DSM model-to-text generation: from MetaDepth to Android with EGL

Rafael Ugaz

Antwerp University (Belgium), rafaelugaz@gmail.com

Abstract

This article describes the process of a specific model-to-text transformation using the Epsilon Generation Language. Where the source models are created in a text-based Domain-Specific Modeling Language implemented in the MetaDepth meta modeling framework. The domain in question is of Role-Playing Games and the target text to be generated is Java code for an existing Android application framework.

Keywords: DSML, EGL, MetaDepth, model-to-text, Android

1. Introduction

In Model-Driven Engineering, models are the primary artifacts in the development process, as opposed to code in Code-Driven Engineering. Therefore powerful and efficient *model management* tools are needed. One important task of model management is model-to-text (M2T) transformations, which allow the generation of code or documentation from models. The Epsilon Generation Language (Rose et al., 2008), a model-to-text transformation language that is a component in the Epsilon model management tool chain (Kolovos et al., 2010).

In this paper, we will present and discuss a solution that uses EGL to transform a model created in a Domain-Specific Modeling Language (DSML) or formalism into Java code for an Android application. The domain of the formalism is the one of Role-Playing Games (RPG), which means that it's purpose is creation of RPGs. The RPG formalism has been implemented in the MetaDepth meta modeling framework, where also the RPG models that conform to this formalism can be created and, thanks to it's integration with EGL, the generation of text from the models can be performed. The

generated text will be used as Java code for populating a model of an RPG Android application. This application has been implemented with the specific purpose of displaying, simulating and allowing interaction with the user of any (well formed) model created with the RPG formalism.

The paper starts with a small explanation of Domain-Specific Modeling in Subsection 2.1. In Section 5 Subsection 2.1

2. Background

2.1. Domain-Specific Modeling

Domain-Specific Modeling (DSM) is a software engineering methodology that aims to raise the level of abstraction and at the same time narrow it's design space to a specific problem domain. This is achieved by the creation of three things:

- A domain-specific modeling language
- A domain-specific code generator
- A domain framework

With these 3 elements, a developer needs only to create models in the DSM language and the applications are automatically generated as code running on top of the domain framework (Kelly and Tolvanen, 2008). With DSM, a problem of the specific domain can be solved using familiar domain concepts and without writing a single line of code.

In the concrete case of the presented solution the modeling language is the RPG formalism, the code generator an EGL template and the domain framework the RPG Android application.

2.2. MetaDepth

MetaDepth (de Lara and Guerra, 2010) is a framework for multi-level meta modeling that supports deep meta modeling which means it can work with more than two meta modeling levels at the same time. In this project however, we have only used two meta modeling levels (see Subsection 3.1). MetaDepth also features integration with the Epsilon family of languages (Kolovos et al., 2010) which allows the use of EGL for transforming a MetaDepth model into code with just a couple of commands. It also hosts EOL (another

Epsilon language) as action and constraint language, which make the checking of the consistency of a model easier.

One of the drawbacks of MetaDepth however, is that it has a command-line textual interface. This is no problem for programmers that are used to interacting with a textual interface but it is certainly a big obstacle for people with no experience in programming. The creation of models is also text-based and even though the syntax is not complicated and that it can be written in an external text-editor, it still can't compete with visual modeling tools when used by non-programmers.

Regarding the use of MetaDepth to generate EGL, it requires just a couple of commands and it can even be done automatically in a script where multiple commands are passed. One problem however is that the error trace given when there is a problem with the EGL code: it prints many Java exceptions that give no information about the possible source of the error. A command to perform a syntax check of egl programs would be useful and could save a lot of time, specially in case of very large EGL files. There exists a syntax check of EGL that metaDepth shows but it fails to give a message in some cases like missing brackets of EOL for or if constructs, it actually does not throw any error but it simply does not generate any code; these type of syntax errors are very hard to find when looking at the code.

2.3. The Epsilon Generation Language (EGL)

EGL is a template-based language for model-to-text transformations. It has been built atop the Epsilon suite of tools and domain-specific languages for model-driven development. Epsilon is meta model-agnostic in the sense that it supports models written in any language and therefore also the generation of code from these models with EGL. It provides built in support for models designed in EMF, XML, CSV or Bibtex but it also allows interaction with other modeling technologies that conform to the interface of the Epsilon Model Connectivity (EMC) layer. Which is what allows MetaDepth (see Subsection 2.2) to use some of the Epsilon languages e.g. EGL.

Template-based means that EGL programs resemble the text that they generate. An EGL template is constructed of *sections*. Static sections contents appear verbatim (copied word for word) in the generated text. Dynamic sections on the other hand, contain executable code that give control over the generated text. The executable code of the dynamic sections is written in EOL, which support the navigation and modification of models. A snippet of the code used for generating the a part of the solution can be seen in

Listing 1 and the generated Java code in Listing 2. To delimit a dynamic section, the tag pair [% %] is used. Any text that is not inside such a tag pair belongs to a static section.

```
1 [% for (v in Villain.allInstances()) { %]
2 Villain [%=v%] = new Villain([%=v.onTile%], "[%=v.name%]", [%=v.
      attack%], [%=v.hp%]);
3 [%=v.onTile%].setCurrentCharacter([%=v%]);
4 [% } %]
```

Listing 1: A snippet of the EGL template used in the solution

```
1 Villain bernard = new Villain(s25, "Bernard", 5, 100);
2 s25.setCurrentCharacter(bernard);
3 Villain kevin = new Villain(s55, "Kevin", 5, 100);
4 s55.setCurrentCharacter(kevin);
5 Villain donald = new Villain(f34, "Donald", 15, 100);
6 f34.setCurrentCharacter(donald);
7 Villain john = new Villain(c12, "John", 20, 100);
8 c12.setCurrentCharacter(john);
```

Listing 2: The text generated from the template in Listing 1

In order to use M2T trasformations in the large, it is not enough to only generate text but also multiple files from a single model. EGL provides a co-ordination engine that supports generating text directly to files. The *Template* data-type allows nested execution of M2T transformations. One or many Templates can be invoked from within another Template, this allows EGL programs to share and re-use Templates. Typically, an initialization Template is executed first that sets the destination of the text to be generated and then invokes other Templates to perform the actual generation of text in the specified directory. This approach has also been used in this project.

To simplify the creation of Template objects, a factory object, Template-Factory, is built in and can be used to load templates from a file in a specified system path and store them in variables of the current Template.

3. Solution

3.1. Domain-specific modeling language: RPG

The domain specific language of the solution is the RPG formalism, that allows the creation of Role-Playing Games (RPGs). This formalism has been

implemented in the MetaDepth meta modeling language (See Subsection 2.2) and it has been enriched with constraints that help to check if a model is well formed.

The structure of an RPG consists in a World that contains one or many Scenes, for example a forest, a swamp or a castle. Each Scene consists of Tiles connected to each other where at most one character of the game can stand on at the same time. There are two types of characters in the game: the Hero, which is the main character and is controlled by the player, and the Villains, which are the enemies of the Hero and are controlled by the computer. There can only be one hero in the game but there may be multiple villains, each with different characteristics. The objective of the game is to collect all the Goal items with the Hero without dying. There are also other type of items that the Hero can pick up: Weapons, that make the attack of the Hero stronger and Keys, which allow the Hero to open Doors. Doors are a type of Tile that allow the player to go from one Scene to another. A Tile can also be a Trap, which damages the Hero when stood upon, or an Obstacle, which block the way and where no character can stand on. A class diagram of the RPG formalism can be seen in Figure 1.

The RPG meta model is also enriched with constraints which check if models are well formed. Typical checks made by the constraints are for example:

- There can only be one hero in the game.
- No character can stand on an obstacle tile.
- There exists at least one key that opens each door.

The creation of RPGs using the RPG formalism is done in a textual way either in a text editor of choice or directly in MetaDepth. After it has been created, a model can be loaded into MetaDepth and selected as the current context. From here a user can either validate the model (recommended) or load the EGL template (discussed in Subsection 3.2) to generate code from the model. This process can be automated with a script that is also part of the solution.

3.2. Domain-specific code generation: EGL

For the code generation, two EGL programs or templates were used. The first one simply sets up the path and name of the generated file and then

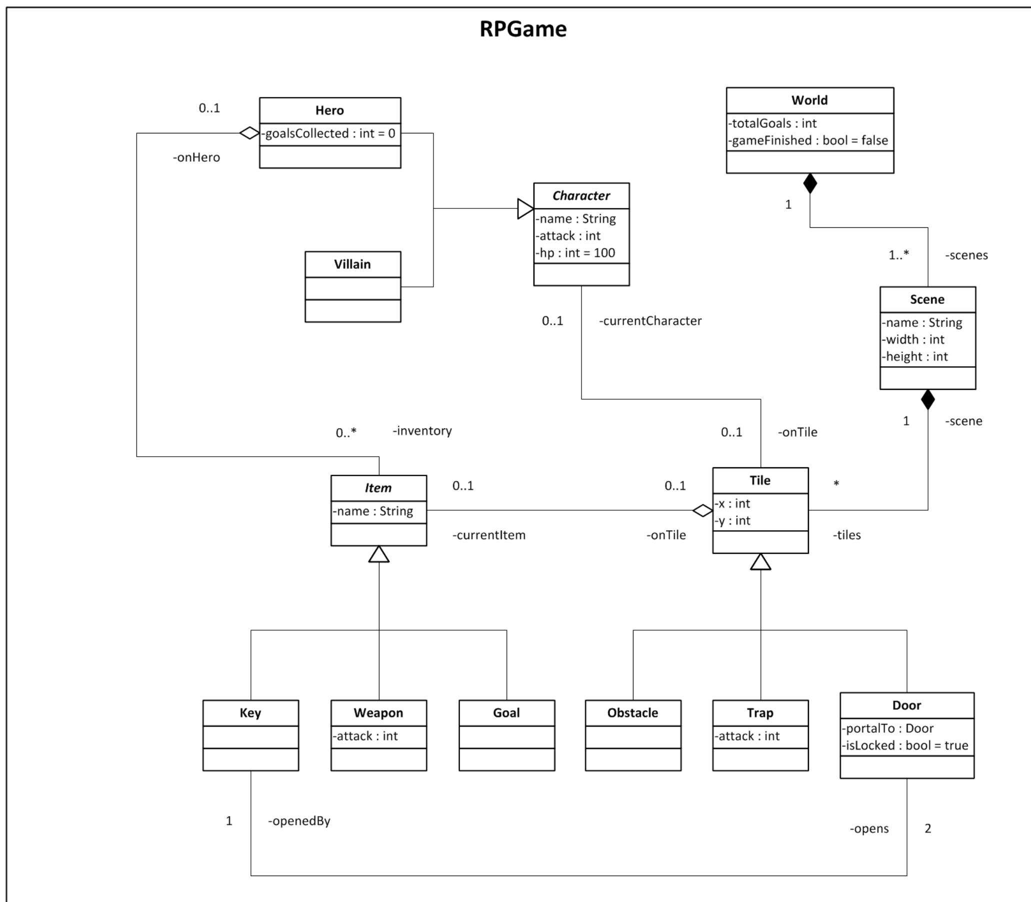

Figure 1: Class diagram of the RPG meta model

invokes the template that performs the actual code generation. The name of the file created is the name of the RPG model variable concatenated with the '.Java' extension.

The code generation template first imports all the necessary classes of the framework (this is static and depends on the framework). Then it creates one class that extends the RPGame class (an abstract class that all generated models subclass). Inside this only class, also named after the RPG model variable, the model objects are instantiated in Java syntax. First all scenes and their respective tiles, then the characters, the items and finally doors and keys are linked with each other.

3.3. Domain framework: Android

The Android application used for this project can be divided into four parts: the graphical user interface, the operational semantics, the meta model and the models. Only the last part, the models, are generated with EGL in the way explained in Subsection 3.2.

Figure 2: The RPG Android application

The rest of the application which is the domain framework has been implemented manually and for the specific purpose of displaying, allowing interaction with the user, and simulating an RPG model. The details of the Android application's implementation have been left out because it escapes

the scope of this paper. The finished RPG Android application can be seen in Figure 2.

4. Conclusions and further work

The presented solution in this paper is, although simple, a complete DSM implementation in the sense that it contains the three fundamental things that form a DSM solution: a domain-specific language, a domain-specific code generator and a domain-specific framework.

To evaluate our solution one can compare it with the manual implementation of RPG models directly in Java. Most of the benefits of our solution are similar to those of any DSM solution:

- First, the need to use the Java programming language which is a general purpose language instead of the RPG specific language means that the model designers must have knowledge of Java, even if it is for the creation of one simple class like the one generated in this solution. Even though designing the model through MetaDepth resembles using Java because of being text-based and not visual, the fact that the domain is much more limited and that the abstraction is slightly higher (no need to worry about imports, class or package declarations) makes it easier to use.

- The generated code needs no further editing and is free of most kinds of syntax or logic errors. Only inconsistency errors could occur at runtime but this is discussed in next bullet.

- The *validate* command of MetaDepth can be used after the creation and loading of the model in order to check if the model violates any of the existing constraints of the meta model. This prevents more complex errors from happening at runtime before any code has been generated.

For this small example one could argue that one of the weaknesses of this approach is the amount of extra work invested in order to automate the creation of one single file. But this was of course just a simple illustration example compared to real applications of DSM where this is not the case. A real weakness of this approach in our opinion is the fact that the modeling can not be done visually. This limits the use of the application to people with certain expertise in programming, who can create a model in a text-editor in the MetaDepth modeling language.

For further work, a natural next step would be to create a domain-specific *visual* language that would make the creation of models much more accessible. One possibility could be the use of the AToM3 modeling tool which supports the creation of visual concrete syntax for domain-specific languages.

5. References

de Lara, J., Guerra, E., 2010. Deep meta-modelling with metadepth. TOOLS Europe 2010: 48th International Conference on Objects, Models, Components, Patterns 6141, 1–20.

Kelly, S., Tolvanen, J.-P., 2008. Domain-Specific Modeling. John Wiley & Sons, Inc., Hoboken, New Jersey.

Kolovos, D., Rose, L., Garcia-Dominguez, A., Paige, R., 2010. The epsilon book. Structure.

Rose, L., Paige, R., Kolovos, D., Polack, F., 2008. The epsilon generation language.