Software & Systems Modeling 12.3 (2013): 555–577

DOI: http://dx.doi.org/10.1007/s10270-011-0211-2

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Engineering Model Transformations with *trans*ML

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

[1] Universidad Autónoma de Madrid (Spain), e-mail: {Esther.Guerra, Juan.deLara}@uam.es
[2] University of York (UK), e-mail: {dkolovos, paige, osantos}@cs.york.ac.uk

Received: date / Revised version: date

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 as well. 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. In this way, even though transformations should be built using sound engineering principles – just like any other kind of software – there is currently 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 enabling the *engineering* of transformations. 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 several transformation implementation languages.

Key words Model Driven Engineering – Model Transformation – Domain-Specific 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 the automation of model management operations. 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 [12]. Prominent examples of such languages are QVT [52], ATL [36], Triple Graph Grammars [57] and ETL [40].

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 nowadays 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 [1,6] 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 several transformation implementation languages. We also provide support for reengineering transformation

Send offprint requests to:
⋆ *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

code by its parsing into *trans*ML models, and facilitating platform migration.

This paper is an extended version of [29]. Most notably, here we incorporate a new diagram type for transformation testing – in a style similar to the xUnit testing frameworks [4] – together with a supporting platform; we illustrate the use of platform models to characterize different implementation languages and validate rule diagrams for them; we support the generation of QVT code from *trans*ML models; and we elaborate on two extended case studies.

The paper is organized as follows. Section 2 reviews previous attempts to model transformations, pointing out limitations and motivating the needs for high-level modelling languages for transformations. Next, Section 3 gives an overview of *trans*ML, our family of languages that cover the identified needs to build transformations in the large. The following sections describe each language of the family: Section 4 describes our support for the requirements and analysis phases; Section 5 shows our notation to model the architecture; Section 6 introduces the languages to model the high-level (mappings) and low-level (rule structure and behaviour) design; and Section 7 explains our support for testing. Then, Section 8 shows how all these languages are integrated by means of traceability relations. Section 9 presents tool support for forward and reverse transformation engineering, followed by Section 10, which evaluates the approach with two case studies. Finally, Section 11 concludes.

2 Related work

Most research in M2M transformation focuses on the implementation phase, either to develop new languages to implement transformations, 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. In the following we review languages and approaches directed to modelling transformation development phases other than implementation.

Requirements and Analysis. Very few attempts to describe methods for capturing and representing model transformation requirements can be found in the literature. In [26], the authors propose applying test-driven development [4] to model transformations. Thus, requirements are captured in the form of test cases made of an input model, together with output fragments and assertions expressed in OCL. This representation is however not suitable to capture non-functional requirements, or to express relationships between functional ones. Other recent works only target a specific non-functional requirement type, such as [34, 48], where the expected quality attributes for transformations are modelled and subsequently used to discriminate between alternative transformation design solutions.

Design. As for analysis, there is limited work proposing design notations for transformations. For instance, [53] presents a language to design transformations, but focused only on their implementation. There are several approaches that use UML object diagrams to represent each rule pre and post-conditions [11, 20] as well as notations similar to activity diagrams to represent rule control flow. As an example, UML-like class diagrams are used to represent the structure of rules and cover the low-level design of transformations in [18]. In [27] the authors propose a UML profile for modelling transformations. In particular, they use a stereotyped activity diagram where each activity is tagged with the name of a rule, and rule behaviour is expressed by class diagrams with stereotypes (like *create* and *destroy*). The aim of that work is to enable the generation of code for different transformation engines.

There is also work on architectural design languages allowing for the composition and orchestration of transformations. Whereas [55] is a specific language for composing ATL transformations, the approach in [64] is more platform independent. Kleppe proposed the MCC environment [37], which offers a scripting language with composition operators enabling the design of transformation chains. In [66] the authors propose mechanisms to compose transformation chains by defining correspondence meta-models. In [2] the authors present a tool integration framework enabling the description and execution of MDE processes. In all cases, other phases of transformation development are neglected.

Validation and Verification. Validation and Verification is an integral task of software development. In the context of model transformations, there are many works targeting the verification of transformation implementations. We distinguish two approaches: those based on the use of a formal transformation language, like graph transformation [16, 17], and those translating the transformation specifications into a formal domain enabling analysis, such as Petri nets [14], rewriting logic [8] or a SAT problem [10].

There are also works dealing with model transformation testing [3, 9, 21, 43]. In [46], the authors present a testing framework for the C-SAW transformation languages atop the GME meta-modelling environment. A test case in this approach consists of a source model and its expected output model. An execution engine runs the tests and displays the differences between the actual and expected output. The approaches in [26, 41] follow a similar philosophy to the xUnit framework but for transformation testing.

Another branch of related works deals with the generation of input test models for transformation test-

ing [21,58]. These works tend to consider only the features of the input meta-models in order to generate the test models, but not properties of the transformations (i.e. they support black-box testing). There are a few exceptions though. For example, in [43], the authors propose using all possible overlapping models of each pair of rules in a transformation as input models for testing as well (i.e. white-box testing).

Other works target the testing of transformation engines [13,61]. For instance, in [13], the authors generate test models in order to test the pattern matching algorithm of graph transformation engines. The generated test models consider the structure of the rule pre-conditions, the specific semantics of graph transformation (e.g. dangling edges, non-injective matches) and fault injection techniques.

Finally, some approaches bring techniques from compiler testing for the purpose of testing model-based code generators [56,62]. For example, MetaTest [56] is a tool directed to testing model-based processors, like code generators or model simulators. It allows the specification of the model syntax using context free grammars, and their semantics using inference rules. Then, the tool uses the inference rules to generate models satisfying certain syntactic and semantic coverability criteria. In [62], the authors test optimization rules in model-based C code generation from Simulink models. The semantics of the optimization rule under test is given by a graph transformation rule, and its (possibly infinite) input space is partitioned into a finite number of equivalence classes. Then, test cases are automatically generated to compare the execution of the Simulink model and the generated C code.

Complete Life-Cycle. Only a few works cover several phases of transformation development, being closer to our engineering view of developing transformations. For instance, [59] identifies a transformation development life-cycle and proposes describing transformations incrementally, starting from transformational patterns and partial specifications that are gradually refined. Unfortunately, no concrete notation or tool is proposed. The position paper [42] 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 seems ad-hoc for their particular transformation approach and cannot be applied to other implementation languages.

In the context of the QVT RFP, the authors in [5] propose using a tool/technology- independent transformation representation, which could be used to realize the transformation using several implementation languages. The suggested platform-independent representation is UML, modelling transformations as operations in a class diagram, and resorting to activity diagrams to express the transformation behaviour. Even though this work recognises that transformations should be developed similarly to other software, it obviates the fact that activities like analysis and testing are required as well, and that transformation development requires from specific abstractions and primitives not present in UML. Thus, from the point of view of transformation designers, it is more productive to have notations with native support for transformation concepts like mapping, rule or transformation.

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 transformation 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 for model transformations [1,6,33].

3 Overview of *trans*ML: 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 [5]. 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 their relations. The upper part of the figure shows the languages in the family: a requirements diagram, formal specification diagrams and scenarios to cover the transformation analysis; an architecture diagram to break the transformation in modular units; a 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 enable tracing 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 to the bottom of the figure: the transformation code (in any implementation language such as QVT or ETL), the generation of test cases and testing models (also supported by *trans*ML) to exercise the transformation using different criteria, the run-time verification of transformation code, and the orchestration of transformations.

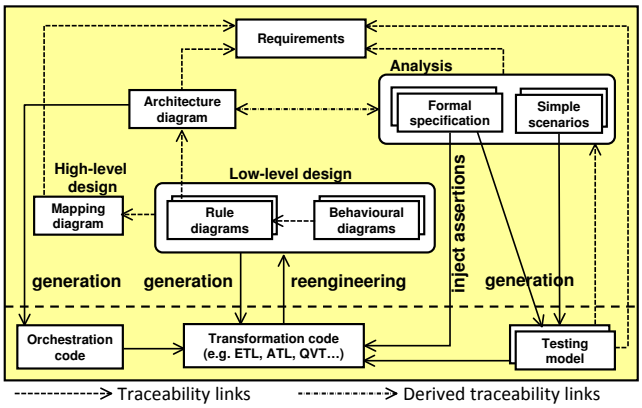

Fig. 1 Model transformation framework.

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. Our testing models allow test-driven development of transformations as well [26]. We do not suggest either that *all* diagrams have to be used when building a transformation, just like when building object-oriented systems it is not mandatory 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. Nonetheless, the full power of *trans*ML comes by using its diagrams in combination.

The following sections present *trans*ML in detail. We will use as a running example the class-to-relational transformation to ease understanding, and provide evaluation of its use with two more complex transformations – one of them in an industrial project – in Section 10.

4 Requirements and analysis

4.1 Requirements elicitation

Just like in the development of any other kind of 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 can use any technique and notation from the Requirements Engineering community [63]. However, in order to trace requirements into subsequent phases, *trans*ML includes a representation of requirements in the form of diagrams. In particular, we use a notation similar to SysML requirements diagrams [22], where we have left out elements not deemed necessary for gathering transformation requirements, and added other concepts specific to transformations (e.g. a classification of the source of requirements).

The meta-model for this representation, shown in Fig. 2, enables hierarchical decomposition, classification, refinement and traceability of requirements. Requirements contain an index indicating their relative position in the hierarchical decomposition, and are classified in a dual way: attending to whether they are functional or not, and to whether they are requirements of the input models, the output models or the transformation itself.

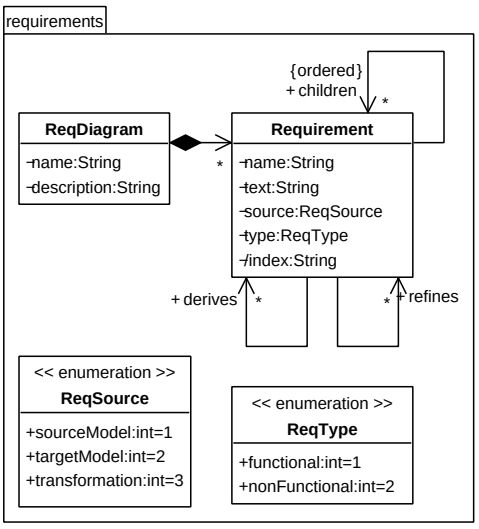

Fig. 2 Requirements meta-model.

Since transformations are sometimes not meant to work properly with all possible instances of the input meta-models, but with a restriction of them, it is important to explicitly record the requirements needed by input models to qualify for the transformation. Similarly, the transformation may not be able to generate each possible instance of the output meta-models (i.e. it may be not surjective), and this knowledge should be recorded too. In our diagram it is possible to differentiate between these two requirement types. We will see in next subsection that our formal specification language is able to precisely describe some of these requirements.

As an example, Fig. 3 shows the requirements diagram for the class-to-relational transformation. Requirements for the input model are annotated with a rightwards arrow in the upper right corner, whereas requirements of the transformation are annotated with dented wheels. The children of a requirement are shown below it,

connected with lines terminated in a divided circle, and indexed concatenating consecutive numbers to the index of the parent. In the figure, requirement `0.1` restricts input models to have no redefined attributes, whereas requirement `0.3.1` derives from requirements `0.3.2` and `0.3.3`.

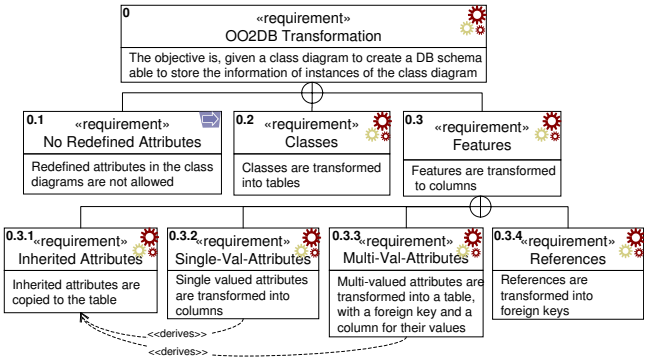

Fig. 3 Requirements for the example transformation.

4.2 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 that provide 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, Fig. 4 shows a transformation case explaining that a multi-valued attribute should be translated into a table with a foreign key from the table associated to its owner class. Actually, this transformation case is given through model fragments as, in a correct OO model, classes need to be enclosed in packages, whereas in correct DB models tables should belong to a schema. Although in this case we have used the abstract syntax for both models, we could use a concrete syntax as well.

The use of *transformation cases* serves different purposes. 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 [65] which derive a rough sketch of the transformation. Third, they can drive the transformation implementation using test-driven development approaches [26], and they can also be used as test cases to validate the transformation implementation (see Section 7).

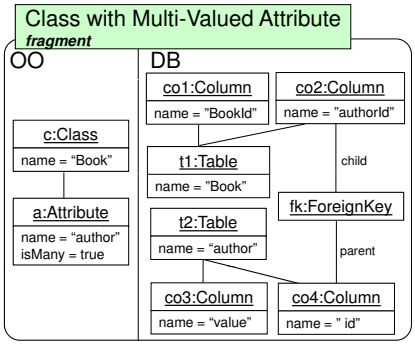

Fig. 4 Transformation case with model fragments.

The second notation we use in this phase is a visual, formal *specification* language [28]. Similar to the role of Z [60] or Alloy [35] 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 through declarative patterns that express *allowed* or *forbidden* relations between two models. Its meta-model is shown in Fig. 5, and its formalization is available at [28,30]. Patterns have a graphical part (class *ConstraintTripleGraph* in the meta-model), and can include conditions on the attribute values and constraints (we use EOL [38] 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 permit specifying properties for both uni-directional and bi-directional transformations, as they can be interpreted source-to-target and target-to-source.

Since *trans*ML is designed to be independent of the language used to implement the transformation, our specification language supports the two most common styles for M2M transformation: trace-based and traceless, depending on whether explicit traces are given between source and target elements or not. Examples of languages that use an explicit declaration of traces are QVT-Core and Triple Graph Grammars, whereas examples of traceless languages are QVT-Relations, ATL and ETL. In the case of a trace-based specification, patterns can define positive and negative graph pre-conditions (associations *positivePrecondition* and *negativePreconditions* in the meta-model of Fig. 5), and the graphical part of the patterns is made of two *Graph*s related through a *CorrespondenceGraph* which stores the traces.

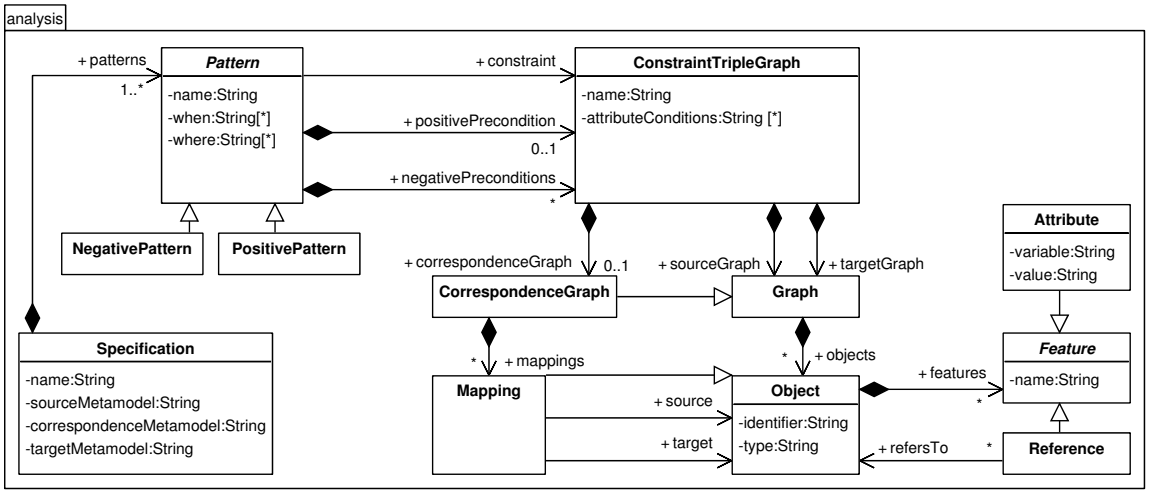

Fig. 5 Meta-model of the specification language.

In the case of a traceless specification, there is no correspondence graph, but patterns are similar to QVT relations [52] and can include graph pre- and post-conditions (*when* and *where* clauses respectively).

Our patterns have a formal semantics which allows answering correctness questions about specifications, e.g. whether there are conflicts between patterns. In addition, we provide a compilation of patterns into OCL expressions for the purpose of testing if a pair of models (usually an input model and the result of its transformation) satisfies the pattern or not. The details of this compilation are given in [28].

Finally, we maintain traceability between our patterns, the transformation cases and the requirements of the requirement diagram. A pattern addressing some requirement in the requirements diagram is said to *refine* it. As patterns can be used also for testing, this traceability enables the detection of non-satisfied requirements.

As an example, the left of Fig. 6 shows a negative pattern (indicated by the `N(...)`) used to express a restriction on the input models. The pattern *refines* requirement `0.1` in Fig. 3 (no redefined attributes). It 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`). In our language, attributes may have a specific value, or may be assigned to a variable, which then can be included in a constraint. As the pattern is negative, models in which the pattern occurs are invalid. In this respect, we can inject in the transformation code the OCL expressions generated from the pattern in order to test whether a given input model qualifies for the transformation.

The right of the figure shows a positive pattern (indicated by the `P(...)`) expressing a property of the transformation. The pattern refines requirement `0.3.1` (inherited attributes). It 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

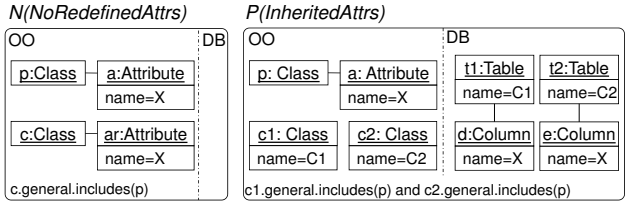

Fig. 6 Restriction on the input model (left). Verification property (right).

the classes are transformed are located by equality of names (variables `C1` and `C2`), but any formula relating their names is also allowed. We can use patterns like this one for several purposes. First, we can inject the OCL code generated from the pattern in the transformation, in order to check whether the implementation generates target models satisfying these properties. Second, we can use the patterns as assertions in testing models, so that combined with a suitable set of input models enable transformation testing in a similar style to the xUnit framework. Finally, we can use these patterns to reason about the correctness of the requirements themselves.

Although there is considerable research in languages to express patterns on graphs [31,51,54], our language has the characteristic of being specifically designed for M2M transformations. Hence, our patterns contain both a source and a target model (graph pattern languages only consider one graph) and their bidirectional semantics enables their interpretation forwards and backwards.

5 Architecture

Large software is seldom monolithic, but is decomposed into interacting blocks. Hence engineers have to design its architecture. Moreover, it is often the case that a transformation has to be integrated with further soft-

ware components providing extra functionality, such as code generators. For these reasons we have included a modelling language for architectural design which permits the modular decomposition of transformations in functional units, as well as their integration with other software artifacts. This is very useful in large-scale transformations that need to be split in different units and orchestrated. For the design of this language we have taken some ideas from works dealing with orchestration of transformations [55,64], as well as from architectural description languages [25].

Our architectural language is made of *components* and connectors, as the meta-model in Fig. 7 shows. Components interact through directional interfaces with a type given by meta-models, event types, artefacts 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). As we will see later, *trans*ML foresees different kinds of constraints: interpreted constraints in some language like OCL, opaque constraints written e.g. in natural language, and constraints given by a pattern specification like the ones used during the analysis phase (previous section). Constraints act like a contract [50] to further restrict the expected inputs and outputs of a component. Moreover, assigning constraints to ports enables conformance checking when connecting two components.

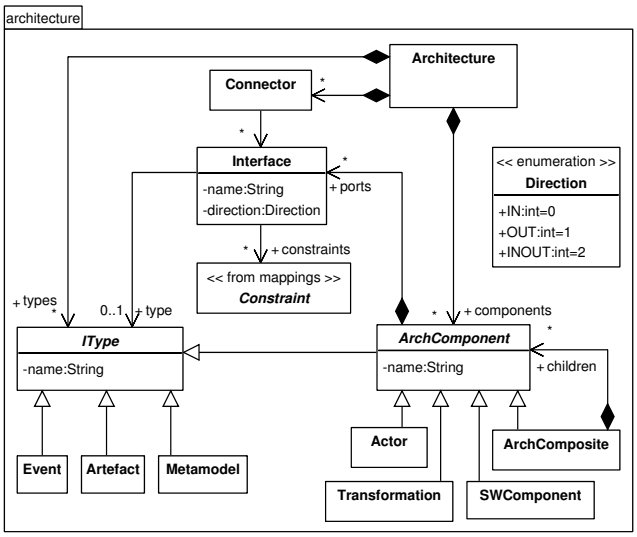

Fig. 7 Architecture meta-model.

Fig. 8 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 a meta-model, together with extra constraints to rule out models that conform to the meta-model but are not handled by the transformation. Models conforming to those interfaces can be input/output of the transformations. For example, the input interface of component `OO2DB` has the UML meta-model as type, and is constrained by the pattern `N(NoRedefinedAttrs)` shown in Fig. 6. The type also allows checking compatibility when connecting two transformation components.

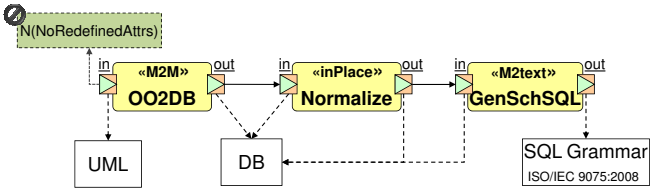

Fig. 8 Architectural diagram: transformation-centric view.

Fig. 9 shows a type-centric view of the same model. This view is similar to a mega-model [19], 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.

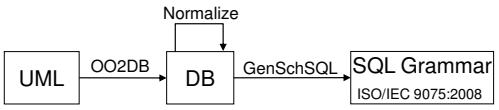

Fig. 9 Architectural diagram: type-centric view.

6 Design

6.1 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. We use a concrete syntax similar to Triple Graph Grammar schemas [57], however our mappings are not intended to be used as an auxiliary tracing mechanism to guide the actual execution of the transformation code.

Fig. 10 shows the mappings meta-model. A *mapping model* is established between several languages, each one of them defined by a meta-model. The directionality of

the transformation is established using the *navigable* attribute in *ModelEnd*s. Models are structured in packages, each one of them containing mappings that can be organized hierarchically as well. Mappings connect elements in the meta-models of the involved languages through *MappingEnds*. Mappings are provided with constraints to express when a mapping holds. Constraints can be given either in uninterpreted text, in some language like OCL, or using our formal specification patterns. The mapping meta-model refers to the meta-models of the languages involved in the transformation. We use an abstract class *ModellingElement* to represent meta-model elements, which can be replaced by any concrete meta-modelling infrastructure.

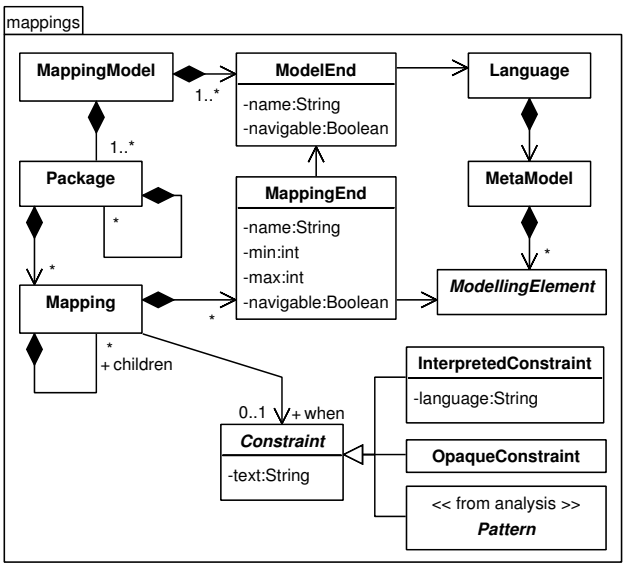

Fig. 10 Mappings meta-model.

Fig. 11 shows a mapping diagram. It has one block for each language, containing the relevant elements of their meta-model. Another block in the middle 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 not shown) and a direction (to denote either access or creation of elements). As our example 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. 11 we have omitted element *ForeignKey* of the DB meta-model. Later, if needed, one can add more details: further mapping ends, additional mappings, or new constraints to refine existing mappings. For instance, Fig. 12 shows an excerpt of a mapping diagram that refines the previous one by attaching an OCL constraint to one of the mappings. The

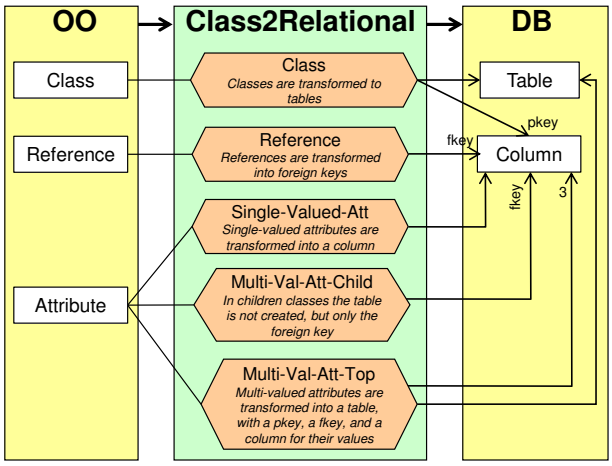

Fig. 11 Mapping diagram example.

constraint demands the owner class of the attribute *at* to be top-level, so that the mapping makes sense only in this case.

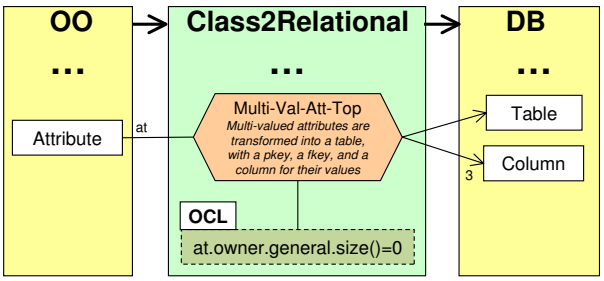

Fig. 12 Adding constraints to mappings.

The mapping diagram is a high-level design notation, and hence 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.

6.2 Low-level design: Rule structure and rule behaviour

Low-level detailed design diagrams indicate *how* the transformation has to be implemented. Here we use a rule-based style for transformations, and separate the description of the rule structure from its behaviour (other transformation styles such as functional are not currently supported). 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. In particular, rule structure diagrams are helpful to specify and understand the relationships between rules, both control flow and parameter passing. They provide high-level structuring mechanisms like

blocks, which perhaps are not present in the target implementation language, as well as a compact notation to express common dependencies. Our notation can also help in describing good practices and transformation patterns, in the same way as UML helps to record object-oriented patterns. Finally, rule diagrams are also useful to generate code for different platforms and reengineering of existing code, so that implementation code can be better understood using a more abstract notation.

Fig. 13 shows part of the meta-model of the rule structure diagrams. This kind of diagram depicts the structure of rules (input and output parameters), their execution flow, and data dependencies between them (e.g. parameter passing). 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. It is also possible to specify helpers, i.e. auxiliary operations to be used by rules.

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*. Among the flows, we differentiate between rule execution order (class *After*), alternative execution flows (*Choice*), and explicit invocations to other rules (or blocks of rules) which may imply parameter passing (classes *When* and *Call* for invocations that occur before or after the rule, respectively).

Fig. 14 shows a rule structure diagram with four rules, which considers ETL as the implementation language of the transformation. 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 any 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. As previously stated, note how blocks serve as a useful structuring mechanism, which may not be present in the target implementation language (ETL in this case).

In contrast to the previous stages in the transformation modelling, where the designer can remain oblivious of the language finally used to implement the transformation, in this stage the particular language should be taken into account. This is so as the rule structure diagram models the actual rules of the transformation, their relations and the execution flow. Since the features of the

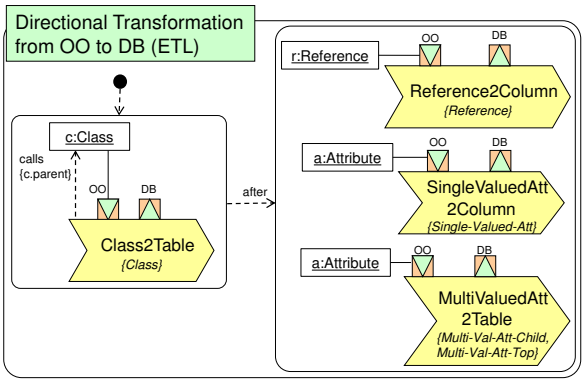

Fig. 14 Rule structure diagram for ETL.

great variety of implementation languages are heterogeneous, targeting one language or other may result in different rule diagrams. For instance, Fig. 15 shows the rule structure diagram in case the transformation is being written with QVT-Relations. In this case, *Class2Table* is a top rule (indicated with a double thicker border), there is an additional rule *ChClass-Table* to handle the transformation of children classes, and the rules in the right block are explicitly invoked receiving a class and table as parameters (since all rules in this block receive the same parameters, we use a shortcut notation and indicate the parameters in the block instead of in each one of the rules). This diagram cannot be used for ETL because ETL rules are allowed to have only one parameter in the source domain.

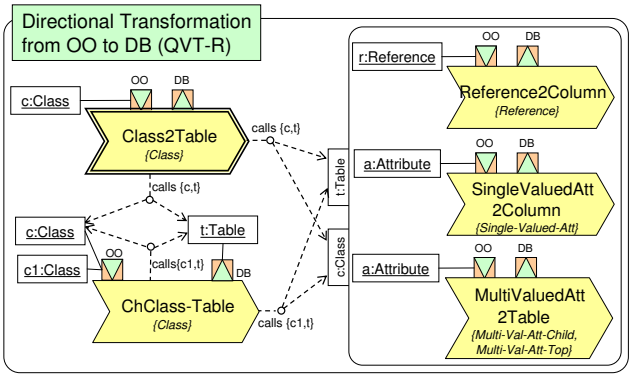

Fig. 15 Rule structure diagram for QVT-Relations.

Thus, although our rule language captures the main features of transformation languages, 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 they only have one input parameter if we use ETL, and we have object patterns if using QVT-Relations. 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 or layers, in ETL rules

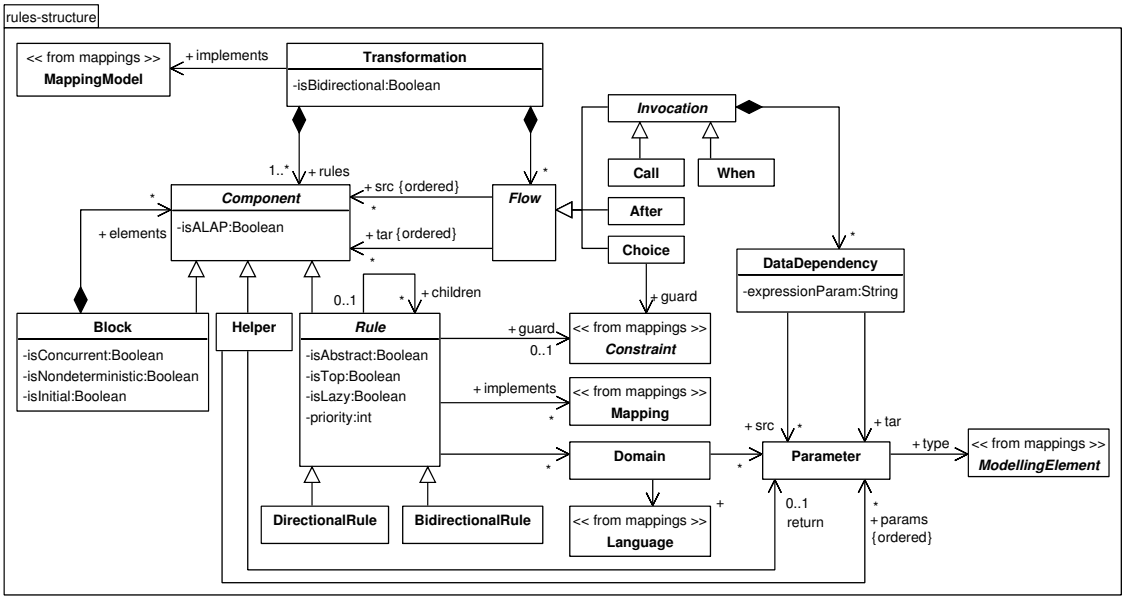

Fig. 13 Excerpt of the meta-model of the rule structure diagram.

are executed once at each instance of the input parameter type. Hence, even though our language for modelling rule structure 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 language, 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. Hence, they act as a kind of profiling mechanism for different transformation languages.

Fig. 16 shows the meta-model of our platform models. This allows customizing the supported control flow constructions, rule features, number and type of rule parameters, and execution policy of transformation languages. Boolean features, such rule extension and abstraction, can be either present or absent in a certain language. Features with type *SupportType* can be either the only possible choice in the language (value *all*), it can be an optional feature (*selectable*) or *unsupported*. For instance, in QVT-Relations rules are always executed as long as possible (*all*), we can choose whether they are top or not (*selectable*), and rule priorities are not supported.

As an example, Table 1 depicts in the form of a table the platform models for the languages ETL and QVT-Relations. Platform models allow comparison of transformation languages. While QVT-Relations supports bidirectionality and rules with arbitrary parameters, ETL is directional and rules support one parameter in the *from* domain. Hence, platform models allow comparing different transformation approaches, in a similar way as in [12].

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

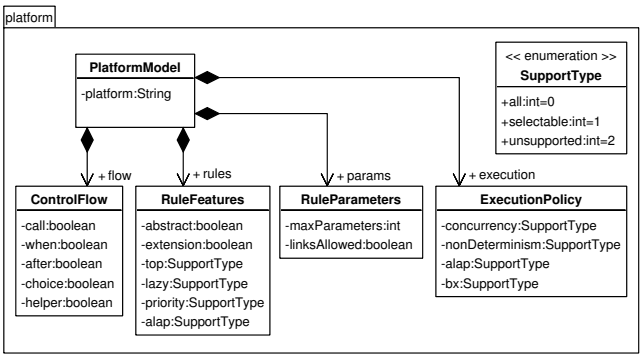

Fig. 16 Platform meta-model.

Table 1 Platform models for ETL and QVT-Relations.

		ETL	QVT-R
Control flow	call	*true*	*true*
	when	*false*	*true*
	after	*true*	*false*
	choice	*true*	*true*
	helper	*true*	*true*
Rule features	abstract	*true*	*false*
	extension	*true*	*false*
	top	*unsup.*	*selec.*
	lazy	*selec.*	*unsup.*
	priority	*unsup.*	*unsup.*
	as long as possible	*all*	*all*
Rule parameters	max parameters	1	$*$
	links allowed	*false*	*false*
Execution policy	concurrency	*unsup.*	*unsup.*
	non determinism	*unsup.*	*all*
	as long as possible	*all*	*all*
	bidirectionality	*unsup.*	*selec.*

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 [57]. The third option is present in tools like Fujaba [23]. As an example, the left of Fig. 17 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.

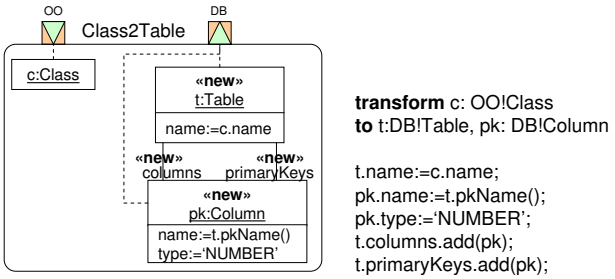

Fig. 17 Behavioural rule diagram in visual (left) and textual (right) notation.

7 Implementation and testing

*trans*ML does not include any implementation language, but we can use any existing target language 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. Currently, we support both ETL and QVT-Relations, but many other languages, like ATL, could be targeted as well.

With respect to testing, *trans*ML includes a dedicated language for model-based testing, of which its meta-model is shown in Fig. 18. This language enables the description of test cases, including both the test input models (class *TestInput*) and the expected outcome or oracle function (class *Assertion*). The language supports four different formats for the input data: models (which can be either a file or a *trans*ML graph model), meta-models, graph constraints and constructive operational specifications (for instance written in EOL). In the case of meta-models and graph constraints, it is possible to specify cardinality for the number of instance models to be generated as input, as well as a set of generation criteria guiding the test generation procedure and defining coverage criteria [21] (e.g. class, association and attribute coverage among others). The expected output for a specific test case can be given as an operation over a given model or as an *oracle function*. The first possibility is useful to check whether the transformation result is *equals*, *includes* or *overlaps* with a given model. The second approach allows checking verification properties of the resulting model, which can be defined either as a specification (e.g. written in OCL) or as a graph constraint using a pattern of our formal specification language.

Test cases can be automatically generated from the *transformation cases* as these already define the expected output for a given input, as well as from the formal specification built in the analysis phase, in this case by deriving *AssertionGraphConstraint* assertions from each one of the verification properties. Thus, having an explicit model for the test suite makes it possible to trace back detected errors to the specific transformation cases and properties that were violated during a particular transformation execution.

As an example, Listing 1 shows a testing model for the class-to-relational transformation, using a textual concrete syntax. It contains a test case derived from the transformation case in Fig. 4, which exemplified the transformation of multi-valued attributes. Its input is the OO model in the transformation case where, in addition, the operation 'complete' in line 9 indicates that the model fragment is being added the necessary elements to obtain a valid instance of the OO meta-model. This is necessary as, in general, transformations assume correct input models. The assertion in lines 11-19 checks whether transforming the OO model yields a model that includes the DB model in the transformation case. Note how, in lines 3-4, the test case is annotated with the name of the transformation case from which it was derived.

```
1  TestSuite "OO2DB.etl" <OO:"OO", DB:"DB"> {
2
3    @transformationcase(name=
4     "Class with Multi-Valued Attribute")
5    test Class_with_Multi-Valued_Attribute {
6      input model OO {
7        c:Class { name="Book"; features=@a; }
8        a:Attribute { name="author"; isMany="true"; } }
9        .complete()
10
11     assert model DB {
12       t1:Table { name="Book"; columns=[@co1,@co2]; }
13       t2:Table { name="author"; columns=[@co3,@co4]; }
14       co1:Column { name="BookId"; }
15       co2:Column { name="authorId"; }
16       co3:Column { name="value"; }
17       co4:Column { name="id"; }
18       fk:ForeignKey { child=@co2; parent=@co4; } }
19       .included()
20   }
21 }
```

Listing 1 Test case derived from a transformation case.

Listing 2 contains another set of test cases for our transformation. In this case the input models are files specified at the test suite level (lines 3 and 4), therefore they are being used as input by all the test cases in the suite. Lines 8-23 correspond to a test case derived from the specification property shown to the right of Fig. 6.

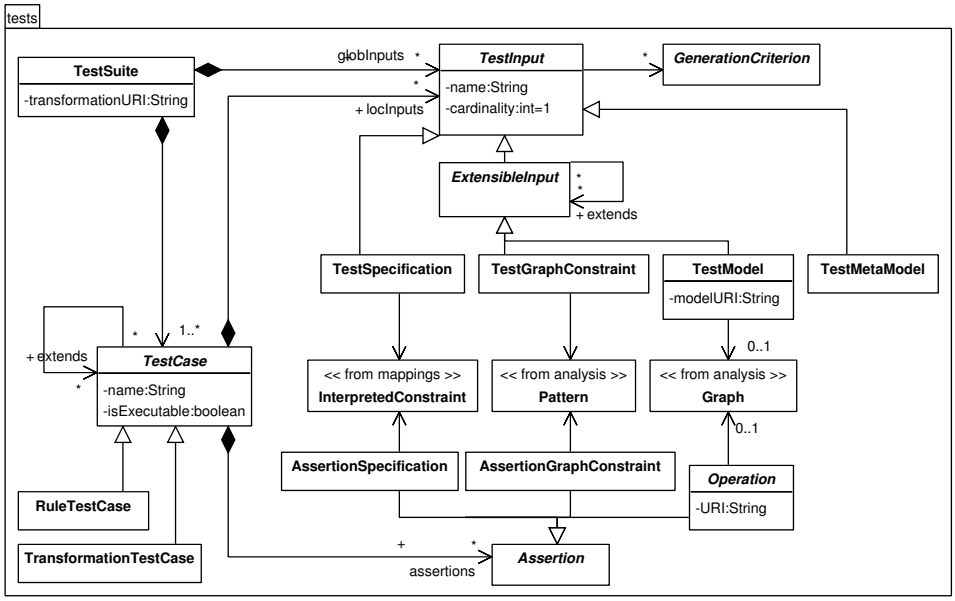

Fig. 18 Excerpt of the meta-model of the tests diagram.

This is an assertion of type *AssertionGraphConstraint*, and it uses a textual concrete syntax for the property. Another test case is shown in lines 25-34, which illustrates the use of a constraint language such as EOL to specify model assertions. The first assertion checks the absence of duplicated columns in tables, while the second checks that as many tables are generated as classes, or more (in case of transforming multi-valued attributes).

```
1  TestSuite "OO2DB.etl" <OO:"OO", DB:"DB"> {
2
3    input model "models/OOInstance1.xmi"
4    input model "models/OOInstance2.xmi"
5
6    @property(name="InheritedAttrs")
7    @requirement(name="0.3.1")
8    test InheritedAttrsProperty {
9      assert ppattern InheritedAttrs {
10       OO {
11         p:Class  { features=@a; }
12         c1:Class { name=C1; }
13         c2:Class { name=C2; }
14         a:Attribute { name=X; } }
15       DB {
16         t1:Table { name=C1; columns=@d; }
17         t2:Table { name=C2; columns=@e; }
18         d:Column { name=X; }
19         e:Column { name=X; } }
20       condition: "c1.general.includes(p)",
21                  "c2.general.includes(p)";
22     }
23   }
24
25   test transformationOfClasses {
26     assert constraint nonDuplicatedColumns (
27           "DB!Table.allInstances().forAll(t |
28             t.columns.forAll(co1 |
29               not t.columns.exists(co2 |
30                 co1.name=co2.name and co1<>co2)) )")
31     assert constraint numberOfTables (
32           "DB!Table.allInstances().size() >=
33            OO!Class.allInstances().size()")
34   }
35 }
```

Listing 2 Test case derived from a verification property and test case defining constraint-based assertions.

The automatically generated test cases perform black-box testing of transformations, where the granularity level is the complete transformation as the expected output is evaluated only after the transformation is completely executed. Nonetheless it is also possible to check assertions after rule executions through class *RuleTestCase*, allowing for a finer control of errors. Regarding white-box testing, here one is interested in specifying input models enabling the execution (or not) of sequences of rules. Some degree of automation could be achieved for this purpose by taking into account the defined rule diagrams. The more detailed and complete the rule models, the more accurate the generated tests.

Complementing the test cases, it is also possible to generate assertion code from the formal specification of transformation properties, and directly inject it in the final transformation code for its run-time verification. This injected code is an *oracle function*, independent from the actual language used to implement the transformation. As an example, Listing 3 shows part of the EOL code automatically generated from pattern `N(NoRedefinedAttrs)` in Fig. 6, which can be injected into the *pre* section of the ETL transformation code to discard non-supported input models.

```
1  operation sat_NoRefinedAttrs () : Boolean {
2    return not
3      OO!Class.allInstances().exists(p |
4        OO!Class.allInstances().exists(c | c <> p and
5          OO!Attribute.allInstances().exists(a |
6            p.features.includes(a) and
7            OO!Attribute.allInstances().exists(ar |
8              ar <> a and c.features.includes(ar) and
9              checkatt_NoRefinedAttrs(p, c, a, ar)))));
10 }
11
12 operation checkatt_NoRefinedAttrs(p:OO!Class,
13   c:OO!Class, a:OO!Attribute, ar:OO!Attribute)
14   : Boolean {
15   var X := a.name;
16   var Xar := ar.name;
```

```
17   return c.general.includes(p) and X=Xar;
18 }
```

Listing 3 EOL code derived from a specification property.

8 Putting everything together: 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. For this purpose, we have defined traceability relations between the different diagrams as shown in Fig. 19. These relations correspond to the dotted arrows in Fig. 1.

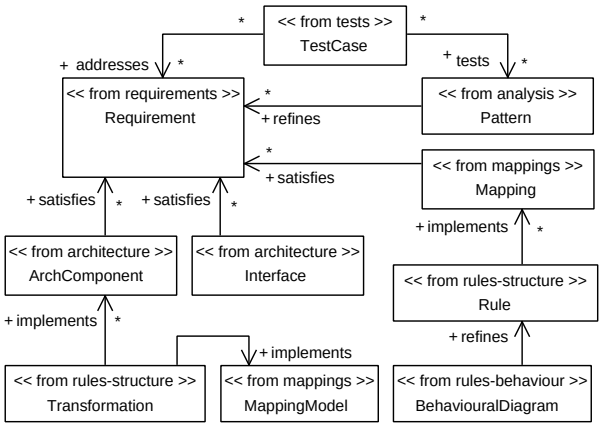

Fig. 19 Traceability links.

In particular, it is possible to trace which requirements are addressed by a given transformation case or specification pattern (*refines* relationship). Architectural components, mappings and test cases are also traced to requirements. We can trace the mappings a rule implements, and the behavioural diagrams that refine a rule. Therefore, we can trace the requirements each rule addresses and vice versa. Finally, test cases can refer to the requirements they address, and to the transformation cases or verification patterns they test.

Making available explicit traces between the different *trans*ML models permits automating the generation of requirement traceability matrices [63] documenting the requirements of a transformation against its test cases or analysis properties. As an example, Fig. 23 shows the requirement traceability matrix for the test cases of our example, using a web-based user-friendly format.

Finally, we provide some automation for traceability. For instance, we create traces when generating the skeleton of a rule structure diagram from a mapping diagram (see Fig. 20) or when generating a test case from a transformation case (see Listing 1). However, most traceability links have to be specified by hand (see for example the annotations in Listing 2). A higher degree of automation (e.g. inference or maintenance of traces) is left for future work.

9 Tool support: Towards the model-driven engineering of transformations

We have built Ecore meta-models for the presented languages, together with several model transformations and code generators that allow automating the conversion between diagrams, as shown in Fig. 20. The purpose of these transformations is to provide partial automation for model refinement from requirements to code generation and testing. Hence, they enable the construction of model transformations using MDE. For example, given a mapping diagram we generate a skeleton of a rule structure diagram, which has to be completed with the rule behaviour model by the transformation developer. All model transformations have been implemented with ETL, and all code generators with EGL.

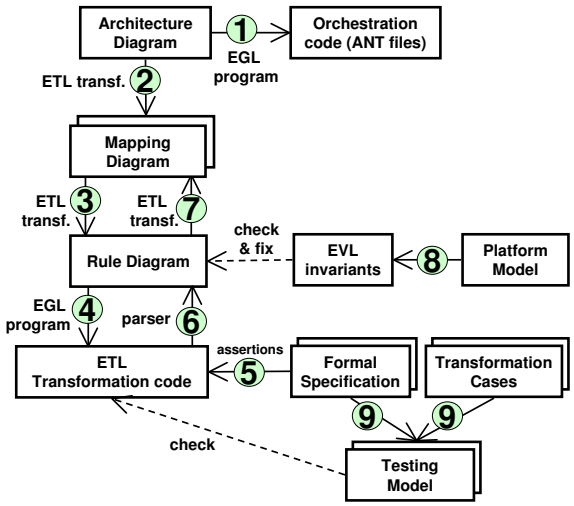

Fig. 20 Scheme of tool support.

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. The mapping diagrams are added a mapping for each concrete class defined by the input port types. Then, the transformation designer is in charge of connecting the mappings with the appropriate output 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 execution platform. In order to check whether a rule diagram fits a particular platform, we have created an EVL [39] program made of OCL-like constraints which validate a rule diagram for a specific platform model, discovering whether it conforms to the features of the platform, and automatically repairing the detected errors when possible (label “8”).

In “4”, code is generated from the structural rule diagram, taking into account the flow directives. In our current implementation we generate ETL code, and a parser for reverse engineering (label “6”) generates a rule diagram from ETL code. Although not shown in the figure, we support the generation of QVT-Relations code from structural diagrams as well, but not its reverse engineering.

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. When the execution of a transformation finishes, 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. 38.

In order to enable systematic transformation testing, a testing model is generated from the formal properties and transformation cases (label “9”). The generated test cases can be executed to test the final transformation. Currently, we only support model files as input test data, and all assertion types in our meta-model. Supporting other criteria for input model generation is left as future work. The execution of a test case for a given input model returns whether the result of transforming the model violates any of the specified assertions. As an example, Fig. 21 shows in the upper right window part of a test case (the complete test case was previously shown in Listing 2). The lower right window contains the result of running the test case.

Additionally, starting from the *trans*ML models, we generate different requirement traceability matrices which can be visualized in a web browser. In particular, we trace requirements against test cases to analyse whether every requirement has been tested, as well as against transformation cases and verification properties. Fig. 22 shows a screenshot of a requirements diagram being edited, while Fig. 23 shows the generated matrix tracing requirements (rows) against existing test cases (columns). Crossed cells have different colours depending on whether there is a direct traceability link between the requirement and the test case (e.g. requirement `Inherited Attributes` and test `InheritedAttrs`), or if the test covers a subrequirement of a given one (e.g. requirement `Features` and test `InheritedAttrs`). Moreover, each requirement in the matrix is linked to a web page containing the details of all its traces.

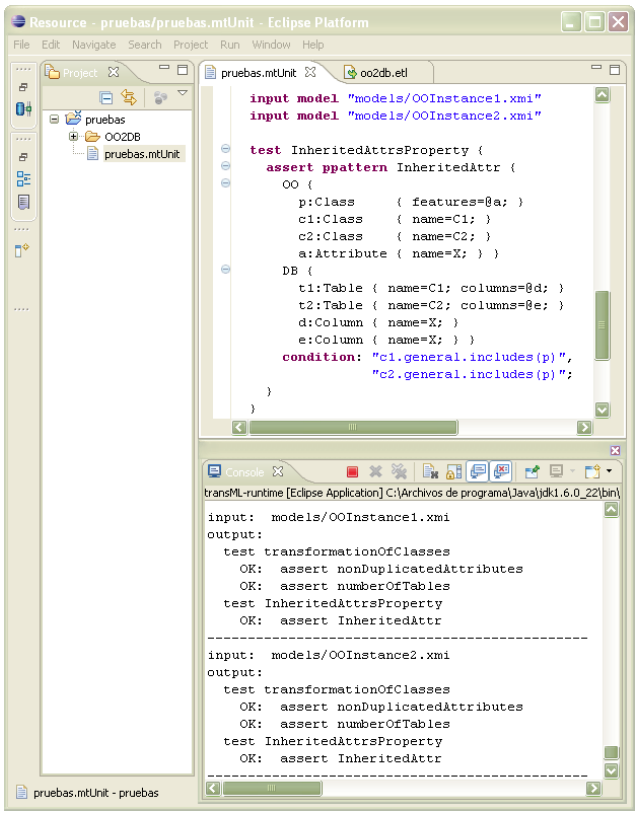

Fig. 21 Result of running a transformation test case.

10 Case studies

This section illustrates the use of *trans*ML with two real case studies. In the first one the aim was to translate models of productions systems into Petri nets for their analysis, and the chosen transformation language was QVT-Relations. The second example is an industrial project in the railway domain for which we used ETL.

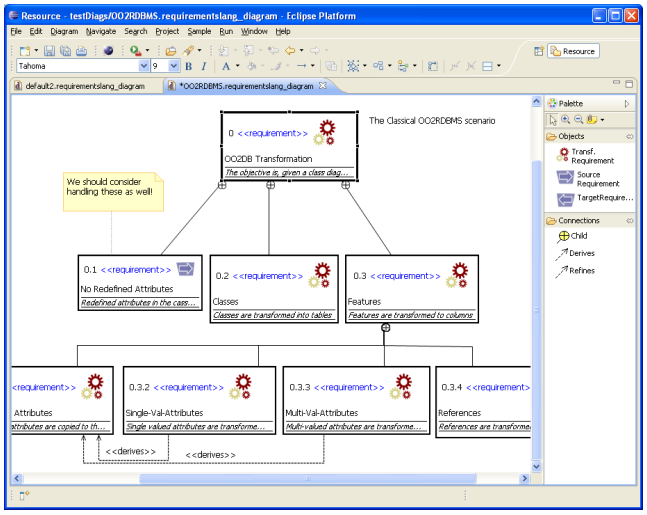

Fig. 22 Editor for the transformation requirements.

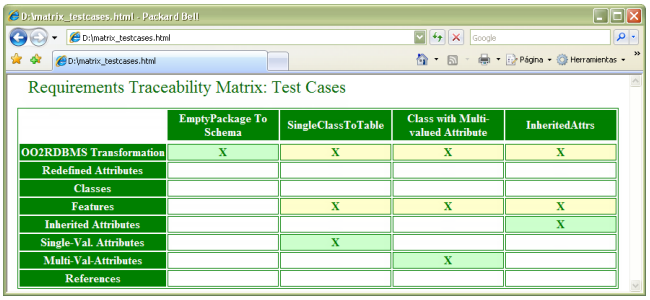

Fig. 23 Generated requirement traceability matrix.

10.1 From production plants to time Petri nets

In this section we describe the use of *trans*ML for building a transformation chain for the verification of production plant models [15] using time Petri nets [49]. Petri nets have powerful analysis techniques and hence are a frequently used analysis domain for higher-level modelling languages like UML or domain-specific languages [7].

We use a domain-specific language to describe production plants as nets of interconnected machines and conveyors with a certain capacity. There are four types of machines: generators of cylinders and bars, assemblers, and packaging machines. Generators introduce a given kind of part (a cylinder or a bar) in a factory conveyor, assemblers take one cylinder and a bar and produce an assembled part, and packaging machines remove assembled parts from the factory. All machines are characterized by some processing delay given by a uniform time interval, whereas for simplicity conveyors have zero transport time.

As a first step in the development of our transformation, we defined several transformation cases with examples of specific models or model fragments and the expected time Petri net in each case. For example, Fig. 24 depicts how the connection between two conveyors of capacity 4 and 3 should be expressed in time Petri nets, where in addition the target conveyor is full as it contains three parts (two cylinders and one assembled part). In this example we can see that conveyors should be transformed into a set of places, one for each kind of part (e.g. the places `cvs cyl`, `cvs bar` and `cvs assem` correspond to conveyor `cvs`). The Petri net representation for conveyors should also include an additional place to ensure the maximum capacity constraint of conveyors (places `cvs cap` and `cvt cap`). This transformation case is an example of situation where the target conveyor is full (zero tokens in the place for capacity constraint), so that no part from the incoming conveyor can move.

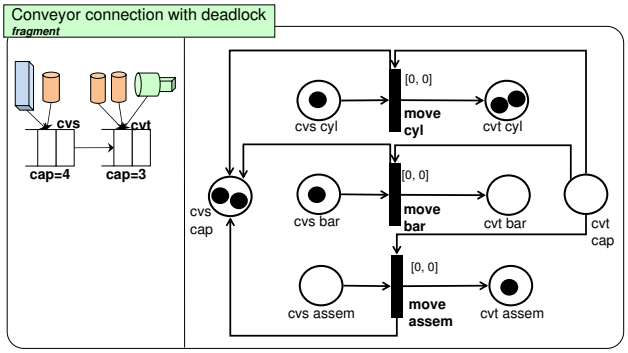

Fig. 24 Transformation case showing the time Petri net semantics of two connected conveyors.

Fig. 25 shows another transformation case, this time an example of the transformation of a complete factory model. The factory contains a generator of bars `gb`, a generator of cylinders `gc`, an assembler `asse`, and a packaging machine `pack`. One conveyor feeds pieces into the assembler, and another conveyor moves the assembled parts to the packaging machine. From this example we learnt how to transform the different kinds of machines and that unused places of conveyors should be deleted.

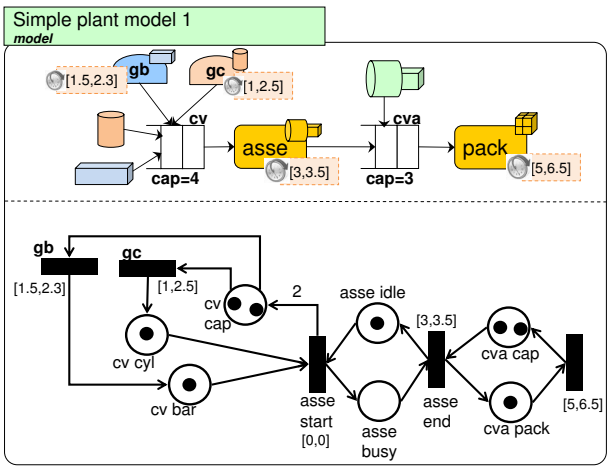

Fig. 25 Transformation case for simple plant model.

The Petri nets generated by our transformation must ensure the preservation of the maximum capacity for conveyors. As the transformation case in Fig. 24 showed, this should be implemented by an extra place connected to each transition, and removing or adding tokens to the set of places modelling the conveyor. More in detail, if a transition adds tokens (parts) to the places of the conveyor, it should remove the same amount of tokens from the capacity constraining place. This property can be formally expressed using our specification language as shown in Fig. 26. We also defined a similar pattern for incoming transitions to capacity places.

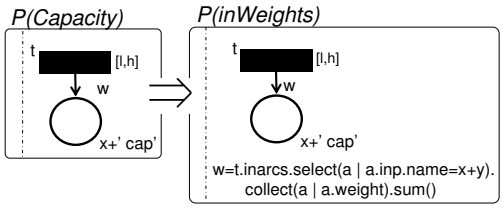

Fig. 26 Correctness property for the target model.

Other desired properties of the generated time Petri nets were the absence of non-used places (which could come from conveyors not connected directly or indirectly to some generator) as well as the absence of dead transitions (i.e. transitions that can never fire as they are connected to an incoming place with zero tokens and without any other input transition). These two properties were specified with the patterns shown in Fig. 27.

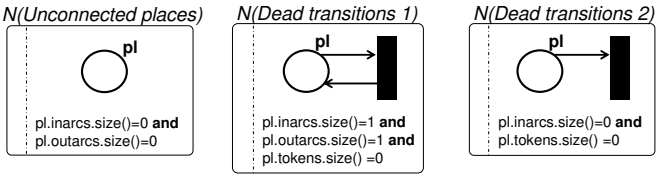

Fig. 27 Verification properties for the target model.

Regarding the transformation design, we decided to break the transformation in two steps: the first one transforming the factory models into time Petri nets, and the second one simplifying the resulting net by eliminating unconnected places and dead transitions. Altogether, our architectural diagram is shown in Fig. 28. The two transformation steps are included in a composite block, for which the output is constrained by some of the identified verification properties. Afterwards, a model-to-text transformation would produce code for its analysis with the Romeo tool [24].

Next, we built the mapping diagram shown in Fig. 29 for the first transformation step, named `ProdSys2TPNets`. The diagram only reflects the transformation of entities (i.e. conveyors, machines and parts) but not the net topology (i.e. connections). This was

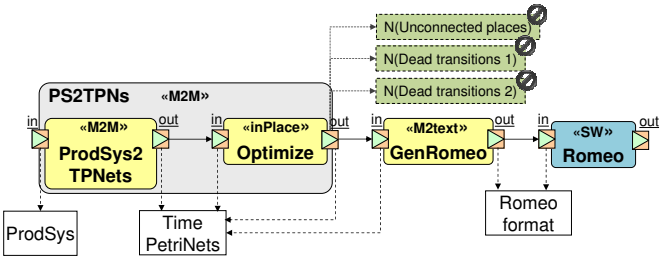

Fig. 28 Architecture of the transformation chain.

quite useful as we only wanted to focus on this aspect of the transformation, instead of having to specify all details. Thus, the diagram shows that any kind of generator should be translated into one transition, that any part should be transformed into a token, and that assemblers should be transformed into two places, two transitions and one token.

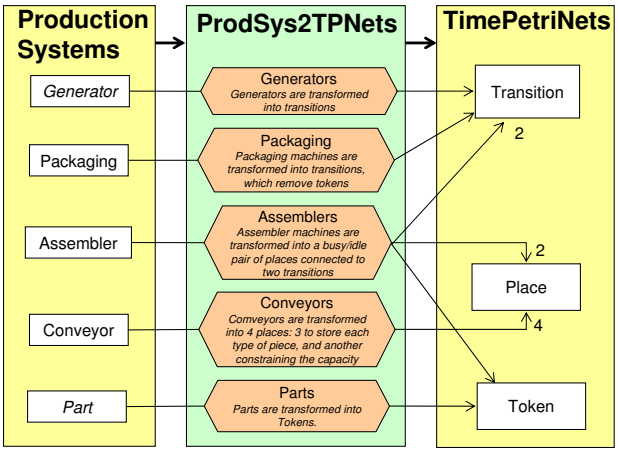

Fig. 29 Mapping diagram for the first transformation.

We chose QVT-Relations to implement the first transformation due to its declarative nature, which was very convenient because the source and target languages were heterogeneous and we had to create complex patterns in the target model. Fig. 30 shows the rule structure diagram, which is organized in two main blocks: *Entities* and *Topology*. The *Entities* block contains one rule to transform each entity kind, according to the mapping diagram of Fig. 29. The translation of parts is handled by block *genParts*, which is collapsed so that it only shows the number of rules it contains (4, one for each kind of part and another one for the capacity constraint places). The *Topology* block contains rules to connect the Petri net fragments generated by the first block. Indeed, these rules have dependencies with the rules in the first block. For instance, rule *Connect Packaging* can be executed only *when* rules *Packaging2Transition* and *Conveyor2Places* (this latter dependency indicated in the block) have been executed. Altogether, we found this diagram useful to understand the relationships between

rules. Moreover, it provided structuring mechanisms like blocks which were not present in QVT-Relations, as well as a compact notation to express common dependencies (e.g. we specified a when dependency for all rules in block *Topology* with a single arrow).

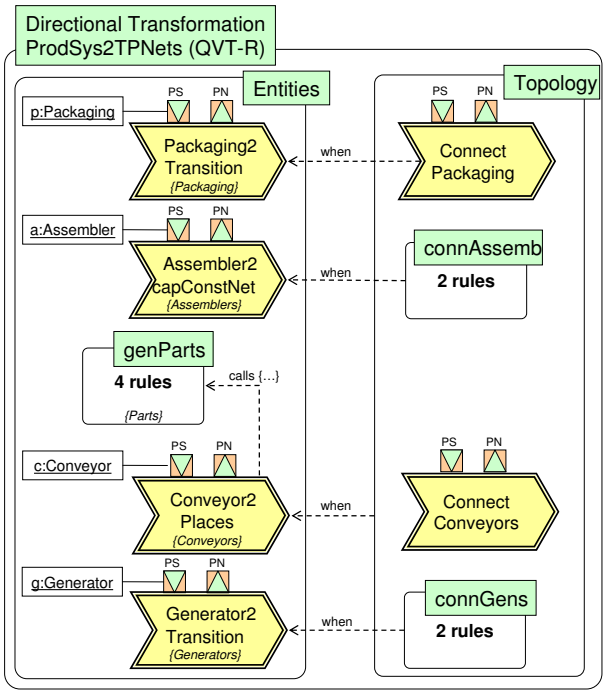

Fig. 30 Rule structure diagram.

Starting from our rule diagram, we used our code generator for QVT-Relations to generate a skeleton of the transformation implementation, which ended up with some 350 lines of code (LOC). The simplification transformation was written in EOL and contained 7 operations and 85 LOC. Finally, we wrote the code generator with EGL (60 LOC).

During the process, we also wrote some tests models. Some of them were derived from the transformation cases and the verification properties. Listing 4 shows an excerpt of a test model declaring two test cases. The first one (lines 4-7) tests the scenario of Fig. 25, while the second one (lines 12-43) checks the properties of Fig. 27 on three input models.

```
1  TestSuite "PS2TPNs.ant" <PS:"ProductionSystems",
2                           PN:"TimePetriNets"> {
3    @transformationcase(name="Simple Plan Model 1")
4    test SimplePlanModel1 {
5      input model "SimpleModel1.xmi"
6      assert model "PetriNetModel1.xmi".equals()
7    }
8
9    @property(name="Unconnected places")
10   @property(name="Dead transitions 1")
11   @property(name="Dead transitions 2")
12   test AssertionProperties {
13     input model "SimpleModel1.xmi"
14     input model "SimpleModel2.xmi"
15     input model "ConveyorSequence.xmi"
16
17     assert npattern UnconnectedPlaces {
18       PS {}
19       PN { pl:Place {} }
20       condition: "pl.inarcs.size()=0",
21                  "pl.outarcs.size()=0";
22     }
23
24     assert npattern DeadTransitions1 {
25       PS {}
26       PN { pl:Place { outarcs=@a1; inarcs=@a2; }
27            a1:ArcPT { outt=@t; }
28            a2:ArcTP { intt=@t; }
29            t:Transition {} }
30       condition: "pl.inarcs.size()=1",
31                  "pl.outarcs.size()=1",
32                  "pl.tokens.size()=0";
33     }
34
35     assert npattern DeadTransitions2 {
36       PS {}
37       PN { pl:Place { outarcs=@a; }
38            a:ArcPT { outt=@t; }
39            t:Transition {} }
40       condition: "pl.inarcs.size()=0",
41                  "pl.tokens.size()=0";
42     }
43   }
44 }
```

Listing 4 A test suite for the case study (excerpt).

Altogether, *trans*ML offered a step-by-step guideline to engineer the transformation. In particular, we intensively used the transformation cases, which were used for automated testing later. The specification language was useful to explicitly formalize knowledge of Petri net patterns and idioms (e.g. how capacity constraint places work, Fig. 26), as well as specific properties of our target models (Fig. 27). The architecture diagram was a means to modularize and organize the transformation. The mapping diagram provided a first sketch of how elements in both languages corresponded to each other, and served well for the purpose of understanding the transformation design. Finally, the rule structure diagrams allowed a more structured way to coding, and constitute a good means for documenting and understanding the transformation code.

10.2 From UML (for railway systems) into PROMELA

INESS (INtegrated European Signalling System – `http://www.iness.eu`) is an industry-focused project funded by the FP7 programme of the European Union, comprising 30 partners, including 6 railway companies. Its objective is to provide a common railway signalling system that integrates existing European ones. In the project, experts have been modelling a specification of the proposed integrated signalling system using xUML [47]. This is a subset of UML comprising class diagrams and state machines, as well as an action language to specify class operations and state actions. The idea is to use the specified xUML models to check for inconsistencies in the requirements and against core properties of the system provided by professional railway engineers. Currently, xUML models can be analysed only via simulation, but due to safety-critical requirements involved in railway signalling systems, our work

in the project is to enable the analysis of models using formal verification.

In order to achieve this goal, we are using a transformation-based approach for the formal analysis of the xUML models. This entails on the automatic translation of xUML models to the input language of formal verification tools, like SPIN [32]. While previous works in the literature [44,45] have analysed subsets of UML (in particular Statechart diagrams) using model checkers, the main feature of this project is that we deal with the action language of xUML in its full generality.

Because of the research nature of the project, we were not given initial requirements about the transformation. Instead, they began emerging as we started designing the transformation. In the first stages, we agreed on some guidelines for the experts to build the xUML models. Expressing these requirements using *trans*ML verification patterns automates checking if a given xUML model satisfies these guidelines and therefore qualifies for the transformation. Some of the main requirements included: (i) classes always have exactly one associated behaviour, of type state machine; (ii) multiple-inheritance is not allowed; (iii) a transition always has a unique trigger; (iv) a change-event can be associated to at most one trigger; (v) transition triggers have to be of type change-event, time-event or signal-event; (vi) a special class called "Application" is used to instantiate a scenario (representing a railway track layout) for the execution of the model; (vii) objects can only be created in the state-machine of the "Application" class. As an example, Fig. 31 shows the verification pattern formalizing requirement (i), while Fig. 32 partially describes requirement (vii). For this latter requirement, another similar pattern is needed, to check the creation of objects in transition actions.

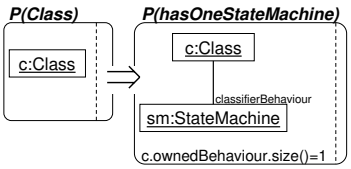

Fig. 31 Pattern expressing requirement (i).

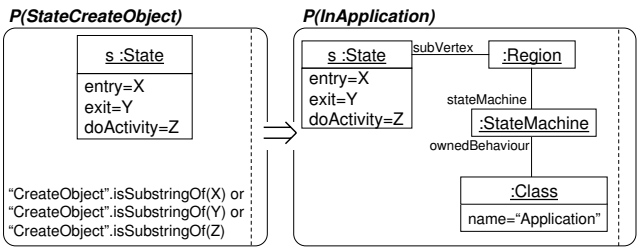

Fig. 32 Pattern expressing requirement (vii).

The main difficulties found in our transformation were dealing with both the action language and inheritance in the xUML language. First, dealing with the action language in its full generality means that certain actions have to be specially encoded in the target language (formal languages normally comprise a more basic set of statements). Second, since every class has an associated state-machine, by inheriting a class, two state machines have to be executed in parallel in the target language. Additional sources of complexity included collecting the possible state configurations in the state machines, the transitions that can fire in parallel as well as resolving the order of firing entry and exit actions.

In order to deal with these issues in a more structured and efficient way, we decided to split the transformation in several steps. The architecture representing the transformation to the input language of the SPIN model-checker is shown in Fig. 33. 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. In this way, the first transformation performs a flattening of the classes and states machines, and the second one transforms the obtained tbUML model into a PROMELA model – the input language to SPIN – from which code conforming to the PROMELA grammar is generated as input to SPIN.

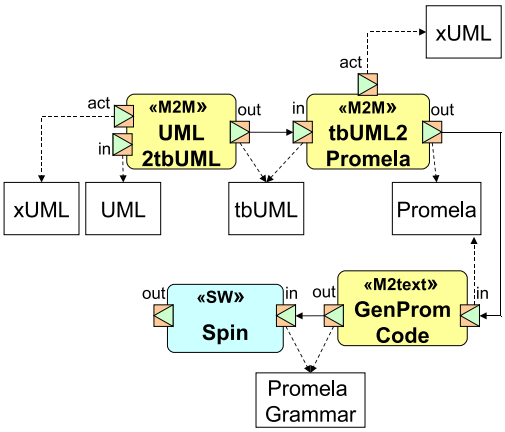

Fig. 33 Architecture diagram for the project.

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 described in Fig. 34, the other requirements about the flattening of classes are defined similarly. 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. We were also able to

express these requirements using our specification language.

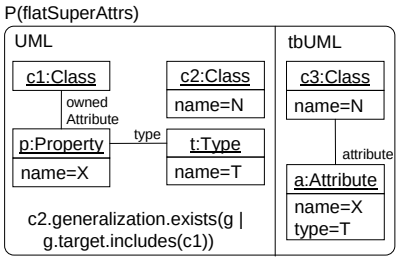

Fig. 34 A verification property for UML to tbUML.

We chose ETL to implement the transformations and EGL for code generation. We did not define a mapping diagram for the first transformation (UML to tbUML) as it was straightforward, but we built the mapping diagram shown in Fig. 35 for the second transformation (tbUML to PROMELA). In this case, *StateNode*s and *Signal*s are translated to *Constant*s with a unique value; *Object*s (i.e. classes that appear in the "Application" class) are translated to a *Proctype* which represents a PROMELA execution unit; and models are transformed into another model and an *Init* process that is responsible for starting the objects in the model – for example, creating communication channels and setting references accordingly.

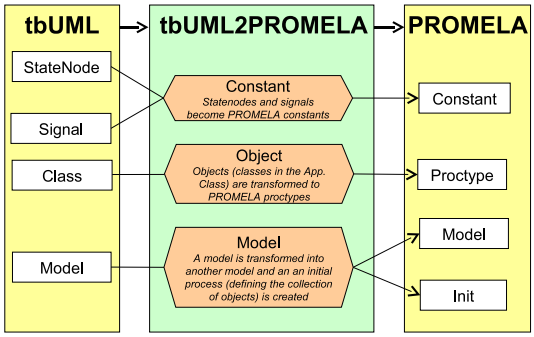

Fig. 35 Mapping diagram for tbUML to PROMELA.

In this way, the complexity of our transformations did not come from a large number of rules, but from the complexity of the helper computations associated to certain rules. For this reason, the rule diagrams for this case study made intensive use of *helpers*. Fig. 36 shows the rule structure diagram for the second transformation, which displays the two main helpers in the transformation: *parseActions* and *createInit*. The former is called whenever an action is found in the transition body/entry and exit body of a transition and a state. The function is responsible for translating the xUML action language into PROMELA constructions. The *createInit* helper method is used by rule *Mod2Mod*, and creates an *Init* process in the target model, provided the correct reference to the objects (i.e. classes in the "Application" class).

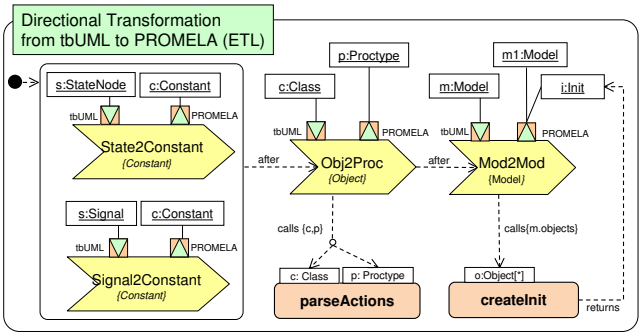

Fig. 36 Rule structure diagram for tbUML to PROMELA.

We also attached a behaviour diagram to some of the rules, giving additional details about their expected implementation. As an example, Fig. 37 shows the diagram that corresponds to rule *Model2Model* of the first transformation. This creates a tbUML model with two special states (called *StateNode*s) starting from a UML model. The state node `root` is created in order to maintain a reference that can be used to compute the Least Common Ancestor during the flattening of the state machines – all other states are within this one. The state node `init` represents the initial pseudo-states in the state-machines of the model. This way, initial pseudo-states are abstracted and the initial actions found in them are directly associated to the classes.

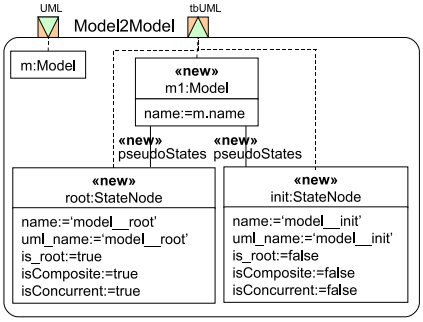

Fig. 37 Behaviour diagram for rule *Model2Model*.

The implementation of the first transformation in our architecture ended up with more than 5000 lines of ETL code, whereas the second one had more than 2500 LOC. In order to validate both implementations, we made use of the verification patterns specified at the first stages of our design. In particular, we generated assertion code for the run-time verification of the transformations. Fig. 38 shows a moment in the execution of the first transformation, 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. Additionally, we built some testing mod-

els which enabled automated testing of the verification properties as well as checking the correct translation of specific input UML models.

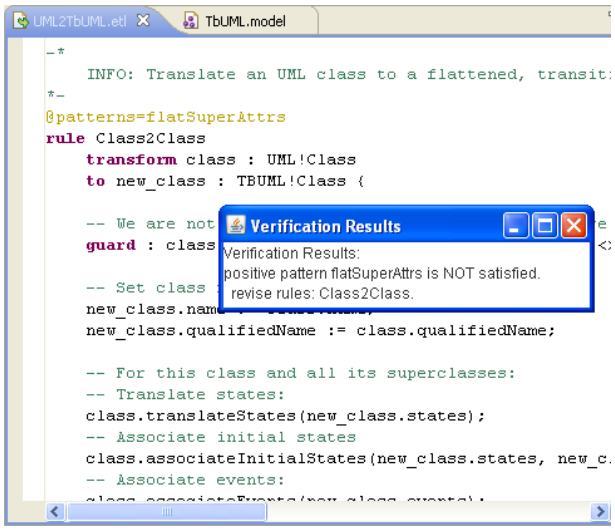

Fig. 38 Testing the implementation.

Altogether, in this project we found particularly useful the formal specification language, as it enabled to formally gather requirements about the input models as well as verification properties for the transformation. Guaranteeing the satisfaction of these safety-critical requirements was essential due to the application domain of the project. Moreover, we were able to provide a means to automatically validate such properties on input xUML models. However, our specification language cannot be used to express dynamic correctness criteria for the transformation (e.g. that a step in the Statechart corresponds to a step in Promela), which is left for future work. For this project, the rule structure diagrams were less useful, as the complexity of the transformations were not on the number of rules or on their data dependencies, but it was on the operations and helpers used by the rules. Nonetheless, rule structure diagrams were really useful in the first case study due to the complexity of rule dependencies. Hence, we can conclude that each transformation project has its own characteristics, and engineers need to select the most useful *trans*ML diagrams for each particular situation, just like software engineers select the most useful kinds of UML diagrams to construct software systems.

11 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 complete 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 several case studies, 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. Regarding our language for testing, we are currently developing tool support for the automatic generation of input test models according to different coverage criteria, combining techniques based on meta-models, constraints and specifications as seed for the generation. We are also planning the use of *trans*ML in further case studies, and investigating *processes* for transformation development.

Acknowledgements. We thank the referees for their useful comments. This work has been sponsored by the Spanish Ministry of Science and Innovation with project METEORIC (TIN2008-02081), and by the R&D program of the Community of Madrid with projects "e-Madrid" (S2009/TIC-1650). Parts of this work were done during the research stays of Esther and Juan at the University of York, with financial support from the Spanish Ministry of Science and Innovation (grant refs. JC2009-00015, PR2009-0019 and PR2008-0185).

References

1. A. Agrawal, A. Vizhanyo, Z. Kalmar, F. Shi, A. Narayanan, and G. Karsai. Reusable idioms and patterns in graph transformation languages. *Electron. Notes Theor. Comput. Sci.*, 127:181–192, 2005.
2. A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and et al. Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 224–248. Springer, 2010.
3. B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.
4. K. Beck. *Test Driven Development: By Example.* Addison-Wesley, 2003.
5. J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In *UML*, volume 2863 of *LNCS*, pages 175–189. Springer, 2003.
6. J. Bézivin, F. Jouault, and J. Paliès. Towards model transformation design patterns. In *EWMT'05*, 2005.
7. A. Bondavalli, I. Mura, and I. Majzik. Automatic dependability analysis for supporting design decisions in uml. In *HASE'99*, pages 64–72, 1999.
8. A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *FASE'06*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.
9. E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE*, pages 85–94. IEEE CS, 2006.

10. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
11. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA - visual automated transformations for formal verification and validation of uml models. In *ASE'02*, pages 267–270. IEEE CS, 2002.
12. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
13. A. Darabos, A. Pataricza, and D. Varró. Towards testing the implementation of graph transformations. *Electron. Notes Theor. Comput. Sci.*, 211:75–85, April 2008.
14. J. de Lara and E. Guerra. Formal support for QVT-Relations with coloured Petri nets. In *MoDELS'09*, volume 5795 of *LNCS*, pages 256–270. Springer, 2009.
15. J. de Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22:297–326, 2010.
16. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
17. H. Ehrig and U. Prange. Formal analysis of model transformations based on triple graph rules with kernels. In *ICGT'08*, volume 5214 of *LNCS*, pages 178–193. Springer, 2008.
18. A. Etien, C. Dumoulin, and E. Renaux. Towards a unified notation to represent model transformation. Technical Report RR-6187, INRIA, 2007.
19. 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.
20. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In *TAGT'98*, volume 1764 of *LNCS*, pages 296–309. Springer, 2000.
21. F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8:185–203, 2009.
22. S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2009. See also `http://www.omg.org/spec/SysML/1.1/`.
23. Fujaba. `http://www.fujaba.de`.
24. G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A tool for analyzing time Petri nets. In *CAV*, volume 3576 of *LNCS*, pages 418–423. Springer, 2005.
25. 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.
26. P. Giner and V. Pelechano. Test-driven development of model transformations. In *MoDELS'09*, volume 5795 of *LNCS*, pages 748–752. Springer, 2009.
27. P. V. Gorp, A. Keller, and D. Janssens. Transformation language integration based on profiles and higher order transformations. In *SLE*, volume 5452 of *LNCS*, pages 208–226. Springer, 2008.
28. E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A visual specification language for model-to-model transformations. In *VLHCC'10*, pages 119–126. IEEE CS, 2010.
29. E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. *trans*ML: A family of languages to model model transformations. In *MoDELS (1)*, volume 6394 of *LNCS*, pages 106–120. Springer, 2010.
30. E. Guerra, J. de Lara, and F. Orejas. Inter-modelling with patterns. *Software and Systems Modeling*, In press, 2011.
31. A. Habel and K.-H. Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 293–308. Springer, 2005.
32. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
33. M. Iacob, M. Steen, and L. Heerink. Reusable model transformation patterns. In *3M4EC'08*, pages 1–10, 2008.
34. E. Insfrán, J. Gonzalez-Huerta, and S. Abrahão. Design guidelines for the development of quality-driven model transformations. In *MoDELS-10*, volume 6395 of *LNCS*, pages 288–302. Springer, 2010.
35. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
36. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. See also `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`. Last accessed: Nov. 2010.
37. A. Kleppe. MCC: A model transformation environment. In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 173–187. Springer, 2006.
38. 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.
39. D. S. Kolovos, R. F. Paige, and F. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *ICST'08*, pages 356–364. IEEE CS, 2008.
40. 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.
41. D. S. Kolovos, R. F. Paige, L. M. Rose, and F. A. Polack. Unit testing model management operations. In *MoDeVVa'08*, pages 97–104. IEEE CS, 2008.
42. A. Kusel. TROPIC - a framework for building reusable transformation components. In *Doctoral Symposium at MODELS*, 2009.
43. J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDELS Workshops*, volume 4364 of *LNCS*, pages 193–204. Springer, 2007.
44. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the SPIN model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.
45. J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In *ASE*, pages 255–258, 1999.
46. Y. Lin, J. Zhang, and J. Gray. A framework for testing model transformations. In *Model-driven Software Development - Research and Practice in Software Engineering*. Springer, 2005.
47. S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

48. J. Merilinna. A tool for quality-driven architecture model transformation. Master's thesis, Technical Research Centre of Finland, 2005.
49. P. M. Merlin and D. J. Farber. Recoverability of communication protocols. *IEEE Trans. Computers*, 24(9), 1976.
50. B. Meyer. Applying "design by contract". *Computer*, 25:40–51, October 1992.
51. F. Orejas. Symbolic graphs for attributed graph constraints. *J. Symb. Comput.*, 46(3):294–315, 2011.
52. QVT. `http://www.omg.org/docs/ptc/05-11-01.pdf`.
53. L. A. Rahim and S. B. R. S. Mansoor. Proposed design notation for model transformation. In *ASWEC'08*, pages 589–598. IEEE CS, 2008.
54. A. Rensink. Representing first-order logic using graphs. In *ICGT'04*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.
55. 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.
56. P. Sampath, A. C. Rajeev, K. C. Shashidhar, and S. Ramesh. Verification of model processing tools. *Int. J. Passeng. Cars Electron. Electr. Syst.*, 1:45–52, 2009.
57. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
58. S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *ICMT*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.
59. 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.
60. J. M. Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.
61. J. Steel and M. Lawley. Model-based test driven development of the Tefkat model-transformation engine. In *ISSRE'04*, pages 151–160, 2004.
62. I. Stürmer, M. Conrad, H. Dörr, and P. Pepper. Systematic testing of model-based code generators. *IEEE Trans. Software Eng.*, 33(9):622–634, 2007.
63. A. van Lamsweerde. *Requirements Engineering. From System Goals to UML Models to Software Specifications.* Wiley, 2009.
64. 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.
65. D. Varró. Model transformation by example. In *MODELS'06*, volume 4199 of *LNCS*, pages 410–424, 2006.
66. A. Yie, R. Casallas, D. Deridder, and D. Wagelaar. Realizing model transformation chain interoperability. *Software and Systems Modeling*, In press, 2011.