Mapping Generated Code to a Reference Implementation – Towards Automatic Code Migration [*]

Daniel Lucrédio[1] **and Renata Pontin de Mattos Fortes**[2]

[1]Departamento de Computação – Universidade Federal de São Carlos
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brazil

[2]Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brazil

`www.dc.ufscar.br/~daniel`

Abstract. *This paper describes an initial step towards automatic code migration in the context of MDE-based projects that use template-based code generation backed up by a reference implementation. Different types of mapping between a template and the generated code are identified, each one covering a different code generation possibility. The mapping is established by the automatic insertion, in the generated code, of special markings that delineate the boundaries of each mapping type. An implementation using a popular template-based code generation engine is also described, demonstrating the viability of the approach. Future work may use the information in this mapping to (semi-)automatically migrate code from the reference implementation back to the template.*

1. Introduction

Model-Driven Engineering strongly relies on code generation to automate implementation tasks [Schmidt 2006]. The current technology of choice for implementing generators are the code templates. A code template is a text file instrumented with selection and code expansion constructions [Czarnecki and Eisenecker 2000]. These constructions are responsible for reading the input, which may be a program, a textual specification, a set of windows and dialogs, or even visual diagrams [Cleaveland 1988]. The information read is then used as a parameter to produce customized code, in any textual language. Code templates have many benefits, like the ease of adoption by the developers [Czarnecki and Eisenecker 2000] and the ability to produce code that is similar to handwritten - hence more readable - code [Cleaveland 2001].

To build such templates, Cleaveland [Cleaveland 1988] highlights the importance of having an example application to use as a basis. It is easier to build a generator by first analyzing existent code in search for recurring patterns and identifying how they may vary, and then inserting this code, together with code generation constructions, into a template. Many approaches in the literature follow this idea [Bierhoff et al. 2006, Visser 2007, Wimmer et al. 2007]. The example application is also known as a reference implementation, and the process of inserting its code into a template is known as code migration [Muszynski 2005]. The benefits of such approach include: (i)

[*]This work was developed with partial support from Brazilian institutions FAPESP (process 2008/11476-8) and the National Institute of Science and Technology for Software Engineering (INES - www.ines.org.br), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

Larger code amounts may be reused, since any pre-existing code can be migrated; (ii) Incremental code migration facilitates the construction of non-trivial generators; (iii) The produced generators have higher quality, since the reference implementation can be previously tested, debugged and validated in the traditional way; and (iv) The developer may use his/her own environment to develop the reference implementation, not having to use a particular tool only because the code generation engine does not work in a different one.

However, there are two problems caused by code migration [Muszynski 2005]: (i) It consumes 20-25% of the reference implementation development time; and (ii) It causes code duplication. Although after the first migration it is possible to completely discard the reference implementation, in practice this does not happen, because, when working directly with the templates, it is much more difficulty to understand, modify, debug and test some code. As a result, the developer often keeps the reference implementation as a working example, migrating the code whenever necessary.

Although not trivial, automation would be a solution to these problems. But existing code generation engines like Java Emitter Templates (JET), Velocity and Microsoft Text Templating[1] do not support automatic code migration. The literature also does not seem to cover this issue, as no directly related work was found. One research area that comes closer to this specific problem is known as round-trip engineering [Hettel et al. 2008]. This research field studies ways to maintain the synchronism between the source and target artifacts involved in a software transformation. For example, it is possible to synchronize an OO model with an E-R model, so that changes can be made on either side, being reflected on the other side [Paesschen et al. 2005]. But no study specific to the code migration problem was found.

Under such motivation, this paper describes a first step towards the automation of the code migration process. Code annotations are used to establish a mapping between the reference implementation and the generated code, so that when changes are made into the former they may be more easily propagated to the latter. Initially, this process is intended to be used in maintenance, reducing the negative effects of the code duplication problem. Later, it can be used in initial development, with partially automated incremental code migration, reducing the extra effort spent when code is migrated for the first time.

Given that there is not much related work, novel contributions are expected. Even partial automation could save precious development time, maybe reducing the 20-25% extra effort by half. In large projects, this may translate into considerable savings.

The paper is organized as follows: Section 2 presents a formalization of the MDE cycle, to serve as a basis for the studies in this research. Section 3 presents details about how the mapping is established, and how it was implemented in a popular code generation engine. Section 4 presents some concluding remarks and future work.

2. The MDE cycle

For the purposes of this research, ten activities were identified in the MDE cycle, as shows Figure 1.

A1. Construction of the reference implementation: This is a normal application development activity, providing an example of the code to be generated.

[1]Available at help.eclipse.org, velocity.apache.org and msdn.microsoft.com

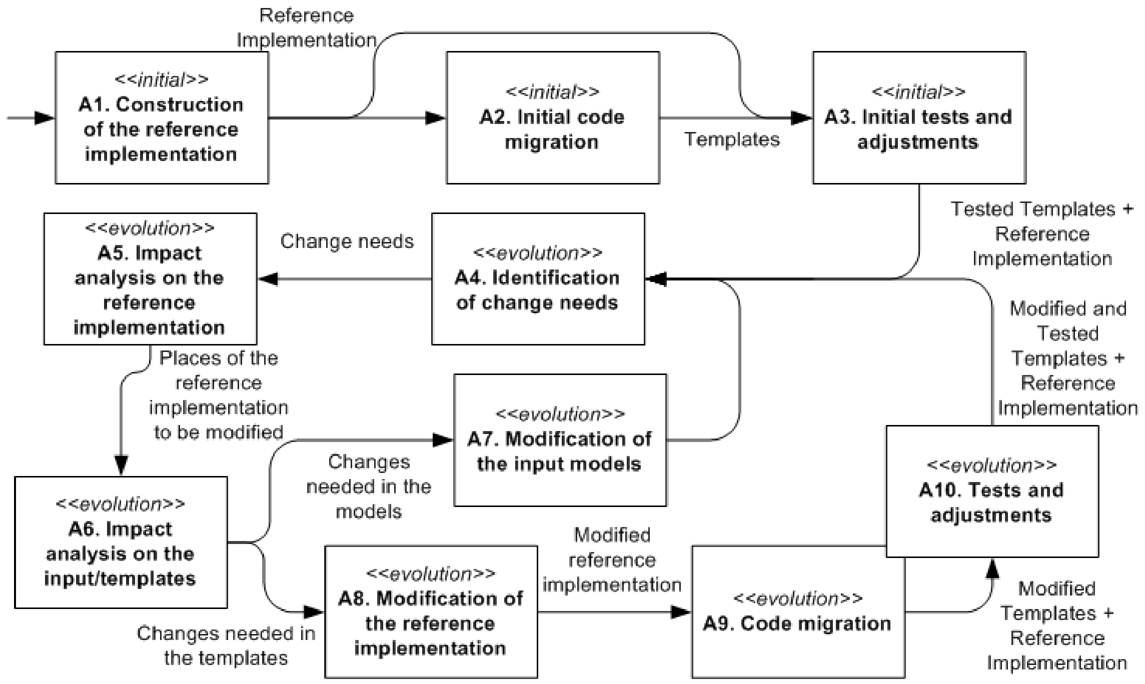

Reference Implementation
<<initial>> A1. Construction of the reference implementation
<<initial>> A2. Initial code migration
Templates
<<initial>> A3. Initial tests and adjustments
Tested Templates + Reference Implementation
<<evolution>> A5. Impact analysis on the reference implementation
Change needs
<<evolution>> A4. Identification of change needs
Modified and Tested Templates + Reference Implementation
Places of the reference implementation to be modified
<<evolution>> A7. Modification of the input models
Changes needed in the models
<<evolution>> A6. Impact analysis on the input/templates
<<evolution>> A10. Tests and adjustments
Modified reference implementation
<<evolution>> A8. Modification of the reference implementation
<<evolution>> A9. Code migration
Changes needed in the templates
Modified Templates + Reference Implementation

Figure 1. MDE process using a reference implementation. <<**initial**>> **indicates activities related to initial development and** <<**evolution**>> **indicates activities related to software evolution and maintenance.**

A2. Initial code migration: Consists in annotating the example code with scripts and tags, transforming it into code generation templates;

A3. Initial tests and adjustments: This activity aims at evaluating the results of the initial migrations, performing adjustments if necessary;

A4. Identification of change needs: Activity that starts a new evolution/maintenance cycle, by identifying change needs;

A5. Impact analysis on the reference implementation: Consists in identifying, in the reference implementation, which parts of the code need to be modified;

A6. Impact analysis on the input/templates: Some changes require only the modification of the input models followed by code re-generation. Others require the modification of the templates, or both. In this activity, these needs are identified;

A7. Modification of the input models: Consists in modifying the models and re-generating the code, which is the canonical use of MDE;

A8. Modification of the reference implementation: This activity is performed when changes are needed in the templates. These are made first in the reference implementation, where they can be tested and validated in a traditional way;

A9. Code migration: In this activity, changes made to the reference implementation are propagated to the templates. The goal is the same as activity A2's, but in this moment the templates already exist, needing only to be updated; and

A10. Tests and adjustments: Aims at evaluating the results of code migration and performing adjustments if necessary.

These activities do not cover all possibilities. There may be cases where some are not necessary, and cases requiring activities not included here. However, the objective here is not to develop a complete process for this kind of development, but rather to build a base for further studies involving the code migration problem.

3. Mapping Generated Code to the Reference Implementation

During the process described in the previous section, the reference implementation may enter and exit a state of synchronism with the templates, as shows Figure 2.

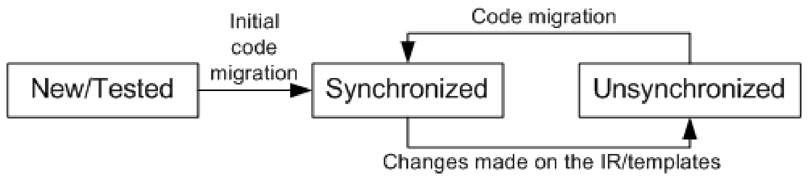


Figure 2. States of the reference implementation during the development cycle

New/Tested: The reference implementation has just been built. It is still not annotated, i.e. there are not any templates associated to it;

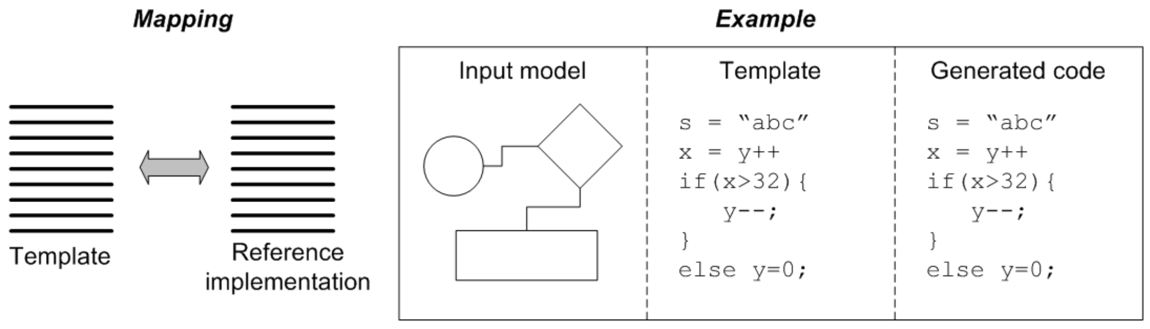
Synchronized: In this state, the reference implementation is synchronized with the code generation templates. This always occurs as a result of code migration; and

Unsynchronized: There are one or more pieces of code where the synchronization is lost due to some modification in the reference implementation and/or in the templates.

It becomes evident that the maintenance of the synchronism is a responsibility of code migration – activities A2 and A9. The strategy used in this work is the following: code annotations are inserted both in the templates and in the reference implementation, to mark the pieces of code that correspond to each other, synchronizing them. To make this possible, after code migration, new code is generated to replace the old reference implementation. This allows code annotations to be automatically inserted into the reference implementation to establish the mapping.

The synchronism must guarantee that the mapping between the reference implementation and the templates is correct, precise and coherent. All possibilities of generating code must be covered, and a different schema must be developed for each one. In this work, seven different code generation possibilities were identified, each one requiring a different mapping type, which were named 1 to 7. Each type is described next.

Type 1 - simple copy: This type of mapping is the simplest. A piece of code in a template is copied into the reference implementation. A one-to-one relation is established, and the input model is not consulted. Figure 3 shows this mapping type:

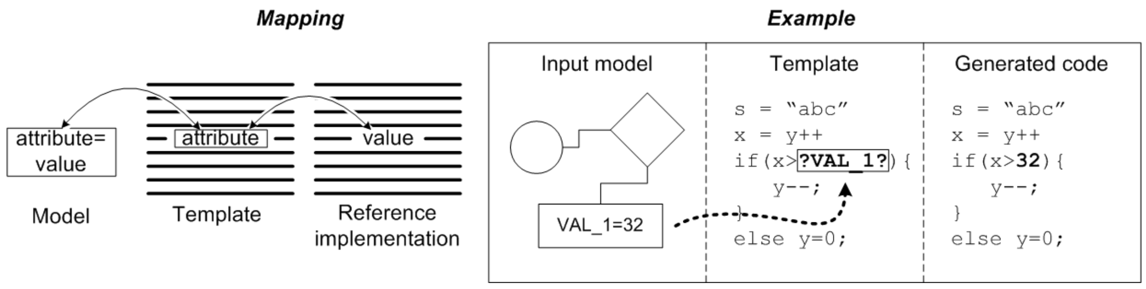


```
s = "abc"
x = y++
if(x>32){
   y--;
}
else y=0;
```

```
s = "abc"
x = y++
if(x>32){
   y--;
}
else y=0;
```

Figure 3. Type 1 mapping

Type 2 - simple substitution: A query to the input model is inserted into the template, being replaced with some value obtained from the input model when code is generated. It is possible to trace the replaced value to the input model, as it is a simple substitution in a one-to-one relationship. Figure 4 shows this mapping type:

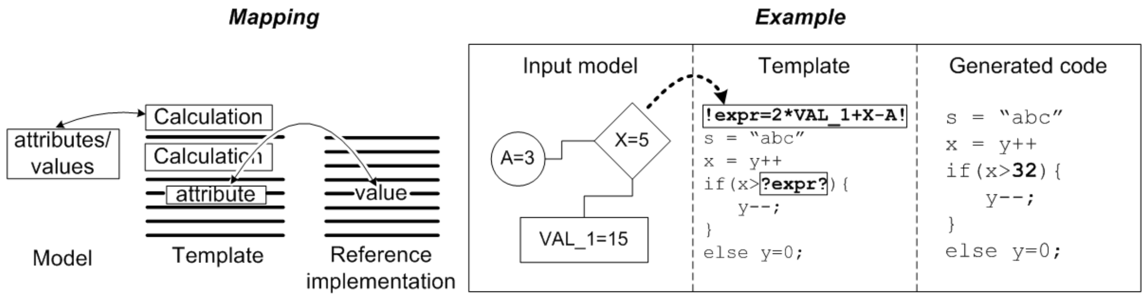


Template:
```
s = "abc"
x = y++
if(x>?VAL_1?){
    y--;
}
else y=0;
```

Generated code:
```
s = "abc"
x = y++
if(x>32){
    y--;
}
else y=0;
```

Figure 4. Type 2 mapping

Type 3 - indirect substitution: Similar to type 2, but here it is not possible to trace the source of the replaced value, as it is not explicit in the model, being calculated in the template. The relationship is one-to-one. Figure 5 shows this mapping type:

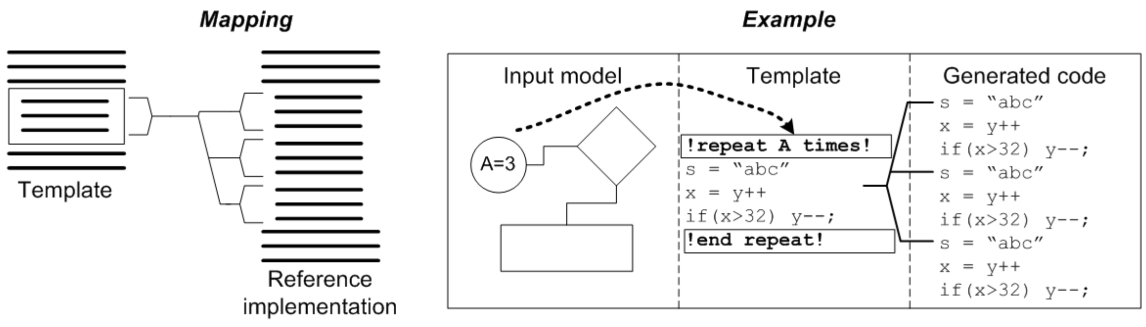


Template:
```
!expr=2*VAL_1+X-A!
s = "abc"
x = y++
if(x>?expr?){
    y--;
}
else y=0;
```

Generated code:
```
s = "abc"
x = y++
if(x>32){
    y--;
}
else y=0;
```

Figure 5. Type 3 mapping

Type 4 - repetition: A piece of code from the template is repeated zero or more times in the reference implementation, according to some criteria, which may be obtained from the input model or not. The relationship is one-to-many. Figure 6 shows this mapping type:



Template:
```
!repeat A times!
s = "abc"
x = y++
if(x>32) y--;
!end repeat!
```

Generated code:
```
s = "abc"
x = y++
if(x>32) y--;
s = "abc"
x = y++
if(x>32) y--;
s = "abc"
x = y++
if(x>32) y--;
```

Figure 6. Type 4 mapping

Type 5 - conditional: A piece of the reference implementation can be mapped to different pieces of a template, based on some condition established in the template. The condition may consult the input model or not. The relationship is many-to-one. Figure 7 shows this mapping type:

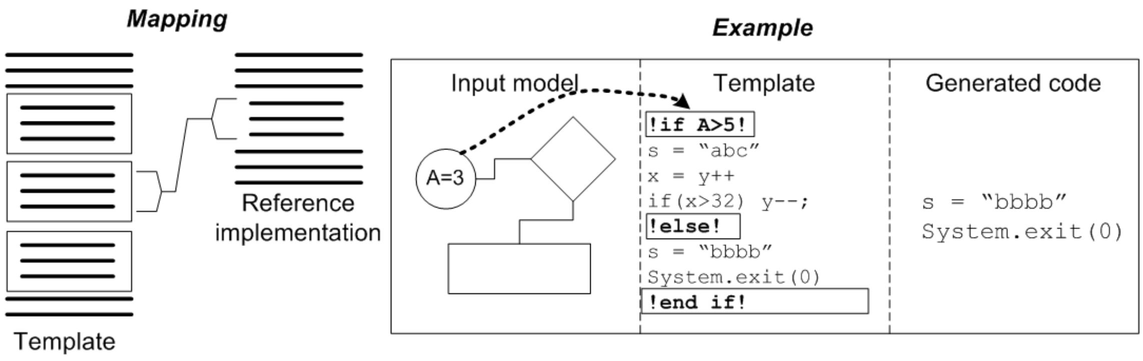


Figure 7. Type 5 mapping

Type 6 - inclusion: A template includes another template to generate a particular piece of code. The relationship between the templates and the reference implementation is many-to-one. It is important to highlight that it is always possible to identify, for each line of generated code, the exact template responsible for its generation.

Type 7 - new file: A template requests the creation of a new file. After this request, the code generated by the template is redirected to this newly created file. The relationship between the templates and the reference implementation is one-to-many. As in type 6, it is always possible to identify, for each different file that contains generated code, the exact template responsible for its generation. Figure 8 shows mapping types 6 and 7.

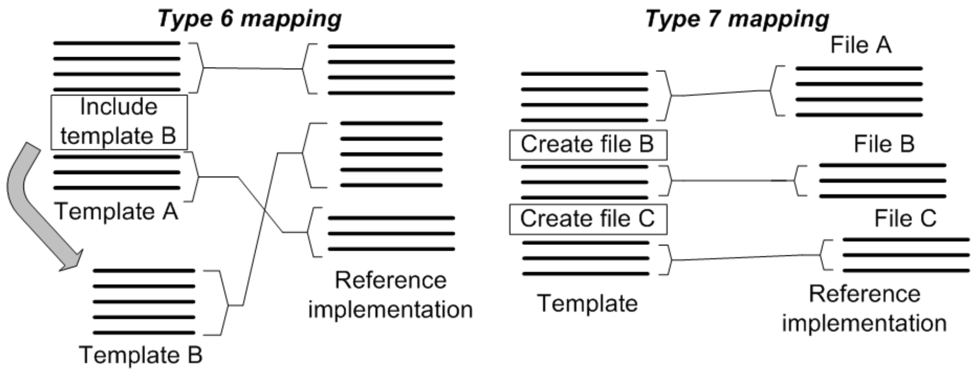


Figure 8. Type 6 and type 7 mapping

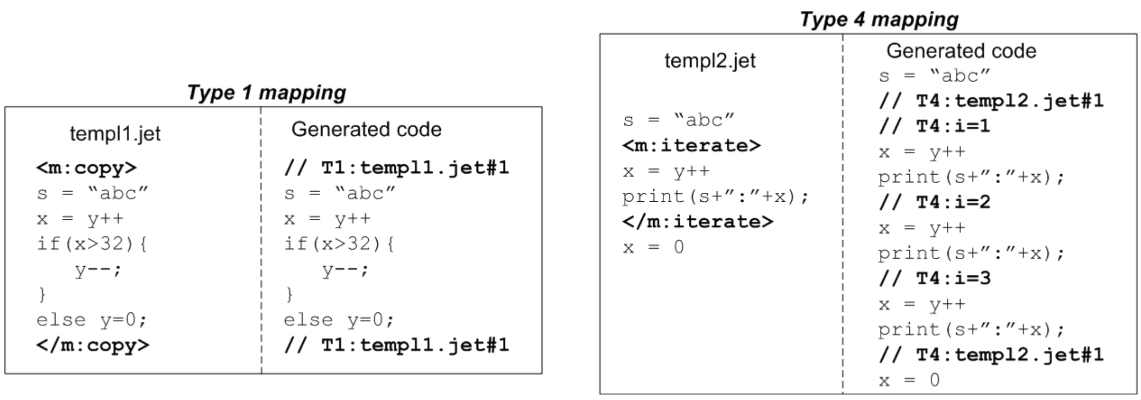
This mapping was developed so that any situation can be described by combining its seven types. So far no situation was faced that required a mapping that is different than these. However, in future developments it might be necessary to revisit this initial list.

As mentioned before, the mapping is physically established using code annotations. After code migration, new code is generated to replace the reference implementation. This new code will contain special markings that delineate the boundaries of each mapping type. To implement this functionality, JET's support for custom tags was used. JET is a popular template-based code generation engine, containing special tags for each code generation option. For example, tag `<c:if>` implements conditional code generation, and tag `<c:iterate>` implements repetition. Besides these predefined tags, it is possible to create new tags, defining its behavior programmatically (using Java).

In this research, one custom tag was created for each of the seven mapping types described earlier. The prefix `"m:"` distinguishes the new tags from the original ones:

- `<m:copy>`: simple copy;
- `<m:get>`: simple substitution;
- `<m:set>`: variable attribution, used in indirect substitution;
- `<m:if>`: conditional;
- `<m:iterate>`: repetition;
- `<m:include>`: inclusion of another template; and
- `<m:file>`: creation of a new output file.

The new tags reproduce the behavior of their corresponding original tags. For example, `<m:if>` works in the same way as `<c:if>`. But the new tags also generate annotations (Java comments) that mark relevant code locations, according to each mapping type. For example, the tag `<m:copy>` copies a piece of code from the template to the generated code, and automatically inserts a comment in the beginning and end. The tag `<m:iterate>` implements repetition, marking the beginning and end of the generated code, and also the start of each iteration. Figure 9 shows these two examples.



Type 1 mapping

templ1.jet

```
<m:copy>
s = "abc"
x = y++
if(x>32){
   y--;
}
else y=0;
</m:copy>
```

Generated code

```
// T1:templ1.jet#1
s = "abc"
x = y++
if(x>32){
   y--;
}
else y=0;
// T1:templ1.jet#1
```

Type 4 mapping

templ2.jet

```
s = "abc"
<m:iterate>
x = y++
print(s+":"+x);
</m:iterate>
x = 0
```

Generated code

```
s = "abc"
// T4:templ2.jet#1
// T4:i=1
x = y++
print(s+":"+x);
// T4:i=2
x = y++
print(s+":"+x);
// T4:i=3
x = y++
print(s+":"+x);
// T4:templ2.jet#1
x = 0
```

Figure 9. Mapping types 1 and 4 implemented using JET custom tags

For type 1, the inserted comment (`//T1:templ1.jet#1`) indicates the type of mapping (`T1`), the template that generated this piece of code (`templ1.jet`), and a unique identifier (number 1 after character `#`). For type 4, similar comments surround the generated code (`//T4:templ2.jet#1`). In addition, other comments (`//T4:i=1`, `//T4:i=2` and `//T4:i=3`) mark the places where each iteration starts.

With the new tags, it is possible to generate exactly the same type of code as with the original tags, but with the side effect that now the generated code is completely annotated, and the mapping between generated code and the templates is established.

The result is that, for each piece of generated code, it is possible to identify the mapping type, the template that generated this particular piece of code and even which part of the template was responsible for it, by locating the nearest surrounding generated comments. Code migration can be made based on this information.

4. Concluding remarks and future work

This paper presented an initial step towards automated code migration. The next step is to identify the different change possibilities that may occur in the reference implementation, and to develop a mechanism that attempts to automatically replicate each type of change made in the reference implementation back into the templates, using the mapping information to do that. Some of the mapping types clearly allows automatic code migration, like simple copy, while others, like repetition, are more complex. Suppose, for example, that the developer modifies only one of three iterations. This may arise the need for a conditional tag, in addition to the code migration.

There are many potential problems with this approach. For example, if the user accidentally removes a comment during a change, the mapping may be lost. There are ways to avoid this problem, like protecting this particular line from modifications, but further studies are needed to determine exactly what is needed. The important is to stress that the goal is not to achieve 100% automatic code migration, what seems to be unlikely to succeed. Even partial automation has impact in many MDE-based projects. This more realistic goal is the main contribution pursued in this research.

References

Bierhoff, K., Liongosari, E. S., and Swaminathan, K. S. (2006). Incremental development of a domain-specific language that supports multiple application styles. In *6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*, pages 67–78.

Cleaveland, J. C. (1988). Building application generators. *IEEE Software*, 7(1):25–33.

Cleaveland, J. C. (2001). Separating concerns of modeling from artifact generation using XML. In *1st OOPSLA Work. on Domain-Specific Visual Languages*, pages 83–86.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Hettel, T., Lawley, M. J., and Raymond, K. (2008). Model synchronisation: Definitions for round-trip engineering. In *ICMT 2008*, volume 5063/2008 of *Lecture Notes in Computer Science*, pages 31–45. Springer.

Muszynski, M. (2005). Implementing a domain-specific modeling environment for a family of thick-client GUI components. In *The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA*, pages 5–14.

Paesschen, E. V., Meuter, W. D., and D'Hondt, M. (2005). Selfsync: A dynamic round-trip engineering environment. In *MoDELS conference*, volume 3713/2005 of *Lecture Notes in Computer Science*, pages 633–647. Springer.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31.

Visser, E. (2007). WebDSL: A case study in domain-specific language engineering. In *GTTSE 2007*, volume 5235 of *Lect. Notes in Computer Sci.*, pages 291–373. Springer.

Wimmer, M., Strommer, M., Kargl, H., and Kramler, G. (2007). Towards model transformation generation by-example. In *40th Hawaii International Conference on System Sciences (HICSS'07)*, pages 285–294, Hawaii.