Using ATL to support Model-Driven Development of RubyTL Model Transformations

Álvaro Jiménez, David Granada, Verónica Bollati, Juan M. Vara

Kybele Research Group,
Department of Computing Languages and Systems, Rey Juan Carlos University,
C/ Tulipán s/n, 28933, Móstoles, Madrid (Spain).
{alvaro.jimenez, david.granada, veronica.bollati, juanmanuel.vara}@urjc.es

Abstract. Model transformations are the main artefact in any Model-Driven Engineering proposal. However, being software artefacts more effort should be dedicated to apply model-driven principles in the development of model transformations. In this context, this work presents some tooling to ease the model-driven development of RubyTL model transformations. In particular, we present a metamodel for RubyTL, a model transformation to move from high-level to RubyTL transformation models and finally a TCS injector/extractor to move from RubyTL models to RubyTL source-code and back.

Keywords: Atlas Transformation Language (ATL), Model-Driven Engineering (MDE), Model Transformations, Transformation Models, Textual Concrete Syntax (TCS), RubyTL.

1 Introduction

With the advent of Model-Driven Engineering (MDE) [3] the role of models has changed drastically since they became the main artefact along the development process. In such context, model transformations are typically the link between the different steps of the process. In the context of Model-Driven Software Development (MDSD) [16] such transformations aim at lowering the abstraction level of the target models until they can be (almost) serialized into working code. In other fields, such as model compare, transformations are used to generate difference models [13]. Despite the processing task for which transformations are developed, there is no doubt of their key role in any MDE proposal.

As a response, a number of languages and tools to develop model transformations have arisen during the last years (see [7] or [18] for detailed reviews). They differ in many aspects [7], such as the preferred approach (declarative, imperative, graph-based, etc.), tooling support (complete IDEs, command-line tools, etc.), underlying metamodelling framework (EMF, MDR, built-in, etc.) and so on. This diversity brings additional complexity to the development of model transformations.

In order to: a) address the inherent complexity of model transformations development and b) alleviate the problem of the diversity of available languages for model transformation, we advocate in favour of applying MDE principles to the

development of model transformations. In particular, we adopt the idea collected in [4]. Handling model transformations as transformation models we can process them as any other model, i.e. we can generate them, transform them, merge them, simulate their execution or perform any other model processing task. Unfortunately, there are not many languages that have adopted such approach, apart from ATL [9].

Besides, it might bring interoperability to the scope of existing model transformation languages: if we can inject a model transformation coded with the `FOO` language into a model, then we can map such model into another one conforming to the metamodel of the `BAR` language. Next, we can extract such model into the corresponding `BAR` working-code.

Moreover, if we are able to define a high-level metamodel for a set of model transformation languages, we should be able to use it as a pivot metamodel to bridge such languages.

In this line, the main contributions of this work are: a) the specification of an EMF metamodel for RubyTL [14], a hybrid model transformation language; b) the development of an ATL transformation to map high level specifications of model transformations into RubyTL models and c) the development of an injector/extractor for RubyTL using TCS [8]

The rest of this paper is structured as follows. Section 2 presents the proposed development process which includes an ATL transformation defined to obtain RubyTL models from high abstraction level transformation models. In order to achieve this goal, also, we present a metamodel according to RubyTL language. Finally, we describe the mechanism to translate transformation models into code, and code into transformation models. Section 3 uses a case study to illustrate the tool developed to support the proposal. Finally, in Section 4, we conclude by summarizing the contributions and outlining future works.

2 Model-driven development of RubyTL transformations

This work has been addressed in the context of MeTaGeM, a methodological and technical framework for model-driven development of model transformations [5]. In particular, the tasks addressed here constitute part of the proof of concept provided for MeTaGeM. This way, Fig. 1 provides an overview of the process proposed for the development of RubyTL transformations.

At the PSM-level (*Platform Specific Model*) we define a high-level transformation model that conforms to a high-level transformation metamodel (see [5] for a detailed insight). We refer to this metamodel as platform-specific metamodel since it is intended to abstract the main concepts handled by model transformation languages adopting a hybrid approach, such as ATL or RubyTL. New metamodels can be defined for graph-based, pure imperative or pure declarative model transformation languages.

Next, an ATL model transformation consumes the previous model and generates a PDM (*Platform Dependent Model*) transformation model that conforms to the metamodel of the targeted transformation language, in this case, the RubyTL

metamodel. Finally, the low-level transformation model is serialized into RubyTL source code by means of the TCS injector/extractor.

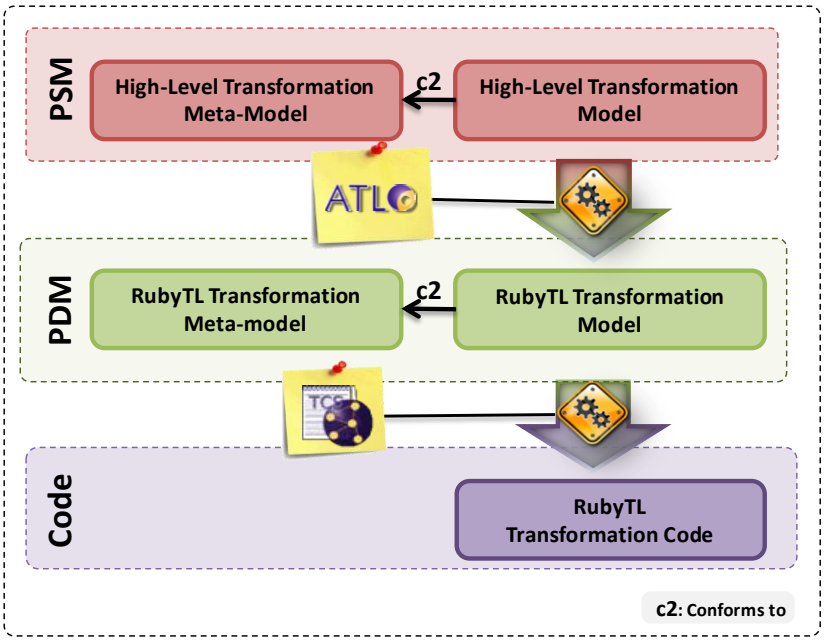

Fig. 1 – Model-driven development of RubyTL transformations

2.1 A metamodel for RubyTL

Although some partial specifications can be found in existing literature [14, 15], so far there is no complete metamodel for RubyTL. Therefore, in order to be able to generate and process RubyTL transformation models we have had to first specify and implement such metamodel. To do so, we have analysed previous works around RubyTL and we have count with the help of their developers. Fig. 2 shows the new metamodel, for more details see [5].

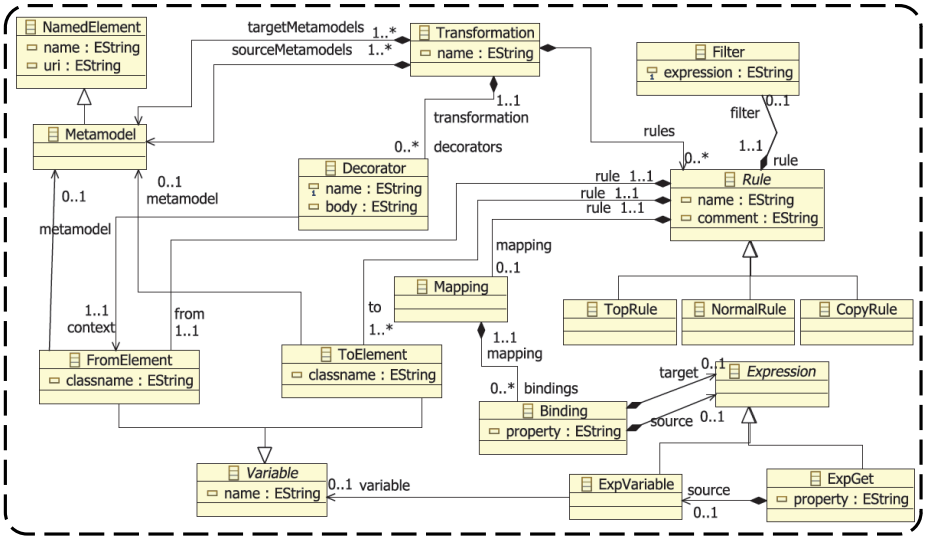

Fig. 2 – RubyTL metamodel specification

2.2 From high-level transformation models to RubyTL transformation models

This section focuses on the development of the model transformation that allows synthesizing high-level transformation models into RubyTL transformation models.

With regard to the development of model transformations, in [12] it is stated that "the mapping description may be in natural language, an algorithm in an action language, or a model in a mapping language". Accordingly, in previous works [17] we sketched a generic process to address the development of model-to-model transformations:

- First, the mappings between models are defined using natural language.
- Next, those mappings are structured by collecting them in a set of rules, expressed again in natural language.
- Then, these mapping rules are formalized using graph grammars.
- Finally, the resulting graph transformation rules are implemented using one of the existing model transformation approaches. In particular we decided for using ATL from the very beginning. Nowadays such decision has proven to be right since ATL is considered the *de-facto* standard for model transformations. Indeed, we have compared it with other engines such as QVTo [6] and we have concluded that it remains the most convenient in terms of available documentation, tooling support and cases of success.

According to this process, Table 1 collects the mapping rules to move from high-level transformation models to RubyTL transformation models. It is worth mentioning that these rules are the result of a continuous refining process that might continue during the next months.

High-Level Transformation Meta-model		RubyTL Meta-model	
Module		Transformation	
InMetaModel		MetaModel (Input)	
OutMetaModel		MetaModel (Output)	
Rule	isMain = true and in = 1	Rule	TopRule
	isMain = true and in > 1		TopRule (use of allObjects method in Rule.filter)
	isMain = false and typeAttribute <> #unique and in = 1		CopyRule
	isMain = false and typeAttribute <> #unique and in > 1		CopyRule (use of allObjects method in Rule.filter)
	isMain = false and typeAttribute = #unique and in = 1		NormalRule
	isMain = false and typeAttribute = #unique and in > 1		NormalRule (use of allObjects method in Rule.filter)
	in = 0		Static Method of Ruby
SourceElementRule		FromElement	
TargetElementRule		ToElement	
ElementIncluded - LeftPattern - RightPattern		Binding - ExpGet (Left side) - ExpGet (Right side)	
Operation - Operation.body + Return.datatype		Decorator - Decorator.body + dataType.toString()	

Table 1 – Mapping rules: from High-Level to RubyTL transformation models

According to the above-described process, next step was the formalization of the mapping rules using graph grammars [1]. Again, the complete set of graph transformation rules can be found in [5].

To provide with an example, Fig. 3 shows the graph transformation rule to map Rule elements from the PSM to TopRule elements in the PDM: whenever a Rule element, whose IsMain property is set to true, is found on the PSM (①), a TopRule element is added on the PDM (①'). The properties From and To of the new TopRule element are initialized with references to the FromElement and ToElement elements

of the PDM. These are created from the source elements `SourceElementRule` and `TargetElementRule` respectively (②⇨②' and ③⇨③').

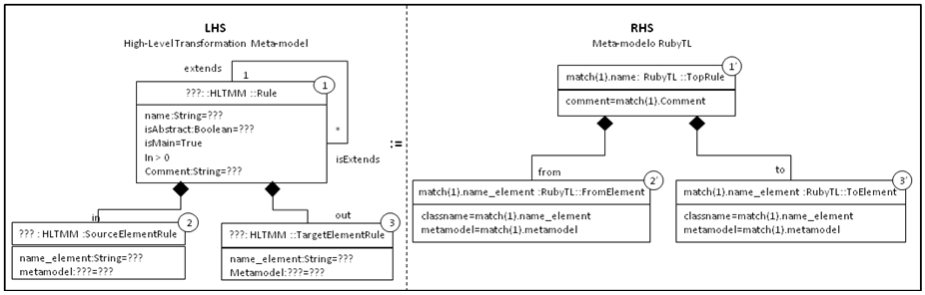

Fig. 3 – Graph Transformation Approach: Rule `Rule2MatchedRule`

2.3 Implementation

Based on the rules shown in Table 1 and the formalization of them using graph grammar, we have implemented it using ATL language [9]. As an example, Fig. 4 shows the ATL rule that implements the transformation between the `Module` elements of the high-level transformation metamodel and `Transformation` elements of the RubyTL metamodel. In `Transformation` element are defined the `name` property, the references: `sourceMetamodels` and `targetMetamodels`, which indicate, respectively, the source and the target metamodels involved in the model transformation; `rules`, which represent the transformation rules of the module; and `decorators`, which represent the functions that can be defined.

```
rule Module {
   from
      mm_hybrid : MM_Hybrid!Module
   to
      rubytl : RubyTL!Transformation (
        name <- mm_hybrid.name_module,
        sourceMetamodels <- mm_hybrid.inMM,
        targetMetamodels <- mm_hybrid.outMM,
        rules <- mm_hybrid."rule",
        decorators <- mm_hybrid.operations)
}
```

Fig. 4 - ATL transformation rule: `Module`

The `Rule` meta-class in RubyTL is defined as `abstract`, and it is specialized by: `TopRule`, `NormalRule` and `CopyRule`. Therefore, it is necessary to define three different kinds of rules, establishing a guard condition in each one. This guard condition evaluates the `isMain` property and the return value of the helper `getSizeIP()`, which verifies the number of `SourceElementRule` elements dependent of the element `Rule`.

For example, Fig. 5 shows the one of these ATL rules, the rule which map the `Rule` element at PSM model to the `TopRule` element at PDM model. The

createRule2TopRuleMulti rule states that for every Rule found in the source model, a TopRule, a Filter and Mapping elements are created in the target model; when the isMain property, of the Rule, is True and the return value of the helper getSizeIP() is greater than one, that is the Rule element has more than one SourceElementRule. A set of direct bindings initialize some attributes of the Rule, such as the name, which indicate the name of the rule; from, which indicate the source element; to, which indicate the target element; comment, where it is possible to define a comment.

Moreover, it is necessary to define the following elements: mapping, which combines the properties of the target element; and filter, which performs a verification of the existence of all elements of type SourceElementRule. In order to obtain this verification, we define a helper getFilterMultiIN().

```
rule createRule2TopRuleMulti{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()>1
     and mm_hybrid_rule.isMain=true)
   to
     rubytl : RubyTL!TopRule (
      name <- mm_hybrid_rule.name_rule,
      "from" <- mm_hybrid_rule."in".asSequence().first(),
      "to" <- mm_hybrid_rule.out.asSequence(),
      comment <- mm_hybrid_rule.getComment(),
      filter <- afilter,
      mapping <- amapping),

     afilter : RubyTL!Filter(
      expression <- mm_hybrid_rule.getFilterMultiIN()),

     amapping : RubyTL!Mapping (
      bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Fig. 5 – ATL transformation rule: createRule2TopRuleMulti

Fig. 6 shows an ATL transformation rule that generates Binding elements (PDM), from ElementIncluded elements (PSM). This rule establishes the relationship between a source element (*right-hand side*) and a target element (*left-hand side*) at PDM level.

As Fig. 6 shown the source element is generated by the helper defineBinding() (Fig. 7) that defines the source element by means a set of conditions and calls to others helpers. These conditions verify if in the source defined at PSM level, there are calls to other rules, operations, references or is defined a constant value.

```
rule Bindings {
  from
    elemInc : MM_Hybrid!ElementIncluded
  to
    rubytl : RubyTL!Binding (
      --Right side of formula, that has the value - issues
      source <- elemInc.defineBinding(),
      --Left side of formula, that will receive the value
      target <- atargetvalue  ),
    atargetvalue : RubyTL!ExpGet(
      --property of target
      property <-
            elemInc.left.targetElement.asSequence().first().name_element,
      source <- asourcename),
    asourcename : RubyTL!ExpVariable(
      variable <- avariabletrg),
    avariabletrg : RubyTL!ToElement (
      name <-
            elemInc.refImmediateComposite().name_element.toLower()+'_out')
}
```

Fig. 6 – ATL Transformation rule: `Bindings`

The `target` element of the `Bindings` rule is defined by setting `property` and `value` attributes, which are defined with the generation of the `ExpGet, ToElement` and `ExpVariable` elements, as shown in Fig. 6.

```
-- Helper -> To call the correct lazy rule to define the Binding
helper context MM_Hybrid!ElementIncluded def:defineBinding():ATL!OclExpression=
    if (self.right."rule".asSequence().first().oclIsUndefined()
        and self.right.operation.asSequence().first().oclIsUndefined()
        and self.right.sourceElement.asSequence().first().oclIsUndefined()
        and self.right.reference.oclIsUndefined()) then
            thisModule.getConcreteBinding(self)
    else
        if (not self.right.reference.oclIsUndefined()) then
            if self.right.reference.oclIsTypeOf(MM_Hybrid!SourceElementRule)
            then
                  thisModule.getComplexBinding(self)
            else
                  thisModule.getSimpleBinding(self)
            endif
        else
            thisModule.getComplexBinding(self)
        endif
    endif;
```

Fig. 7 – ATL Transformation helper: `defineBinding`

2.4 Code Generation

The next step was to serialize the RubyTL transformation models generated into working-code. To that end, we have opted for the Textual concrete Syntax (TCS) [8]. TCS provides with a DSL for the specification of the correspondence between the metamodel and its textual representation. From that, an ANTLR grammar together with a parser for this grammar is generated. Such parser (also known as injector) takes as input a textual program of the DSL and generates a model conforming to the DSL

metamodel. In addition, TCS also generates an extractor that provides with model-to-text capabilities.

In order to illustrate this task, Fig. 8 shows a simple example. In particular, the `Transformation` template specification. This template defines the concrete syntax for `Transformation`, `sourceMetamodel` and `targetMetamodel` model elements `from` RubyTL models (Fig. 8 (a)). The serialization of these elements serve to compose the header of a RubyTL transformation, that include the name of the transformation and the name of the source and target metamodels (Fig. 8 (b)).

Also, if the transformation contains `decorators` or `rules`, this template invokes the execution of the respective `templates`.

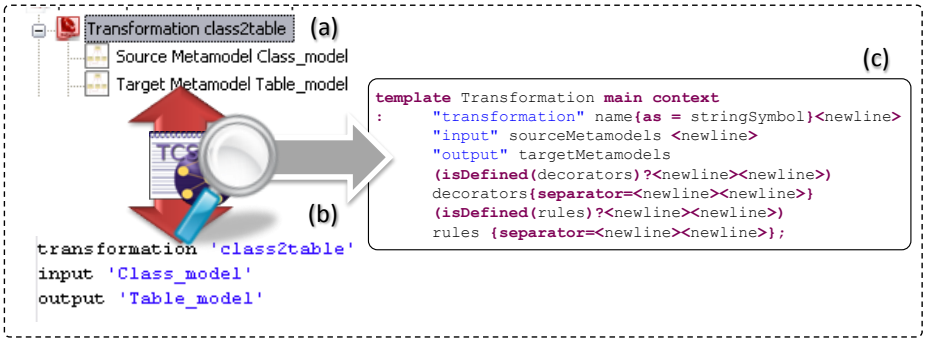

Fig. 8 – Generating code and models with TCS

2.5 Current Limitations

The code generated by the current version of the prototype developed is not fully executable in all cases. There are some scenarios in which the developer needs to manually refine the generated code, adding or modifying some code excerpts in order for the transformation to execute correctly.

In this sense, one of the main issues we have found is that RubyTL does not allow to define multiple input patterns in a rule. For instance, Fig. 9(a) shows an example of a desirable rule containing a multiple input pattern in RubyTL. Since the language does not support this construction, to *simulate* it we should proceed as follows (see Fig. 9(b)):

- First, creating a guard which performs a cartesian product between all the source elements.
- Next, adding the `many` function to the target pattern.
- Finally, merging each pair of objects that resulted from the cartesian product.

(a)

```
# Desirable implementation - Not Allowed
rule 'RuleTest' do
   from InMM::Src1, InMM::Src2
   to OutMM::Trg1
   mapping do | src1, src2, trg1 |
       trg1.name  =  src1.name_element
       trg1.type  =  src2.type_element
   end
end
```

(b)

```
# Solution. Using Cartesian product
rule 'RuleTest' do
  from InMM::Src1
  to many(OutMM::Trg1)
  mapping do | src1, trg_set |
    trg_set.values = InMM::Src2.all_objects.map
    do |src2|
       trg1 = OutMM::Trg1.new
       trg1.name  =  src1.name_element
       trg1.type  =  src2.type_element
       trg1 #trg_set.values = trg1
    end
  end
end
```

Fig. 9 – Multiple input patterns limitation

Besides, the rule shown in Fig. 9(b) uses some constructions, such as `many` or `new`, provided by the host language, i.e. Ruby. Such constructions have not been collected in the presented metamodel for RubyTL. Hence, to address this issue we should extend the metamodel to support the modelling of some Ruby elements.

Another issue is related with the TCS injector/extractor. The concrete syntax of RubyTL hampers the binding of variables and elements from the transformation at the time of injecting it to a model. For instance, according to the RubyTL metamodel shown in Fig. 2, the `src1` variable should produce a `variable` object, whose `classname` property should be `Src1` and its metamodel reference should point to the `InMM` object. However, due to the concrete syntax of RubyTL, we have not been able of establishing such binding when generating the RubyTL model. Probably we might address this issue modifying the RubyTL metamodel, but we are still considering some other solutions.

Finally, the current prototype was developed atop of older Eclipse and ATL versions because the current versions were not stable when it was developed. Hence, update tasks will be also addressed in the near future.

3 Case study

This section presents a simple case study in order to validate our proposal. To that end, we use the traditional scenario of mapping object models to relational models [13]. Fig. 10 provides an overview of the source and target metamodels.

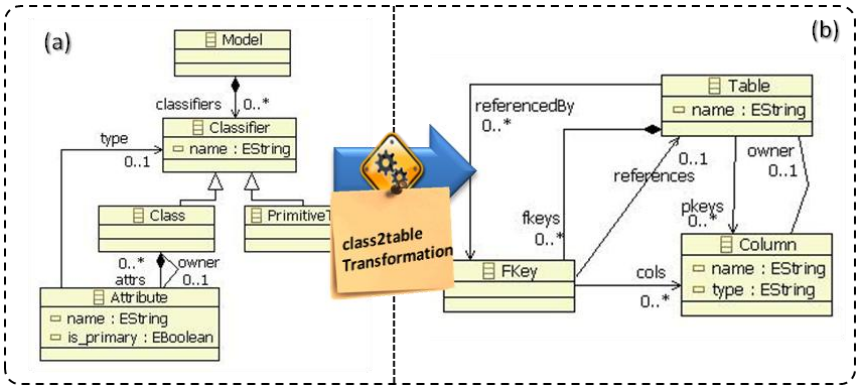

Fig. 10 – Class and Table metamodels

In order to develop a transformation between these metamodels using our proposal, the user must first create a high-level transformation model. Next, the ATL transformation presented in the previous section maps such model into a RubyTL transformation model. As an example, Fig. 11 focuses on the `class2table` rule. This rule enables to transform a class and its attributes into a table and its respective columns.

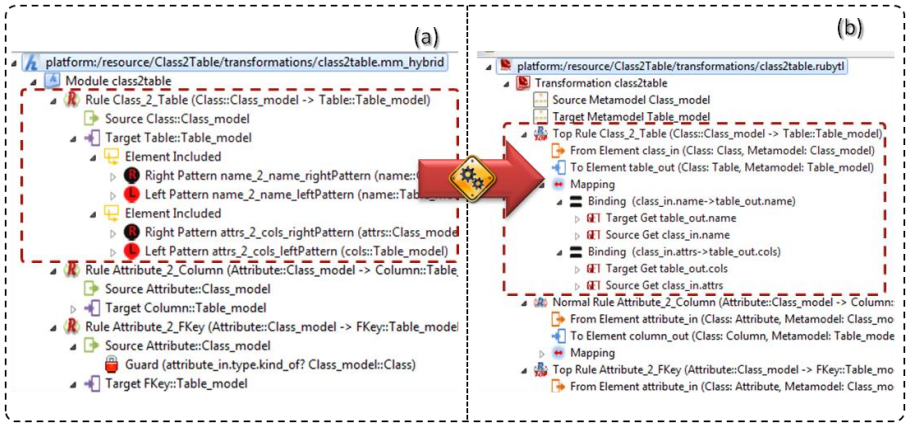

Fig. 11 – From high-level to RubyTL transformation model - `class2table`

Finally, such model is serialized into the RubyTL source code. To illustrate this step, Fig. 12 focuses in the mapping of `TopRule` objects into source code.

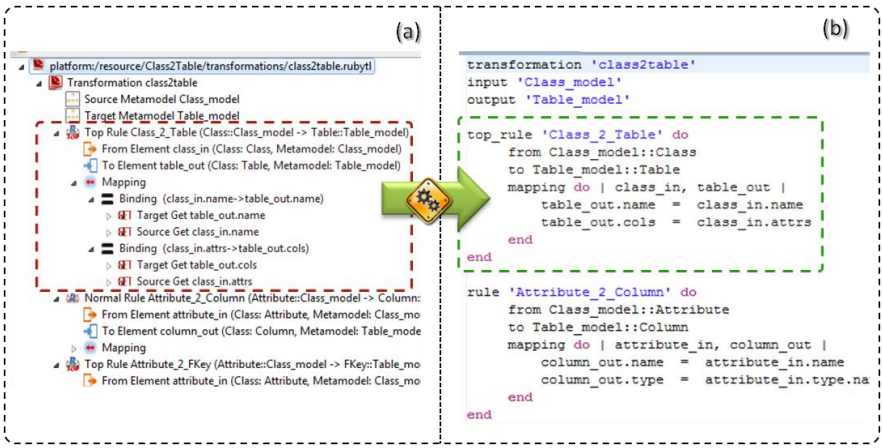

Fig. 12 – From RubyTL model to RubyTL code - `class2table`

The complete case study is available to view and download at the MeTAGeM Web site[1].

Finally, we would like to note that, though this work have used a simple case study to illustrate the prototype developed, we have used it to address more complex scenarios which are also available at the MeTAGeM Web site. For example, in [5] we have developed the *UML2ORDB4ORA* transformation that allows generating ORDB models from conceptual data models expressed by means of UML class diagrams [19].

4 Conclusion and further work

Model transformations play a cornerstone role in any MDE proposal. However, despite the impact of MDE and the relevance of model transformations, there has not been too many works oriented to apply MDE principles to the development of model transformations so far. Indeed, their inherent complexity and the existence of a wide set of model transformation languages make the development of transformations an ideal scenario to apply MDE techniques. This way, we should be able to: a) bring simplicity to the development process b) shorten the distance between different model transformation languages. In the end, a transformation is another software product and thus, subject to the application of MDE techniques to its development.

In order to put this idea into practice, in this work we have introduced the specification of a metamodel for RubyTL, a model transformation language that shares many similarities with ATL. Besides, we have developed a model transformation to map high-level transformation models into RubyTL transformation models. Likewise, we have developed a TCS injector/extractor to move forward and backward from the RubyTL model to RubyTL source-code. Finally, we have shown

[1] http://metagem.wordpress.com/

how this tooling is used in a classical case study (providing with pointers to more elaborated ones that can be downloaded from the Web).

This work provides with a number of directions for future work. The most immediate is to apply the same proposal to support other model transformation languages, such as EpsilonTL [10]. Besides, as long as there are available metamodels for other transformation languages we should be able to develop mappings between them. This way, we should be able to migrate any transformation (semi)-automatically. Besides, as long as we specify high-level transformation metamodels, they could be used as pivot metamodels to move between different model transformation engines. For instance, in this work we use a metamodel for hybrid languages that collects the main abstractions shared by the most adopted hybrid model transformation languages. The same approach can be applied to bridge graph-based or imperative languages. We could even bridge those high-level metamodels to support interoperability not only between languages following the same paradigm, but also between those following different paradigms.

More concrete future works are planned regarding the prototype presented in this work. In particular, to update the prototype to current versions of Eclipse and ATL; to improve the first draft of the RubyTL metamodel and finally, testing other existing tools for injection and extraction, such as Gra2MoL and Acceleo[2].

Acknowledgments. This work is partially funded by the MODEL CAOS project, financed by the Spanish Ministry of Science and Technology (Ref. TIN2008-03582), Agreement Technologies (CONSOLIDER CSD2007-0022) and Technical Support Staff Subprogram (MICCINN-PTA-2009), which is partially financed by the Spanish Ministry of Science and Innovation.

References

1. Baresi, L. and Heckel, R. 2002. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In Proceedings of the First international Conference on Graph Transformation). LNCS 2505. Springer-Verlag, London, pp. 402-429, 2002.
2. Bézivin, J., Rumpe, B., Schürr, A., & Tratt, L. (2005). *Mandatory Example Specification*. CFP of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica.
3. Bézivin, J.: In search of a Basic Principle for Model Driven Engineering. Novatica/Upgrade, V (2), 21-24 (2004).
4. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models!. 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2006, (2006).
5. Bollati, V.: MeTAGeM: Entorno de Desarrollo de Transformaciones de Modelos Dirigido por Modelos. Ph. D. Thesis. Rey Juan Carlos University (2011).
6. Bollati, V. A., Sánchez, V., Vela, B., Marcos, E.: Análisis de QVT Operational Mappings: un caso de estudio. VI Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones, DSDM (2009).

[2] www.acceleo.org/

7. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. Proceedings of OOPSLA'03, workshop on Generative Techniques in the Context of MDA (2003).
8. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. Generative Programming and Component Engineering, 5th International Conference, GPCE (2006).
9. Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, *72*(1-2), 31-39.
10. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. Proceedings of 1st International Conference on Model Transformation, Zurich, Switzerland (2008).
11. Mens, T. and Van Gorp, P. 2006. A Taxonomy of Model Transformation. Electron. Notes Theor. Comput. Sci. 152 (March 2006), 125-142.
12. OMG. MDA Guide Version 1.0. Document number omg/2003-05-01. Ed.: Miller, J. and Mukerji, J. Retrieved from: http://www.omg.com/mda, 2003.
13. Rose, L.M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D., Garcés, K., Paige, R., and Polack, F. A Comparison of Model Migration Tools. In Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I (MODELS'10), Springer-Verlag, Berlin, Heidelberg, 61-75.
14. Sánchez, J., García, J., Menarguez, M.: RubyTL: A Practical, Extensible Transformation Language. European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA (2006).
15. Sánchez, J., García, J., Sánchez, E. V., Estévez, A.: RubyTL a ATC: un caso real de transformación de transformaciones. IV Taller sobre Desarrollo de Software Dirigido por Modelos, DSDM (2007).
16. Stahl, T., Volter, M., & Czarnecki, K. (2006). Model-Driven Software Development: Technology, Engineering, Management: John Wiley & Sons.
17. Vara, J. M., Vela, B., Cavero, J. M., & Marcos, E. (2007). Model Transformation for Object-Relational Database development. SAC '07: *Proceedings of the 2007 ACM symposium on Applied computing*, 1012-1019.
18. Vara, J. M.: M2DAT: A Technical Solution for Model-Driven Development of Web Information Systems. Ph. D. Thesis. Rey Juan Carlos University (2009). http://hdl.handle.net/10115/5145.
19. Vara, J. M., Vela, B., Bollati, V., Marcos, E.: Supporting Model-Driven Development of Object-Relational Database Schemas: A Case Study. Proceedings of International Conference on Model Transformation, ICMT (2009).