to create the XPDL editor using the new schema support of Fuut-je. The model obtained after reading the schema for XML looks rather ugly too, it contains some 65 classes. Here is a part of it:

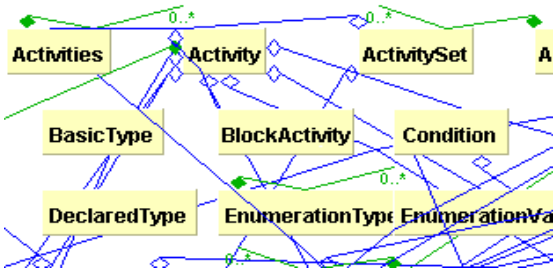

Fig. 6. Part of XPDL Model

We made no attempt to make a nicer layout, or to edit the XPDL model in any way. It is clear that the model could be simplified from a Java or XML perspective. However, we cannot touch the shape of the model. Otherwise it would loose compliance with XPDL as defined by the WfMC.

The binding of a schema to Java classes in Fuut-je is not nearly as comprehensive as what JAXB provides. It is adequate for our purpose though and in addition it provides us with a model from which we can generate other code besides XML marshalling classes.

The processing of Schema's by Fuut-je is not very comprehensive yet, but it can read the XPDL and similar fairly simple schema's

In the future we can improve on it, even make it JAXB compliant, and our work on generating GUI code etc. would not be lost.

4. WORKFLOW ENACTMENT SERVICE

The workflow enactment service, or workflow *engine*, has as responsibility to interpret the XPDL process definition and to orchestrate the starting and stopping of workflow activities accordingly.

Considerable discussion went into the question whether the engine should be the provider of events, where the activities are actively deciding how to react, or, whether the engine decides looking at the state of activities which activities can be started. A major consideration is, that applications should be totally unaware that they are being controlled by a workflow engine.

Fig. 7. Starting the Enactment Service

The currently implemented architecture provides for a control structure for processes and the activities contained within each process. This structure consists of a *shadow object* for each process or activity definition object. Each shadow object maintains the *state* of the definition object it controls. The corresponding Fuut-je model is very simple:

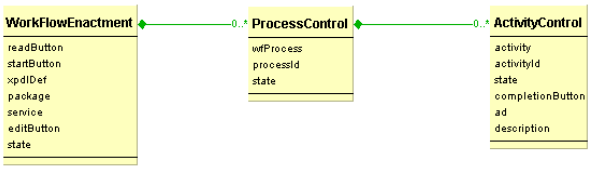

Fig. 8. Model for the Workflow Enactment Service

In this architecture, the workflow engine can be viewed as a state transition machine.

When the state of a shadow object changes, this causes a *propertyChangeEvent* to be fired. Fuut-je has built-in support for this. Processes will subscribe to the state change of activities, and the enactment service subscribes to the state change of its processes.

The *action* taken by a process or the service when it is notified of a state change, is to traverse the structure of its contained processes and activities to see whether any activity or process can be started, depending on the transitions defined.

For example, looking back at fig. 2, you will see that Fuut-je and Eclipse will simultaneously start, when GME is finished. The process and in this case the total workflow, will finish only when both Fuut-je and Eclipse are terminated.

The next step, activity activation, is described in the following section.

5. ACTIVITY ACTIVATION

At the time of writing of this document, activity activation is implemented in a simple way. For each activity a new thread is started and upon the completion of the thread, the activity is set to completed.

Once the enactment service has determined that an activity can be started, the application that implements the activity should be invoked. It is probable that for most applications that could participate in the GMT workflow a small wrapper should be developed, to present the application with input in the proper format. We cannot expect that all applications will be able to read their input and write the output in XMI form. Neither can we expect that these applications will understand XPDL. We assume that this can be implemented without problems.

Another aspect is that the enactment service should monitor the execution of the application, and be notified when the application terminates. The execution of an application that is the implementation of a workflow activity should be decoupled completely from the enactment service itself.

One solution would be to implement the workflow activities as web-services using SOAP. This would have the large advantage that the applications could run at any location as long as it can be reached via HTTP, where the enactment service acts as a client. This opens the exiting possibility of cooperating distributed applications within GMT.

The enactment service could poll an activated application at regular intervals to find out whether it is still running and in this way the enactment service could synchronize the workflow as required by the XPDL definition.

The disadvantage is undoubtedly that this implementation leads to further scope-creep. There is some very good open-source software available that may help us to make this task easier, for example Apache Axis [9], a popular, open source, SOAP toolbox.

With this development we produced a workflow enactment service that provides *simple workflow*. There is no support for conditional execution of activities, for sub-processes, or for a database of MDA component definitions. The workflow as defined now is of limited practical use. It helps to show the efficiency of our development method and the validity of the activity scheduler implemented.

6. OTHER CHOICES

It could be argued that the choices we have made for implementing the “MDA component glue” are not the best, and not obvious from an MDA perspective.

Why did we not use UML activity diagrams instead of workflow? Why did we not wait for a new transformation tool? Why did we not use the reusable asset specification [10], instead of XPDL?

Partly this may be ignorance of the author, partly this is due to our business background, where familiarity with certain tools leads to quicker results.

We could discuss what the differences are between UML activities and workflow activities. Or the essential similarities between XML Schema and XMI. This should not distract us from making progress with implementing GMT.

7. CONCLUSION

The experience of developing a workflow component for GMT shows that it is indeed a high priority for developing GMT itself to have this component in place. We missed it to orchestrate our own activities.

Surprisingly little manual code was needed to implement the workflow engine, and none at all was done for the XPDL editor. More code may be required to implement the activity activation. We hope that new generation templates for web-service support can ease the effort.

To do it right, and to adhere to standards, we need to accept a wider scope for the workflow component. As a consequence, this will make GMT interesting to a wider public, that is not necessarily interested in MDA, but instead in running distributed application development (or applications in general) in a workflow environment. We should avoid to develop a full-function workflow engine however.

With the growing use of web-services where XML schema's play an important role, a potentially very promising opportunity for model driven development can be found by providing tools within GMT for transforming XML Schema's into XMI and for interfacing with JAXB.

As a by-product of the workflow component effort, FUUT-je can now read XML schema's and interpret them as FUUT-je models.

8. ACKNOWLEDGEMENTS

Many thanks to the workshop-paper referees, who have sent me extensive and constructive reviews. In trying to incorporate their suggestions, the paper has become quite a bit longer. I hope that it also has improved.

9. REFERENCES AND NOTES

[1] http://www.eclipse.org/gmt

[2] Jorn Bettin, Software Requirements Specification 0.1 for GMT

[3] http://www.eclipse.org - "Eclipse is a kind of universal tool platform - an open extensible IDE for anything and nothing in particular".

[4] http://www.ofbiz.org/, Open for Business

[5] http://www.wfmc.org Workflow management Coalition.

[6] Workflow Process Definition Interface -- XML Process Definition Language", document number WFMC-TC-1025, version 1.0, 25 Oct. 2002.

[7] FUUT-je, *Fuutje* is Dutch for "small- great crested grebe". *Bron-stee* means "Source-Site", it is the name of a small lake behind my house. As an acronym, FUUT-je stands for: Fantastic, Unique UML Tool for the Java Environment.

[8] http://java.sun.com/xml/downloads/jaxb.html, **Java** Architecture for XML Binding (JAXB) Downloads & Specifications.

[9] http://ws.apache.org/axis/index.html, an implementation of the SOAP ("Simple Object Access Protocol").

[10] http://www.rational.com/media/products/Resuable_Asset_Specification_draft.pdf

[11] We acknowledge trademarks or registered trademarks of Sun Microsystems Inc., International Business Machines Corporation, Rational Software Corporation.

[12] http://www.research.ibm.com/journal/sj/392/vanemdeboas.html

[13] UMLX, Ed Willink, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/umlx/index.html