BIG MODELS
AN ALTERNATIVE APPROACH

Whitepaper Eclipse Summit 2008 Modeling Symposium

Jos Warmer, Ordina (jos.warmer@ordina.nl)

Abstract

Scaling up modeling within project runs into many practical problems. People try to find solutions using complex techniques based on e.g repositories, model merging, etc.

An alternative is to use small model units, and manage the relationships between these small model units. Interestingly many problems with big models simply cease to exist,

This paper describes experiences with the "small models" approach and explains the way this is implemented using the Eclipse Modeling Project tools. The implementation will become open source in April 2009.

1 Introduction

Modeling is becoming more popular as a way to speed up development using code generation. This automatically leads to the need for more and larger models in projects. In practice scaling up the use of modeling is not without problems.

More explicitly dealing with big models from a model management point of view leads to problems like:

- multi-user access
- version management
- reuse of models
- model libraries
- long waiting times for code generation..

When we use DSL's instead of e.g. UML, we also will run into additional problems like:

- evolution of DSL's
- combining different DSLs

When we decided to setup a new model driven software factory the decision was made to use multiple DSLs where each DSL focuses on a specific part of the architecture. These DSL are focused towards developers, we do not target business users at this time.

This paper describes our solution to the problems above and the experiences them. This solution has been implemented first in de Visual Studio environment using the Microsoft DSL Tools and is currently being implemented in Eclipse using various parts of the Eclipse Modeling project.

2 Model Units

When a DSL is defined we can create DSL Models. In many modeling languages, like e.g. UML, there is one large model which describes the whole system. This works nicely for small systems and small models. However, when we have multiple developers and larger models, working like this becomes more and more cumbersome.

When we compare this with programming, we can also say that there is one program describing the system. Imagine that you would have to edit, store and compile the source code of your complete program as a whole, i.e. one big file containing all the Java classes for your system. This is how the old fashioned *Main Program* worked. Such an approach does not scale up. Because of this programming languages, like e.g. Java, divided a program into different compilation units. A compilation unit usually is one file, containing only a small part of the complete program. In Java the class file containing one class is the compilation unit.

In modeling the usual approach is to have one model which is stored as a whole. This can be done either in a file, of in a repository. Sometimes we can store a model in multiple files, but to process the model (e.g. for code generation) we need to put all the pieces from the different files together first. You could say that we are working with one *Main Model*, much like we used to work with one main program in the past. This won't scale up.

To solve this problem we applied the idea of compilation units from programming languages to modeling, and found the need to define a DSL in such a way that we can split a DSL model into multiple ***Model Units***. Such a model-unit can be stored in its own resource, which in our case is just a file. This leads to many practical advantages:

- Different model units can be edited independent of each other
- Parallel development of modeling units by multiple developers
- Version and configuration management become easy because each model unit can be handled as a file and stored in well-known and proven repositories like CVS or SVN.

- Code generation can be done per model unit, having no need for a large and time consuming code generation step from a big model.

An earlier version of the idea of using *model units* has been described as *partial models* at the *6th OOPSLA Workshop on Domain-Specific*.

2.1 References between Model Units

The ability to work with separate model units is something that needs to be built into a DSL. A DSL needs to be defined in such a way that there are "natural" model units, much like the Java language has been designed on purpose to have classes as a separate compilation units.

Of course, when working with model units, we need to ensure that a collection of model units forms a consistent whole. Just as we do with all Java classes in a program. A model unit will, need to be able to refer to model elements defined in other model units and we need an environment that checks and validates all such references. To keep model units independent this must be done in a loosely manner.

One obvious solution, which is used at several places, is to allow one to refer from one model to model elements in another model. This allows the modeler to combine the models and ensure a consistent specification. There is one serious drawback in this solution. If there are direct links between different model units, conceptually we once again have one large model. If the model units are of different DSLs (which we will see into later), we also conceptually have one big modeling language again with one big meta-model. This defeats the purpose of having multiple models and multiple DSLs.

2.1.1 Name Based References

We have chosen a different solution. We define a DSL to contain *Reference To XXX* model elements. These elements are not direct references to elements in another model unit. They are a statement of the form "*This model element XXX is needed by this model unit*". The *Reference To XXX* model element contains attributes to be able to find the referred element. In our case this is a model name plus an element name.

To validate such a *Reference To XXX* each model unit exposes certain information about itself, which we call the model unit interface. In the most simple case this model interface exposes information that is identical to its model elements, but in some cases derived information is exposed.

The solution above is similar to the way that references in programming languages like Java are handled. For each Java class a symbol table is built which contains all the externally visible symbols. If a Java class refers to another Java class this reference is looked up the symbol table of the referred class. There never is a hard link.

2.1.2 Model Libraries

Because a reference only needs the interface information from another model unit, no model units other than the one you are working with need to be loaded. Even stronger, the referred model unit does not even have to be available, as long as the exported information of the model units is. This is similar to the fact that you don't need the source code of another Java class, as long as you have the binary .class file available. This opens up a way to have "binary' model libraries.

2.1.3 Location Independency

Several existing solutions to storing models in multiple files have another disadvantage. The reference is often done though a direct reference to another file, using e.g. a URI or a URL (e.g. the xText import URI). This approach has a disadvantage that the exact location of a referred model unit is stored in the referring model unit. This makes it hard to manage. Also, it makes it hard to refer to model elements in model unit libraries and even harder to plug-in / plug out such libraries.

In our solution we have implemented what we call broker, which takes care of mediating references between model units. Each model unit provides the broker with its exported information. This way the broker has a complete registry of all model units and their information. Whenever a model units needs to validate a *Reference To XXX* it will ask the broker whether such an element exists. The broker provided the answer without the model unit being aware what the location of the providing model unit is.

The broker works with a configured environment, which is a list of locations where he van find model units. This mechanism is much like the CLASSPATH mechanism in Java.

2.1.4 Incremental Code Generation

The use of model units is also central to the structure of our code generator. It is quite common in MDD environments to first build a complete model (whether in memory or in a repository) and generate code from this complete model. This clearly has advantages from the point of view of the code generator because you always have all information at hand that you might ever need. However this way of generating code defeats the purpose of having independent model units. The "complete model" that is built conceptually once again is one monolithic model Also, this kind of full code generation often becomes time consuming for larger models. We have seen times from 15 minute to many hours of waiting time.

Developers using modern IDEs like Eclipse are used to incremental compilation. Your Java code is compiled as you go and you never have these waiting times for "compile all".

For this reason we have decided to generate code from each individual model unit. In this way a model unit also becomes what we could a *generation unit* ,or in more generic modeling terminology a *transformation unit*. Generating code per model units allows us to generate code while you model. There is no need for a developer to first model and at some stage push the "generate all" button. As soon as you save a model, the underlying code is automatically generated and compiled.

To be able to generate code per model unit you need to design a DSL with this in mind. It is not something that comes automatically. You need to think much deeper about dependencies between model elements and have to design a DSL to support this. In our experience the effect is that the DSL itself becomes simpler, because you tend not to make every model element dependent on every other model element.

2.2 Model Units with Multiple DSLs

Because a DSL is a language with a limited scope, it is quite natural to use multiple DSLs. Each DSL allows us to describe a part of the system.

When using multiple DSLs we have the need to ensure that the combination of all model units from all DSLs for a consistent whole. To ensure this we use the same mechanism as within one DSL.

A DSL may de designed to have *Reference to XXX* model elements which refers to elements from another DSL. Validation between model units of different DSLs uses the same mechanism as within one DSL. The broker gets all information from model units of all DSLs and answers validation requests from all DSLs. The schema (or meta-model) of the exported information is generic and allows the broker to work independent of any specific DSL.

We will get all the advantages that we got with this approach for one DSL. Additionally we get some advantages that are specific to multiple DSLs.

2.2.1 DSL Evolution

Separate development and evolution of DSLs is simplified. DSL's can evolve separately or single DSL's can be replaced without effect on other DSLs. Because the internal structure of a DSL (namely its meta-model) is never exposed a DSL can be changed internally without having effects on other DSLs. As long as the interface information remains the same (or is extended) other DSLs will keep working.

In terms of Martin Fowlers description of language workbenches the above solution is a way to implement symbolic integration of DSLs.

2.2.2 Version Management for Large \Models

Large models are divided into many small model units. Each model unit is stored in a file and we use conventional and proven source code control systems like CVS, SVN or TFS to enable version control of models. Our experience is that developers use this without even thinking about it. It fits exactly in the way they work with code. Version Management of model units behaves like version control, for source code for all practical purposes. We have experience with somewhere between 50 and 100 model units in one project and have not encountered any problem yet.

Because this approach works for thousands of source code files we fully expect this to work for thousands of model unit files as well. We have not seen any characteristic of model units that behaves differently from source code from this point of view.

2.2.3 Multi User Access for Large \Models

Multi user access for large models works similar to version management. Developers can freely work on different model units without having to take care of other developers. When they check in their changes and other developers check out those changes, the CrossX validation process will find problems, if any. If there are invalidated references the developer has to solve them.

This is the same approach that is used for source code for many years and work pretty well in practice.

One problem that is not solved by our approach is merging of model units. In our experience, this is not a problem if the DSL is designed in such a way that developers can work with small model units. If a conflict of changes happens, then the developer needs to merge the changes. In the Microsoft DSL Tools environment we work with visual model stored as XML files, merging is therefore cumbersome. In the Eclipse environment we use Xtext for a textual syntax of models. In this case we van use standard merging tools that we use for source code as well.

Although merging is not solved, the need to merge changes in model units arises rarely. Therefore the problem becomes rather unimportant.

3 Implementation

The first environment where we implemented the approach described above is within Visual Studio Microsoft DSL Tools part of the Visual Studio SDK and allows one to define DSLs with a graphical syntax. The DSL Tools provide a stable environment for developing DSLs, but any kind of connection between different DSLs or between different models of the same DSL were missing. Currently some support has been added, but not with the flexibility that we need. To support our way of handling coordination, we developer our own broker . We call it the NDIP: Non-persistent Dsl Information Provider. This implementation has been used in fixed price project for almost two years.

We are currently working on new open source project canned ***Mod4j***: Modeling for Java (see www.mod4j.org). This project is not available yet, but it will be open sourced before the Eclipse Summit Europe 2008. We have chosen to first develop a version in house and test it with a pilot project before making it open source.

Mod4j uses many Eclipse modeling plugins, most notably: Ecore, EMF and Oaw (Xpand, Xtend, Checks, Xtext). The broker component in Mod4j that solves inter-model references is called Crossx. This section describes the implementation of Crossx and the integration into Eclipse. At some places we will refer to our experiences with the Microsoft DSL Tools implementation.

3.1 Model references

Using CrossX we support references between model units, either of the same or of different DSLs. If a reference is needed, the meta-model of the DSL needs to contain a meta-class *Reference To XXX*. The *Reference To XXX* class has two properties: n*ame*, and *modelname*. Modelers can instantiate this class in their model, fill in the values for the properties and they are done.

The validation rules for the DSL should check the *Reference to XXX* elements with Crossx and give an error when it does not. With Check, the code for this validation rules is as simple as:

```
context BusinessClassReference ERROR "BusinessClassReference [" + this.name +
                                     "] not defined in model " + this.modelname:
       classExists(this.modelname, this.name);
```

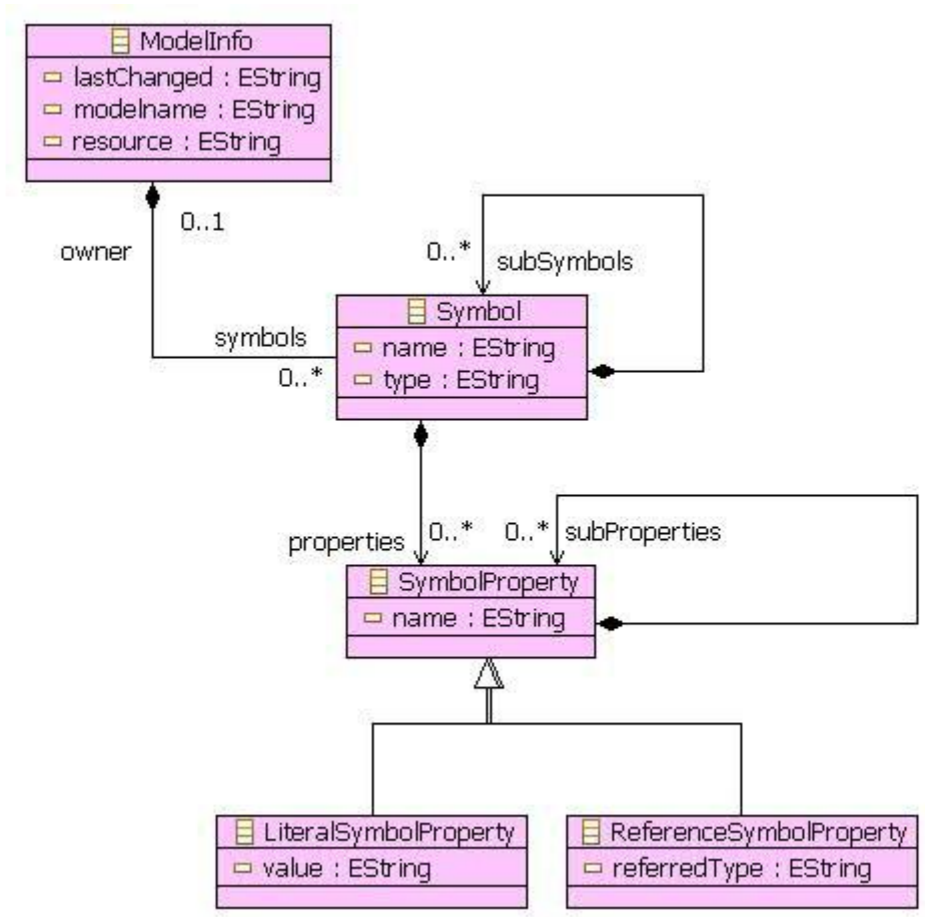
As soon as the model is saved, the model will check whether the referred element exists and give an error message if it doesn't.

The name of a model element is part of a model element, which means that each model element that you want to be able to refer to needs a *name* property. We have designed each of our DSLs such that each model and each referable model element has a *name* property. This allows the use of *name-modelname* pairs to identify a model elements. Models in different model units are allowed to have the same name.

The scope in which CrossX searches for a referred element is currently the project in which a model resides. This means that each Eclipse project can refer to model elements in any model within the same projects. In future versions we plan to enhance this with a real MODELPATH approach where CrossX can also use other Eclipse projects or external locations. In the Microsoft DSL environment the MODELPATH can already contain other projects/solutions. Note that the reference element and the model units in which this is contained has no knowledge where the referred element is located, this is determines by CrossX.

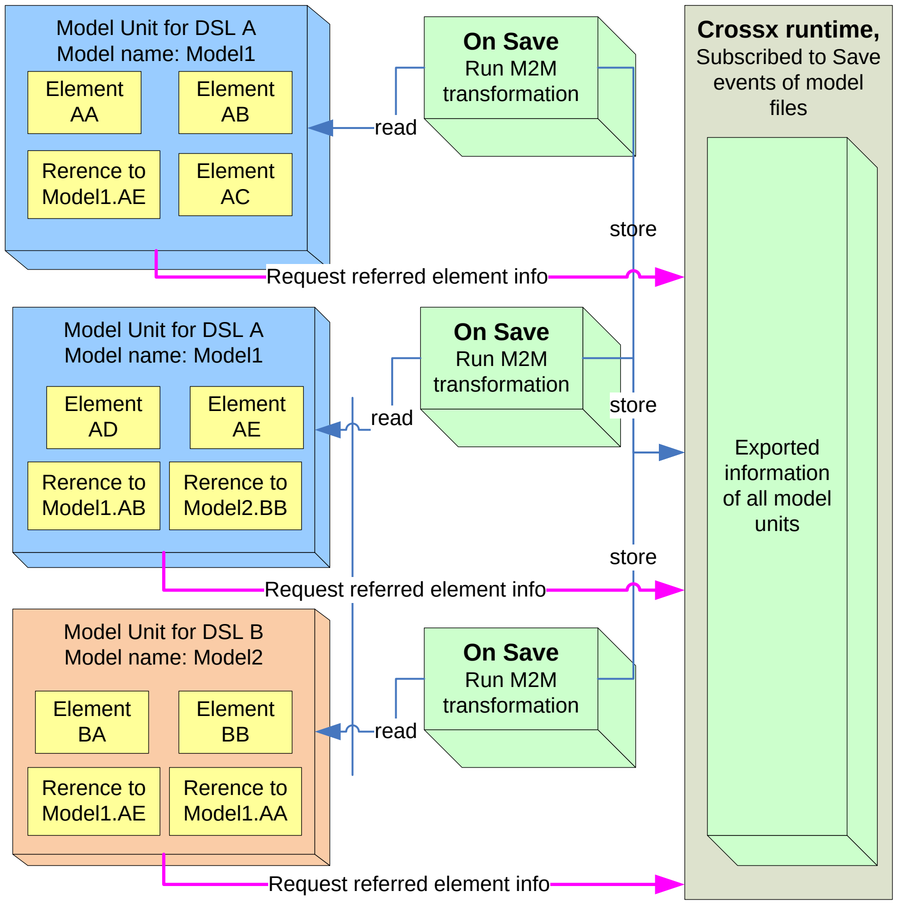
3.2 Exported Model Unit Information

The structure of the exported information for a DSL model has been defined using Ecore with the following meta—model:



As soon as a model unit is saved, CrossX calls a model-to-model transformation to transform the model unit of the DSL into an instance of the above CrossX model. This information is then stored by CrossX. If information about the same resource is already in CrossX, this information is updated with the new information. Whenever a model unit needs to validated its references it requests from CrossX whether referred elements exist.

The following figure shows how this process works.



In this figure we see three model units for two different DSLs. Note, for example, that the references from *Model2* to elements from *Model1* do not contain information about the model unit where the referred elements reside. Therefore a developer may move *Element AA* to the second model unit and *Model2* will still be completely valid. The CrossX component hides the location of the referred element.

To speed up processing when Eclipse is started, all CrossX information is stored in special .CrossX files. This allows the CrossX component to read the exported information from file without having to run a model transformation on all available models.

Mod4j comes with a Cross viewer to enable the developer to view the information the CrossX repository[1].

3.3 Code Generation

Code generation is done incrementally, meaning that as soon as a model unit is saved the underlying code is generated. The code generator is hooked into the Eclipse build system, which means that it runs as an Eclipse builder. The developer may turn the Eclipse automatic build on or off, thus enabling or disabling the code generator just like e.g. the Java compiler.

An Eclipse full build will run the code generator on all available mode units. In our experience (mainly with Microsoft DSL Tools, but also already with Eclipse) this gives the developer the feeling as if the model actually is

[1] Although we use the term repository here, the CrossX repository is strictly runtime within Eclipse. There is no repository as storage mechanism

the code. There is no superficial “generate code” step needed. All developers (sofar ☺) have reacted very positively to this feature.

3.4 Non model artefacts

The CrossX information to refer to model elements elsewhere defined is generic and can also be used for non modeling artifacts. To enable model units to refer to elements in the non-model artifacts we need to act an export mechanism to the non model artifact to produce the CrossX information. This information is available non only as an Ecore model, but also as an XML schema to make this process easier for non-Java artifacts.

To enable a non-model artifact to use information from model units the artifact can call the CrossX broker to request the exported information. CrossX has a simple Java interface and can easily be used from any Java component.

At the moment of writing we have not used this feature. In reality there will be non-model artifacts in any serious project, therefore we anticipate that this feature will become necessary in the future. For example in our DSL approach we acknowledge the fact that DSLs will be used in combination with handwritten Java code and handwritten XML configuration files. Both types of artifacts can be coupled with the DSL model units using CrossX. This enables us to validate all of these artifacts on the source level, avoiding often cryptic runtime errors.

3.5 Code Completion

The CrossX information can be used to provide code completion (called intellisense in Visual Studio) for model elements in CrossX. In the Microsoft DSL Tools implementation this is implemented, in the Mod4j implementation we plan to implement this in the upcoming release.

3.6 Adding a DSL to Crossx/Mod4j

The CrossX plugin defines a DSL extension point. DSLs that want to participate in the CrossX way of combining DSLs and model uinits should extend this point. The extension point defines the following attributes:

- *dslName* - The readable name of the DSL
- *dslMetamodelPackage* - The metamodel package of the DSL, as generated by EMF
- *dslFileExtension* - The filename extention of the DSL
- *dsl2crossxWorkflow* - The oAW workflow that will convert a DSL model unit into a CrossX model.
- *dslCodegenWorkflow* - The location of the worflow file for code generation

Using these attributes CrossX is able to call the workflows to get the model information and generate code at the correct time within the Eclipse build process.

3.7 Running Outside Eclipse

Outside Eclipse CrossX works in collaboration with Maven to allow a continuous build on a build sever without Eclipse. Although this part doesn’t look as flashy as the Eclipse implementation we feel that being able to automatically run automatic build is a necessary requirement for any serious project. We therefore have implemented the CrossX broker and the incremental code generator in such a way that they run as a Maven plugin as well.

4 Mod4j as open Source project

The Mod4j open source project will go online in November 2008. Mod4j will include a collection of three DSLs to start with. However, this is not intended to be a complete collection for everyone.

The goal of Mod4j is to provide an open environment for developing and using multiple DSLs. We plan to add more DSLs in later versions and invite everyone to add his/her own DSL.

The use of CrossX to enable communication between DSLs and model units in a flexible DSL-independent way is crucial to enable this type of collaboration.

5 Conclusion

Independent Model Units solve many of the problems that have been encountered with large models. A summary of problems that can be solved is given in the following list.

1. Each DSL is independent of the internal structure of another DSL. This allows DSLs to evolve over time.
2. Cross model Information is available without the need to know from which DSL it actually comes.
3. Models need not be active or in memory to get the information to do cross referencing.
4. A reference in the above structure does not even know from which model unit the information is coming, it is fully location transparent.
5. Exported information is structured for usability by other models, not by its internal domain model.
6. Multi user access to models becomes easy
7. Version management of models becomes easy using existing version control systems
8. Code completion over multiple model units and multiple DSLs is supported
9. Other types of artifacts, which are not model units, can also use the same CrossX to exchange information with model units. This approach allows us to integrate not just DSL:'s, but any kind of artifact.
10. If both the model file and the exported information file are checked into source control, developers can use the exported information without the need to have the source of the corresponding model unit being installed.
11. Model unit reusability
12. Possibility of "binary" model unit libraries

Points 1 until 8 have been used by us and our experience has convinced us that we did solve these problems to a large extent. This has given us such an improvement over the "main model" approach that we are confident that this can be scaled up ti even larger models.

The last four points (9 – 12) have not been exploited by us at the time of writing, we plan to implement these in the future. Evidence on the first eight advantages suggests that this might work very well, but the proof of the pudding is in the eating. We will report back when we have tried them ourselves.