*trans*ML: A Family of Languages to Model Model Transformations

Esther Guerra[1], Juan de Lara[2], Dimitrios S. Kolovos[3], Richard F. Paige[3], and Osmar Marchi dos Santos[3]

[1] Universidad Carlos III de Madrid (Spain), eguerra@inf.uc3m.es
[2] Universidad Autónoma de Madrid (Spain), Juan.deLara@uam.es
[3] University of York (UK), {dkolovos, paige, osantos}@cs.york.ac.uk

Abstract. Model transformation is one of the pillars of Model-Driven Engineering (MDE). The increasing complexity of systems and modelling languages has dramatically raised the complexity and size of model transformations. Even though many transformation languages and tools have been proposed in the last few years, most of them are directed to the *implementation* phase of transformation development. However, there is a lack of cohesive support for the other phases of the transformation development, like requirements, analysis, design and testing.
In this paper, we propose a unified family of languages to cover the life-cycle of transformation development. Moreover, following an MDE approach, we provide tools to partially automate the progressive refinement of models between the different phases and the generation of code for specific transformation implementation languages.

1 Introduction

Model-Driven Engineering (MDE) relies on models to conduct the software development process. In this way, high-level models are refined using automated transformations until the code of the final application is obtained. A key aspect in MDE is automation of operations applied to models (i.e. model management). In particular, there is a recurring need to transform models between different languages and levels of abstraction, e.g. to migrate between language versions, to translate models into semantic domains for analysis, to generate platform-dependent from platform-independent models, or to refine and abstract models. This kind of transformation is called Model-to-Model (M2M) transformation.

In MDE, transformations are seldom specified with general-purpose programming languages (e.g. Java) but with M2M transformation languages specially tailored for the task of transforming models [3]. Prominent examples of such languages are QVT [12], ATL [1], Triple Graph Grammars [15] and ETL [10].

M2M transformations are deployed as software and, like any other software, they need to be analysed, designed, implemented and tested. Therefore, their development requires systematic engineering processes, notations, methods and tools. This need is more acute in industrial projects, where the complexity of

models and modelling languages makes necessary large and complex transformations. Surprisingly, most transformation languages proposed by the MDE community are either directed towards the *implementation* phase of transformations or are not integrated in a unified engineering process. As a consequence, there is a lack of cohesive support for transformations – involving notations, methods and tools – across all development phases. This makes more difficult the design of large-scale transformations, hinders the standardization and codification of best practices (e.g. patterns analogous to design patterns in UML), and complicates the maintenance and understandability of the transformation code.

In this paper we present a family of modelling languages, called *trans*ML, which covers the whole life-cycle of transformation development: requirements, analysis, design and testing. It can be used together with any transformation implementation language. Moreover, following an MDE approach to the construction of transformations, we provide partial automation for the refinement of *trans*ML models and the generation of code for specific transformation implementation languages. We also provide support for reengineering transformation code by its parsing into *trans*ML models, and facilitating platform migration.

Paper organization. §2 discusses previous attempts to model M2M transformations, pointing out limitations. Next, §3 proposes a set of languages that cover the identified needs to build transformations in the large. §4 presents tool support for forward and reverse transformation engineering, followed by §5, which evaluates the approach with an industrial case study. Finally, §6 concludes.

2 Related Work

Most recent research in M2M transformation has focused on the implementation phase, either to develop new implementation languages, or to test final implementations. This is likely due to the infancy of M2M transformation research, and is analogous to early research on software engineering languages where the focus was directed to implementation languages. There, analysis and design notations came later, when issues of system scale became a concern.

Only a few proposals for design notations for transformations can be found in the literature. For example, [13] presents a language to design transformations, but focusing only on their implementation. Another example is [4], which covers the low-level design of transformations, being able to represent the structure of rules using diagrams similar to UML class diagrams.

Closer to our engineering view of building transformations are the works that consider several phases of development. For example, [16] identifies a transformation development life-cycle and proposes describing transformations incrementally, starting from transformational patterns and partial specifications of transformations, which are gradually refined. However, no concrete notation or tool is proposed. The position paper [11] envisages a mapping and a transformation view for transformations. Its aim is providing a precise semantics for mappings in terms of Petri nets so that the transformation view can be generated from the

mappings view. Still, the framework is ad-hoc for their particular transformation approach and cannot be applied to other implementation languages.

Finally, there is limited work on languages to express composition of transformations; this can be viewed as a kind of architectural design [14, 18] through the definition of new architectural languages. Whereas [14] is a specific language for composing ATL transformations, in [18] the approach is more platform independent. In both cases, other phases of transformation development are neglected.

In summary, we observe a lack of modelling notations and tools to cover the complete life-cycle of transformation development in a cohesive way. Transformation developers should be able to use such notations with their favourite transformation implementation languages, in the same way as the UML can be used with any object-oriented programming language. Having available such transformation modelling notations would make possible to apply systematic engineering principles to transformation development, to trace the models in the different stages of the development (in a non ad-hoc way), as well as to apply MDE techniques to obtain transformation code from high-level models. Such notations are urgently needed in order to be able to benefit from proven software engineering principles, like design patterns [2, 8] for model transformations.

3 A Family of Languages to Model Transformations

How are transformations developed? The answer is too frequently *"in an ad-hoc manner"*. Jumping directly to an implementation language may be sufficient for simple transformations, but this approach is challenging in the large. If transformation technology is to be used in industry, transformations must be constructed using engineering principles. Hence, the process of transformation development should include other phases, in addition to coding and testing, namely: requirements, analysis, architectural design, high-level design and detailed design.

The notations to be used in these phases have to consider the specificities of model transformation development. Fig. 1 gives an overview of *trans*ML, the family of languages we propose, and shows how they are interrelated. In the upper part, the figure shows the family of proposed languages, made of a requirements diagram, formal specification diagrams and simple scenarios to cover the transformation analysis, architecture diagram, high-level design view of the transformation specified as a mapping diagram, and rule diagrams for the low-level design. The figure also shows relations to trace elements across diagrams (e.g. to discover the requirements each rule is addressing). The objective of these diagrams is guiding the construction of the software artifacts shown on the bottom of the figure: the transformation code (in any implementation language such as QVT or ETL), the generation of test cases, the run-time verification of transformation code and the orchestration of transformations.

We do not prescribe a particular process in which these phases should occur, but in our experience, transformations are often built in an iterative, incremental way. We also do not suggest that *all* diagrams have to be used when building any transformation (just like when building object-oriented systems it is not manda-

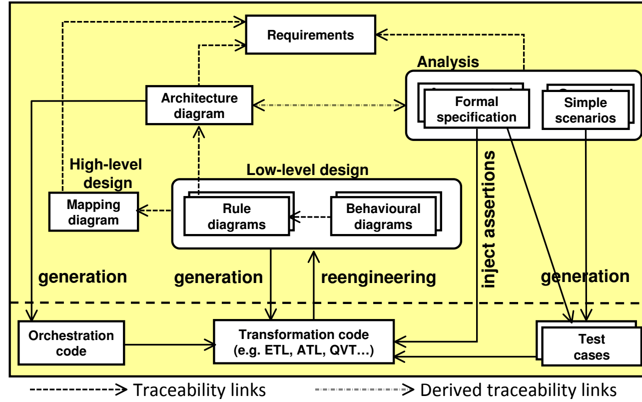

Fig. 1. Model transformation framework.

tory to use all UML diagram types). Depending on the project characteristics, we may emphasize the use of the formal specification language (e.g. for complex transformations that should preserve behaviour), or just use the high-level design diagrams but not the low-level ones for small, one-to-one transformations. However, the full power of *trans*ML comes by using its diagrams in combination.

Next we present *trans*ML in detail. We will use as an example the classical class-to-relational transformation to ease understanding, and provide evaluation of its use with a complex transformation in an industrial project in §5.

3.1 Requirements elicitation

Just like any other software, transformation developers need to record the transformation rationale, identifying functional and non-functional requirements. Therefore, notations helping the hierarchical decomposition of requirements and permitting traceability to further models are especially useful. Here we could use any technique and notation from the Requirements Engineering community. However, in order to trace requirements into subsequent phases, *trans*ML includes a representation of requirements in the form of diagrams, similar to SysML requirements diagrams[4]. The meta-model for this representation is shown to the left of Fig. 2, and enables hierarchical decomposition, classification, refinement and traceability of requirements. Requirements are classified in a dual way: attending to whether they are functional or not, and to whether their source is an input model, an output model or the transformation itself.

As an example, the right of Fig. 2 shows the requirements diagram for the class-to-relational transformation. Requirements for the input model are annotated with a right arrow in the upper right corner, whereas requirements of the transformation are annotated with dented wheels. Thus, requirement `0.1`

[4] `http://www.omg.org/spec/SysML/1.1/`

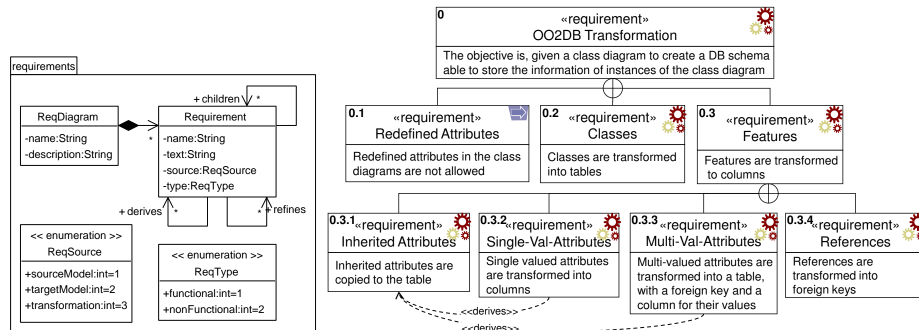

Fig. 2. Requirements meta-model (left). Requirements for the transformation (right).

restricts input models to have no redefined attributes. On the other hand, requirement `0.3.1` derives from requirements `0.3.2` and `0.3.3`.

3.2 Analysis

Software engineers use a variety of mechanisms to analyse, understand and reason about requirements. We have identified techniques based on scenarios and on formal specification languages, which we have adapted for *trans*ML.

First, once some requirements are fixed, engineers can write scenarios, which are examples of the transformation (similar to the role of uses cases in UML). We call these examples *transformation cases*, which describe how concrete source models are transformed into target ones. The examples may contain either full-fledged models or model fragments. As an example, the left of Fig. 3 shows a transformation case that explains the transformation of a multi-valued attribute into a table and a foreign key from the table associated to its owner class.

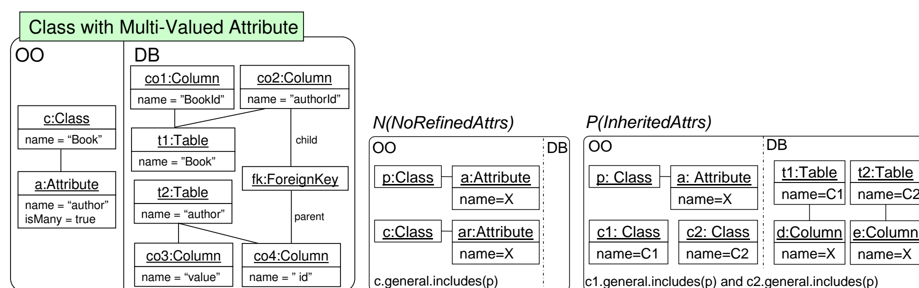

Fig. 3. Transformation case (left). Restriction on the input model (center). Verification property (right).

The purpose of *transformation cases* is twofold. First, they are used to understand and reason about what the transformation has to do. Second, they can be used as input to model transformation-by-example techniques [19] which derive a rough sketch of the transformation, and can also be used as test cases for the transformation implementation.

The second notation we use in this phase is a visual, formal *specification* language [7]. Similar to the role of Z [17] or Alloy for general software engineering, this language is used to: (i) describe in an abstract manner what the transformation has to do without stating how to do it, (ii) specify correctness properties that the implementation should satisfy, and (iii) specify restrictions on the input or output models. These specifications can be used later for formal reasoning of transformation requirements, and for specification-driven testing of transformations through the generation of an oracle function to test the transformation.

Our specification language abstracts from concrete examples, and is based on declarative patterns that express *allowed* or *forbidden* relations between two models [7]. Patterns have a graphical part, and can include constraints (we use EOL [9] for this). Patterns expressing allowed relations are called positive, while those expressing forbidden relations are called negative. Thus, the language supports constructive and non-constructive specification styles (in contrast to scenarios, which are always constructive). Moreover, patterns are bi-directional, so that they can be interpreted both source-to-target and target-to-source. This allows the specification of uni-directional and bi-directional transformations.

Since *trans*ML has been designed to be independent of the language used to implement the transformation, our specification language supports the two usual styles for M2M transformation: trace-based and traceless, depending on whether explicit traces are given between source and target elements or not. In the latter case, patterns are similar to QVT relations [12] and can include positive and negative graph pre- and post-conditions (*when* and *where* clauses respectively). Trace-based and traceless patterns have a formal semantics which allows answering correctness questions about specifications (e.g. whether there are conflicts between patterns). The details of the semantics of this language and its compilation into OCL for testing are available in [7].

As an example, the center of Fig. 3 shows a negative pattern (indicated by the `N(...)`) used to express a restriction on the input models. It *refines* requirement `O.1` in Fig. 2. The pattern checks the existence of two classes `c` and `p` such that `p` is an ancestor of `c`, having both an attribute with same name (represented by variable `X`). As the pattern is negative, models in which such pattern occurs are invalid. As we will see latter, code will be injected in the transformation to test whether a given input model qualifies for the transformation.

The right of the figure shows a pattern expressing a property of the transformation itself. The pattern is positive (indicated by the `P(...)`) and expresses that if a class `p` has two children classes `c1` and `c2`, then each attribute in the ancestor class `p` has to be replicated as a column in the tables associated to `c1` and `c2`. The tables in which the classes are transformed are located by equality of names (variables `C1` and `C2`), but any formula relating their names would also

be allowed. This kind of patterns will be used for the run-time verification of the transformation code, in order to check whether the implementation generates target models satisfying these properties.

3.3 Architecture

Large software is seldom monolithic, but is decomposed into interacting blocks. Hence engineers have to design its architecture. We have included a modelling language for architectural design which permits the modular decomposition of functional units. This is very useful in large-scale transformations, which need to be split in different units and orchestrated. Moreover, it is often the case that the transformation has to be integrated with further software components providing extra functionality, such as code generators. For the design of this language we have taken some ideas from works dealing with orchestration of transformations [14, 18], as well as from architectural description languages [6].

Our architectural language is made of *components* and connectors. Components interact through directional interfaces with a type given by meta-models, event types, actors or other components (to allow higher-order transformations). They can have a set of constraints, can be arranged hierarchically, and may represent transformations (model-to-model, model-to-text, text-to-model or in-place), software (a black-box) or actors (to model human intervention).

The left of Fig. 4 shows a simple architectural diagram for our example. The model depicts a chain of transformations: the first takes an OO model and transforms it into a DB model, the second optimises this DB model, and the third generates textual code for a particular platform. The diagram shows the transformations as components with typed, directional interfaces. The type of the interfaces is given by one or more meta-models, together with extra (OCL) constraints to rule out models which conform to the meta-model but are not handled by the transformation. Models conforming to those interfaces can be the input/output of the transformations. The type also allows checking compatibility when connecting two transformation components. The right of the figure shows a type-centric view of the same model. This view is similar to a mega-model [5], where transformation components are visualized as arrows connecting interface types. This architectural view can bridge modelware and grammarware technical spaces by including model-to-text and text-to-model transformations.

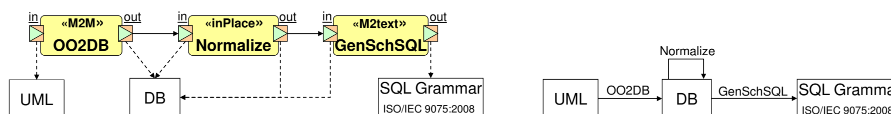

Fig. 4. Architectural diagram: transformation-centric and type-centric views.

3.4 High-level design: Mappings

The design of a transformation benefits from proceeding from a high to a lower level of abstraction, and therefore we provide different notations for them. The high-level design of a transformation is given by a *mapping diagram* that defines the mappings between the elements of the arbitrary number of languages involved in the transformation. This diagram provides an intuition of which is transformed into what, without giving details on the how, thus enabling the transition between analysis and design. This is similar to Triple Graph Grammar schemas [15], however our mappings are not intended to be used as an auxiliary tracing mechanism to guide the actual execution of the transformation code.

Fig. 5 shows to the left an extract of the mappings meta-model. A *mapping model* is established between several languages, each one defined by a meta-model. Mapping models can define the directionality of the transformation using the *navigable* attribute in *ModelEnd*s. Models are structured in packages, each one containing mappings, which can also be organized hierarchically. Mappings connect elements in the meta-models of the involved languages through *MappingEnds*. Mappings are provided with constraints, given either in uninterpreted text or in some language like OCL, expressing when mappings between elements should hold. The mapping meta-model refers to the meta-models of the languages involved in the transformation. We use an abstract class *ModellingElement*, which can be replaced by any concrete meta-modelling infrastructure.

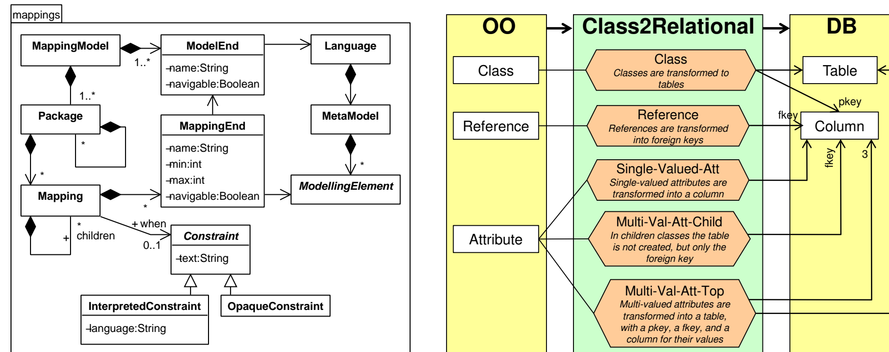

Fig. 5. Excerpt of mapping meta-model (left). Mapping diagram example (right).

The right of Fig. 5 shows a mapping diagram. It has one block for each language, containing the relevant elements of their meta-model. Another block includes mappings connecting some of these elements to indicate a causal relation between them. The links from the mappings to the language elements have a role name (e.g. fkey, pkey), a multiplicity (1 is assumed if it is empty) and a direction (to denote either access or creation of elements). As our transformation is unidirectional, mapping ends are depicted with arrows on the side of the DB.

Mapping diagrams can be used with different levels of detail. One can start with a rough sketch of the mappings and add details as the transformation is better understood. For example, in Fig. 5 we have omitted element `ForeignKey` of the DB meta-model. The mapping diagram is a high-level design notation, independent of the transformation implementation language. Moreover, it is not necessarily the case that a mapping has to be implemented by a unique rule and vice-versa. As we show next, we can use rule diagrams as a way to design the implementation of mappings if more details are needed before coding.

3.5 Low-level design: Rule structure and rule behaviour

Low-level detailed design diagrams indicate *how* the transformation has to be implemented. Here we separate the description of the rule structure from its behaviour. Hence, one or several *rule structure diagrams* may describe the structure of the rules in the transformation, and several *rule behaviour diagrams* attached to the rules can be used to specify what these rules should do. These notations will help in describing good practices and transformation patterns, in the same way as UML helps to record object-oriented patterns. Rule diagrams are also useful to generate code for different platforms, and reengineering of existing code.

Fig. 6 shows part of the meta-model of the rule structure diagrams. This kind of diagram depicts the structure (input/output parameters) of each rule, their execution flow, and data dependencies (e.g. parameter passing) between them. Rule diagrams refine mapping diagrams by giving the low-level design of how the specified mappings are to be realized. In this way, a rule can contribute to implement several mappings, and a mapping can be realized by several rules. Regarding rule structure, we can declare uni-directional or bi-directional rules, their involved domains and their parameters. Concerning the execution flow, we support both explicit flows (subclasses of `Flow`) as well as non-deterministic constructs found e.g. in graph transformation, as one can place a collection of rules inside a non-deterministic `Block`.

Fig. 7 shows to the left a rule structure diagram with four rules. The diagram is semi-collapsed, as it only shows the parameters of the `OO` domain. The diagram shows the rule execution flow by means of rounded rectangles (`Block` objects), in a notation similar to activity diagrams. Hence, the starting point is the block containing rule *Class2Table*, which implements the *Class* mapping. After executing this rule, the control passes to another block with three rules, to be executed in arbitrary order. In particular, rule *MultiValuedAtt2Table* has been designed to implement two mappings: *Multi-Val-Att-Child* and *Multi-Val-Att-Top*. In all cases, rules are applied at all matches of the input parameters.

Our rule language captures the main features of transformation languages. However, a particular rule diagram has to consider the specific implementation platform. For example, rules can have an arbitrary number of input parameters if we use ATL as the implementation language, whereas rules have only one input parameter if we use ETL, and we have patterns if using QVT-R. Also, platforms differ in the execution control of their rules. While in graph transformation the execution scheme is "as long as possible" (ALAP) and we can have rule priorities

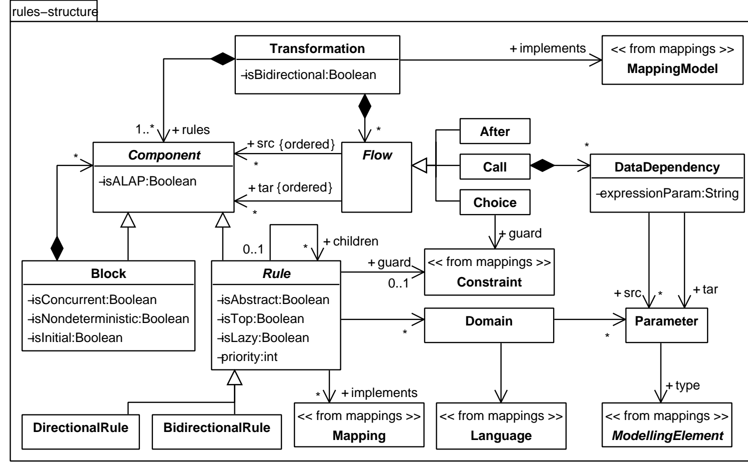

Fig. 6. Excerpt of the meta-model of the rule structure diagram.

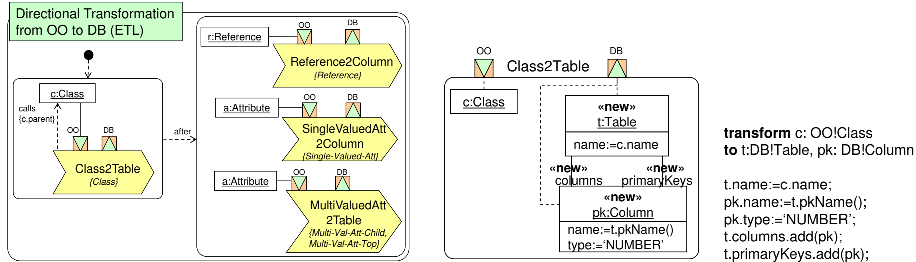

Fig. 7. Rule structure diagram for ETL (left). A behavioural rule diagram in visual (center) and textual (right) notation.

or layers, in ETL rules are executed once at each instance of the input parameter type. Hence, even though the language covers the most widely used styles of transformation, for its use with particular platforms we define *platform models* for different transformation languages. These models contain the features allowed in each languages, and can be used to check whether a rule model is compliant with an execution platform when code is generated, as well as by editors to guide the user in building compliant models with the platform. Nonetheless, we believe that a general design language will enable platform interoperability.

From the point of view of the rule structure diagrams, rules are black-boxes: their behaviour is still missing, in particular, attribute computations and object and link creations are not specified. We use *rule behaviour diagrams* to specify the actions each rule performs. We have identified three ways of expressing behaviour: (i) action languages, (ii) declarative, graphical pre- and post-conditions, and (iii) object diagrams annotated with operations like *new*, *delete* or *forbidden*.

In the case of an action language, one can use the concrete syntax of existing transformation implementation languages such as ATL or ETL. The case of pre- and post-conditions follows the style of graph transformation [15]. The third option is present in Fujaba (`http://fujaba.de`). The center of Fig. 7 shows a behavioural diagram using this third type of syntax where created elements are annotated with the *new* keyword. The right of the figure shows the same rule using an action language with ETL syntax.

3.6 Implementation and testing

*trans*ML does not include any implementation language, but we use existing target languages to implement the transformations (e.g. QVT, ATL or ETL). Using the MDE philosophy, code for different platforms can be generated from the diagrams, specifically from the rule (structure and behaviour) diagrams.

With respect to testing, test cases can be generated from the *transformation cases*, and assertions can be injected in the transformation code from the formal specification built in the analysis phase. This injected code is an *oracle function*, independent of the transformation implementation code. As an example, the following listing shows part of the EOL code automatically generated from pattern `N(NoRefinedAttrs)` in Fig. 3, which can be injected into the *pre* section of the transformation code to discard non supported input models:

```
operation sat_NoRefinedAttrs () : Boolean {
  return not OO!Class.allInstances().exists(p |
               OO!Class.allInstances().exists(c | c <> p and
                 OO!Attribute.allInstances().exists(a |
                   p.features.includes(a) and
                   OO!Attribute.allInstances().exists(ar |
                     ar <> a and c.features.includes(ar) and
                      checkatt_NoRefinedAttrs(p, c, a, ar)))));
}
operation checkatt_NoRefinedAttrs
  (p:OO!Class,c:OO!Class,a:OO!Attribute, ar:OO!Attribute) : Boolean {
  var X:=a.name;    var Xar:=ar.name;
  return c.general.includes(p) and X=Xar;
}
```

3.7 Traceability

Even though the different *trans*ML diagrams can be used in isolation, their power comes from their combined use. This is so, as one can trace requirements into the code and build the final transformation by the progressive refinement of models. In this way, we have defined traceability relations between the different diagrams as shown to the left of Fig. 8. These relations correspond to the dotted arrows in Fig. 1. Thus, it is possible to trace which requirements are considered by a given scenario, specification property, architectural component or mapping. We can

also trace the mappings and components a rule implements, and the behavioural diagram that refines a rule. Therefore, we can trace which requirements each rule addresses and vice-versa.

4 Tool Support

We have developed Ecore meta-models for the presented languages, together with several model transformations and code generators that allow automating the conversion between diagrams, as shown to the right of Fig. 8. The purpose of these transformations is to provide partial automation for model refinement from requirements to code generation. For example, given a mapping diagram we can generate a skeleton of a rule diagram, which has to be completed with the behaviour model by the transformation developer. All model transformations have been implemented with ETL, and all code generators with EGL.

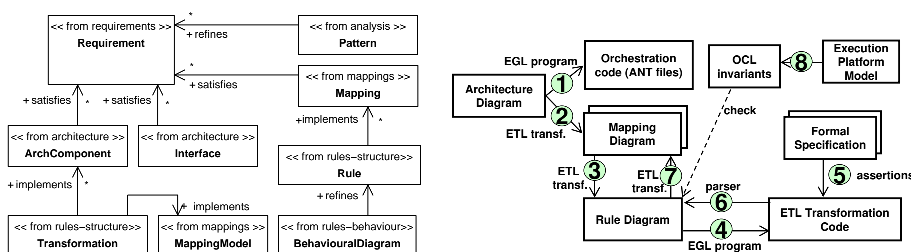

Fig. 8. Traceability links (left). Tool support (right).

The code generator with label "1" takes as input the architecture diagram, and generates ANT files that orchestrate the execution of the transformation chain specified in the architecture (i.e. it will ask the user the models to transform and pass them to the appropriate transformations). This generator also produces one additional ANT file for each transformation in the architecture, which defines tasks to automate the other labelled activities in the figure.

Transformation "2" generates one mapping diagram for each transformation in the architecture. These mapping diagrams are added a mapping for each concrete class defined by the input ports types. Transformation "3" generates a simple rule diagram from a mapping diagram that contains one rule for each mapping. Each rule stores a trace pointing to the mapping it implements. The opposite transformation is also possible for reengineering (label "7").

As stated before, one may use features of rule diagrams that are not available in the specific platform. In order to check whether a certain set of rule diagrams fits a particular execution platform, we have created an OCL code generator (label "8") that, given a platform model (ETL in our case), synthesizes OCL constraints. These constraints are checked on the rule diagrams, discovering whether they conform to the features of the platform.

In "4", ETL code is generated from the structural rule diagram, taking into account the flow directives. A parser for reverse engineering (label "6") generates the diagram from ETL code. Finally, the generator in "5" produces OCL code from the properties defined with the specification language. There are two ways to inject this code into ETL transformations. Firstly, code generated from patterns specifying restrictions on the input model is included in the *pre* section of the transformation, and checked on the input model before the transformation starts. If the model violates these constraints, a pop-up window informs the user of the unsatisfied properties. Secondly, code generated from patterns specifying properties of the transformation or of the expected output models is injected in the *post* section of the transformation, and checked when the transformation ends. This is used to perform run-time verification of the transformation. The user is informed of any violated property and of the rules that are responsible for the error. An example screenshot is shown in Fig. 10.

5 Case Study

In the INESS European project (`http://www.iness.eu`), experts have been modelling railway signalling systems using xUML (Executable UML). Our task in this project includes the formal verification of these models. Amongst other verification efforts, we used *trans*ML to define a transformation from xUML to PROMELA, the language of the SPIN model-checker (`http://spinroot.com`).

Due to the research nature of the project, we were not given initial requirements about the transformation, but they emerged as the transformation was better understood. Examples of requirements for the input models include: (i) classes always have an associated state machine; (ii) multiple-inheritance is not allowed; (iii) a special class is used to instantiate a scenario (representing a railway track layout) for the execution (analysis) of the model; (iv) objects can only be created in the state-machine of the "application" class. Fig. 9 shows to the left a specification pattern expressing the restriction (ii) of no multiple inheritance.

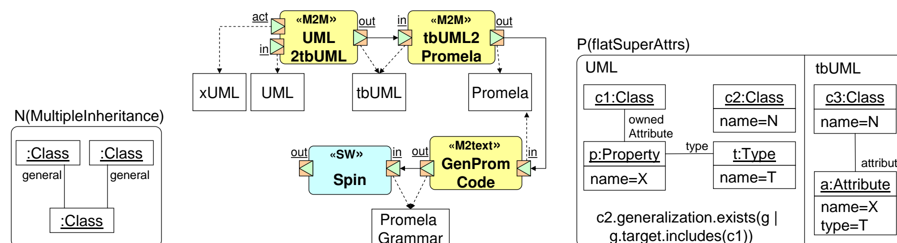

Fig. 9. A restriction on the input model (left). Architecture of the project (center). A verification property for transformation *UML2tbUML* (right).

This transformation poses many challenges, mostly concerned with handling the action language in its full generality. For this purpose, we split the transformation in several steps that could be handled more efficiently. The architecture of the final system is shown in the center of Fig. 9. It makes use of an intermediate meta-model, called (transition-based) tbUML, which is a simplified UML meta-model that only considers the structure of class diagrams and the possible set of transitions of the state-machines. Thus, the first transformation performs a flattening of the classes and states machines. This transformation makes use of the xUML meta-model for handling the action language. Then, the tbUML model is transformed into a PROMELA model, from which code conforming to the PROMELA grammar is generated as input to SPIN.

Splitting the transformation facilitates the elicitation of requirements. For instance, requirements related to the flattening of classes in the first transformation include copying attributes, associations and states for each class and its generalizations (a pattern specifying the requirement on attributes is to the right of Fig. 9). Requirements related to the flattening of state machines include aggregating and creating transitions depending on concurrent events of orthogonal states and of state machines associated to super-classes, as well as on exit actions in composite states.

Fig. 10. Testing the implementation.

We used the mapping diagrams of *trans*ML to understand and reason about corresponding elements in the two M2M transformations, and to generate skeleton rule structure diagrams from them. We also generated assertion code for the run-time verification of the transformations. Fig. 10 shows a moment in the execution of the first transformation (more than 1600 LOC), where a violation of the verification property `flatSuperAttrs` occurs. By having traceability from the models into the code, we were able to identify the erroneous rule.

6 Conclusions and Lines of Future Work

Transformations should be engineered, not hacked. For this purpose we have presented *trans*ML, a family of languages to help building transformations using well-founded engineering principles. The languages cover the life-cycle of the transformation development including requirements, analysis, architecture, design and testing. We have provided partial tool support and automation for the MDE of transformations, and evaluated the approach using an industrial project, which showed the benefits of *modelling* transformations.

We are currently working in improving the tool support for our approach, in particular the usability of the visual editors and the integration of the different

languages. We are also planning the use of *trans*ML in further case studies, and investigating *processes* for transformation development.

Acknowledgements. Work funded by the Spanish Ministry of Science (project TIN2008-02081 and grants JC2009-00015, PR2009-0019), the R&D programme of the Madrid Region (project S2009/TIC-1650), and the European Commission's 7th Framework programme (grants #218575 (INESS), #248864 (MADES)).

References

1. ATL. `http://www.sciences.univ-nantes.fr/lina/atl/`.
2. J. Bézivin, F. Jouault, and J. Paliès. Towards model transformation design patterns. In *EWMT'05*, 2005.
3. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
4. A. Etien, C. Dumoulin, and E. Renaux. Towards a unified notation to represent model transformation. Technical Report RR-6187, INRIA, 2007.
5. J.-M. Favre and T. Nguyen. Towards a megamodel to model software evolution through transformations. *Electr. Notes Theor. Comput. Sci.*, 127(3):59–74, 2005.
6. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
7. E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A visual specification language for model-to-model transformations. In *VLHCC'10*. IEEE CS, 2010.
8. M. Iacob, M. Steen, and L. Heerink. Reusable model transformation patterns. In *3M4EC'08*, pages 1–10, 2008.
9. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
10. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
11. A. Kusel. TROPIC - a framework for building reusable transformation components. In *Doctoral Symposium at MODELS*, 2009.
12. QVT. `http://www.omg.org/docs/ptc/05-11-01.pdf`.
13. L. A. Rahim and S. B. R. S. Mansoor. Proposed design notation for model transformation. In *ASWEC'08*, pages 589–598. IEEE CS, 2008.
14. J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. Orchestrating ATL model transformations. In *MtATL 2009*, pages 34–46, 2009.
15. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
16. M. Siikarla, M. Laitkorpi, P. Selonen, and T. Systä. Transformations have to be developed ReST assured. In *ICMT'08*, volume 5063 of *LNCS*, pages 1–15. Springer, 2008.
17. J. M. Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.
18. B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers. Uniti: A unified transformation infrastructure. In *MODELS'07*, volume 4735 of *LNCS*, pages 31–45, 2007.
19. D. Varró. Model transformation by example. In *MODELS'06*, volume 4199 of *LNCS*, pages 410–424, 2006.