Increasing the Quality of Model Transformation with the Use of Design Patterns

Hüseyin Ergin
Department of Computer Science
University of Alabama
Tuscaloosa AL, U.S.A.
Email: hergin@crimson.ua.edu

***Abstract*—Model transformation is at the heart of model-driven development techniques. The development of model transformations typically consists of designing rewriting rules that are applied on model instances. However, the lack of systematic development methodology and re-use hamper the quality of model transformations. This study presents existing work from the literature on quality evaluation of model transformation as well as the elaboration of model transformation design patterns. We also introduce a design pattern example that demonstrates the impact on some quality aspects of model transformation.**

***Keywords*—model transformation, design patterns, model transformation quality**

I. Introduction

Models are the principal artifacts in model-driven engineering (MDE). They are subject to all kinds of manipulations [1] such as: refactoring, simulation, transformation, comparison, merging. Model transformation therefore plays a pivotal role in MDE. However, current model transformation development suffers from re-use and design guidelines. Moreover, quality concerns are often neglected by transformation developers, which hinder the overall quality of the deployed product. For this reason, we propose to investigate existing quality frameworks and design patterns of model transformation.

A. Background on Model-Driven Engineering

MDE [2] is considered a well-established software development approach that uses *abstraction* to bridge the gap between the problem and the software implementation. MDE uses models to describe complex systems at multiple levels of abstraction. Models are first class citizens and represent an abstraction of a real system, capturing some of its essential properties. Models are instances of modeling languages which define their abstract syntax, concrete syntax and semantics. The *abstract syntax* defines the essence of the language, often defined by a *metamodel* The *concrete syntax* defines the graphical or textual representation of the elements of the metamodel. *Semantics* defines the meaning of the language. The static semantics is specified by the metamodel extended with constraints, while the dynamic semantics is often defined by means of a model transformation (either denotational or operational). A model expressed in a modeling language *conforms* to its metamodel. Metamodels themselves are also modeled in a modeling language called metamodeling language, which has a conceptual foundation called *metametamodel*. Models, metamodels and metametamodels form a three-level architecture in MDE and these levels are called M1, M2, M3 respectively [3].

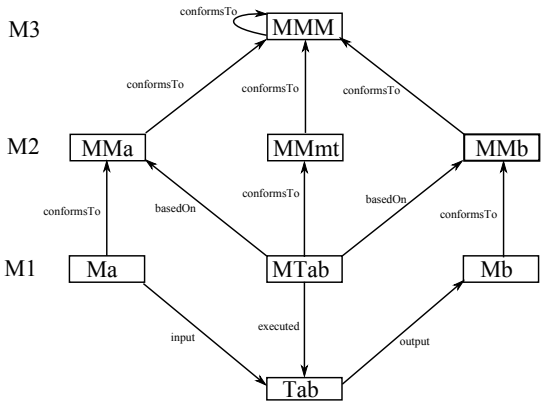

Fig. 1. Model transformation schema in MDE.

In MDE, the core of the development process consists of a series of transformations over models. Typically, a transformation or manipulation is modeled by a *model transformation* that conforms to a specific metamodel. Following Jouault *et al.* [4], the model transformation schema in MDE in Fig. 1 illustrates these terms. Ma and Mb are models that conform to metamodels MMa and MMb respectively. MTab is a model transformation that conforms to metamodel MMmt and takes a model as an input and produces another model as an output. MTab is also based on metamodels of both input and output models. All three metamodels, MMa, MMb, MMmt conform to a standard metametamodel.

Rule-based transformation is the most used paradigm to define a model transformation. Finding a solution to a problem as model transformation consists of designing rules and identifying scheduling between them. Therefore, there is a need for re-usable, proven and qualified structures in this design phase. A design pattern encapsulates a proven solution to a recurring design problem [5]. Good practices in the design of transformations as well as the assessment of high quality transformations are still missing and hinder the design of large-scale transformations. As similarly established in the object-oriented paradigm [6], standardizing and codifying good practices in the form of design patterns of model transformation can solve these quality issues and increase overall quality.

The remaining of the study is organized as follows: section II explores model transformation in more details. The

structure of model transformation languages, intents of model transformation and most common model transformation languages are described. Section III presents information about the quality criteria, metrics to evaluate, and guidelines for a quality-driven model transformation. Section IV reviews the structure of object-oriented design patterns. In Section V, existing model transformation design patterns in the literature are analyzed. We also introduce a new one in order to show the relation between quality and design patterns. The challenges found are listed in Section VI. Section VII summarizes and concludes the study.

II. Model Transformation

A model transformation is defined as "an automated manipulation of models according to a specific intent" in [7]. These intents play an important role while creating a model transformation and are explained in Section II-B. The detailed structure of a model transformation is explained in Section II-A. A transformation mainly consists of source and target languages, transformation rules, and scheduling of the rules. In Section II-C we highlight some distinctions between three model transformation languages.

A. Structure

In [8], model transformation approaches are investigated using domain analysis methods. The following eight features are reported:

- *Specification* mechanism may be pre/post conditions expressed in the Object Constraint Language (OCL) [9], a function between source and target models and a model transformation's being executable or not.
- *Transformation rules* are the smallest units of transformation which are explained in details in Section II-A1.
- *Rule application control* has two aspects; location determination on the models where the rules are applied and scheduling of the rules. The scheduling of the rules determines the order in which they are executed. This is explained in Section II-A2.
- *Rule organization* comprises general structuring issues, such as modularization and re-use mechanisms.
- *Source-target relationship* specifies whether the source and target models are manipulated under different metamodels.
- *Incrementality* means updating the existing target model based on changes in the source models.
- *Directionality* defines whether the transformation is unidirectional (from source to target), bi-directional (a single transformation defines the round-trip computation), or multi-directional (in case more than two models are involved).
- *Tracing* is concerned with the (temporary or persistent) trace links between source and target models, and between rule applications.

From these features two are the most important in this study while deducing/applying design patterns; transformation rules and scheduling of them. In the next subsections, these two are explained in details.

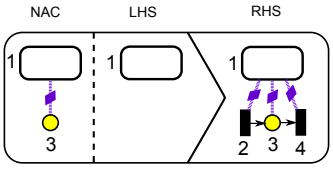

Fig. 2. A sample model transformation rule.

1) Transformation Rules: Transformation rules are the smallest units of a model transformation. A transformation rule has many different features according to [8]. The *domain of a rule* defines how a rule can access elements of models. A rule is a declarative construct that dictates *what* shall be transformed and not *how*. It consists of pre-condition and post-condition patterns. The pre-condition pattern determines the applicability of a rule: it is usually described with a left-hand side (LHS) and optional negative application conditions (NACs). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. Constraints can be also be specified over the attributes of LHS and NAC pattern elements. The right-hand side (RHS) describes the post-condition pattern that must be found in the output model after the rule is applied. Imperative actions can be also be specified over the attributes of RHS pattern elements. An advantage of using the rule-based transformation paradigm is that it allows to specify the transformation as a set of operational rewriting rules instead of using imperative programming languages. A rule example with LHS, RHS and NAC parts is depicted in Fig. 2. This rule is taken from a model transformation that translates an UML activity diagram model to a behaviorally equivalent Petri net model [10]. The rule can be read as "if an activity (labeled 1) that is not associated with a place (labeled 3) is found, then create a place and two transitions (labeled 2 and 4), and relate them with temporary trace links". This rule has a *graphical syntax* using elements from concrete syntax of the source and target domains (activity diagrams and Petri net).

The *body structure* of a rule may consist of strings, terms, or graphs. It holds *variables* from source and/or target languages. Each body may have a *logic* that expresses computations or constraints on model elements. Logic may be object-oriented or functional, and executable or not. Typing of the domain is also important. It can be *untyped*, which is the case of textual templates, *syntactically typed*, in which a variable in the rule is associated with a metamodel element, or *semantically typed*, which allows behavioral properties. The rules may have a *syntactic separation*, which clearly seperates the parts operating on one model from the parts operating on other models. *Multidirectionality* of the rules means executing the rule in different directions. A rule may be applied from the source to target languages direction, vice-versa, or both. There may be some *application conditions* on rules to determine the rule to be executed. The condition must be true in order to execute the rule. Execution of a rule may require creation of some additional *intermediate structures*. These structures is usually temporary and require their own metamodel. The most common example of intermediate structures are traceability links. A rule can be *parametrized* by using control parameters, generics and higher-order rules. Control parameters allow

passing values as control flags. Generics allow passing data types, including model element types. Higher-order rules take other rules as parameters and may provide even higher level of reuse and abstraction. *Reflection and aspects* from imperative programming languages may be used as well in transformation languages.

Each transformation language modifies the structure according to its needs. Some examples are explained in Section II-C.

2) Scheduling: Rule scheduling is an important phase in the development of a model transformation. Scheduling mechanisms determine the order in which individual rules are applied [8]. One can distinguish between implicit and explicit scheduling. When the scheduling of a transformation language is *implicit*, the modeler has no direct control over the order in which the transformation units are applied. On one hand, a transformation language can be *unordered*, i.e., it simply consists of a set of rules. In this case, the order of application of the rules is entirely determined at run-time. It completely depends on the patterns specified in the rules. Applicable rules are selected non-deterministically until none apply anymore. The scheduling of a language can be *explicitly* specified by the modeler. In explicit internal transformation languages, a rule may explicitly invoke other rules. For example in ATL [4], a matched rule (implicitly scheduled) may invoke a called rule in its imperative part. Also, a rule tagged as lazy is applied only after all other rules have been applied. Another example of an explicit internal transformation language is QVT-R [11]. There, the *when/where* clauses of a rule may have a reference to other rules: for when, the former is applied after the latter and for where, the latter is applied after the former. Finally, in an explicit external transformation language, there is a clear separation between the rules and the scheduling logic. Ordered transformations specify a control mechanism that explicitly orders rule application of a set of rules. Examples are: priority-based, layered/phased, or with an explicit work flow structure. Most transformation languages are partially ordered, however. That is, applicable rules are chosen non-deterministically while following the control specification. Another sub-category of explicit external transformations is event-driven transformations, which have recently gained popularity. In these transformation systems, rule execution is triggered by external events.

Rule iteration is the control structure that allows recursion, looping and fix point iteration in the scheduling phase. Transformation languages may also be organized into several *phases*. Each phase may have a specific purpose and only some certain rules can be invoked in a given phase.

Rule scheduling may also exhibit the following properties according to [12]: *atomicity* which means all rules succeed or they all fail, *branching* which is the execution of a sub-structure based on a condition, *non-determinism* which is about the ordering of the rules, *parallelism* which means the rules can be applied in parallel or not, *back-tracking* which is the explicit roll-back mechanism after execution, *hierarchy* which is the rule nesting.

As in rule structure, each model transformation language modifies the rule scheduling properties according to its needs.

B. Intents

A model transformation intent is a description of the goal behind the model transformation and the reason for using it [7]. Classifying model transformation by intents is of paramount importance when working on design patterns. It helps target specific patterns for specific use cases and ensures they are useful in practice. Below is a list and basic description of these intents from [7]:

1) *Manipulation:* Simple atomic or bulk operations on a model is considered model transformation when the system is modeled.
2) *Restrictive Query:* A query requests for some information about a model and outputs a proper sub-model a.k.a. a *view*.
3) *Refinement:* Refinement produces a lower level specification (e.g., a platform-specific model) from a higher level specification (e.g., a platform-independent model).
4) *Abstraction:* Abstraction is the inverse of refinement.
5) *Synthesis:* A model is synthesized into a well-defined language format that can be stored, such as serialization.
6) *Reverse engineering:* Reverse engineering is the inverse of synthesis: it extracts higher level specifications from lower level ones.
7) *Approximation:* Approximation is a refinement with respect to negated properties.
8) *Translational Semantics:* The semantics of a language can be defined in terms of another formalism. In this case, the semantic mapping function of the original language is defined by a model transformation to a reference formalism with well-defined semantics.
9) *Analysis:* A model transformation can be used to analyze a modeling language in terms of another well-known and well-analyzed formalism.
10) *Simulation:* A simulation is a model transformation that updates the state of the system modeled. Simulation defines operational semantics.
11) *Normalization:* Normalization aims to decrease the syntactic complexity of models by translating complex language constructs into more primitive constructs.
12) *Rendering:* This is the assignment of a concrete representation to each abstract syntax elements or group of elements.
13) *Model Generation:* The metamodel of a language can be defined by a graph grammar. Then execution of this graph grammar leads to model transformations able to generate all possible instances of the language.
14) *Migration:* Migration is a transformation from a software model written in one language or framework into another language, keeping the models at the same level of abstraction.
15) *Optimization:* This model transformation aims at improving operational qualities of models such as scalability and efficiency.
16) *Refactoring:* Model refactoring is a restructuring that changes the internal structure of the model to improve certain quality characteristics without changing its observable behavior.
17) *Composition:* Model composition integrates models that have been produced in isolation into a compound model. *Model merging* creates a new model such that every element from each model is present exactly once in the

merged model and *model weaving* creates correspondence links between overlapping entities.

18) *Synchronization:* Model synchronization integrates models that have evolved in isolation but that are subject to global consistency constraints.

Further classifications of model transformations are described in [13]. Source and target languages define another distinction between model transformations. A transformation is endogenous when it operates on models conforming to the same metamodel. Optimization and refactoring can be examples of endogenous transformations. A transformation is exogenous when it operates on models conforming to different metamodels. Synthesis, reverse engineering and migration are examples of exogenous transformations. A *horizontal* transformation is a transformation where the source and the target models are in the same abstraction level. Refactoring and migration are horizontal transformations. A *vertical* transformation is a transformation where the source and the target models are in different abstraction level. Refinement is a vertical transformation. A *syntactic* transformation is merely based on transforming the syntax. Reverse engineering for example, takes a language in one syntax to another language in another syntax. There are also *semantic* transformations such as simulation and optimization. *In-place* transformations are executed within same model such as simulation and *out-place* transformations produce a different model after execution.

C. Model Transformation Languages

There are many model transformation languages in the literature. Some examples are Henshin [14] from Arendt *et al.*, which is a language that operates on models in Eclipse Modeling Framework (EMF) and has visual syntax, editing functionalities, execution and analysis tools; GReAT [15] from Agrawal *et al.*, which consists of three distinct parts; pattern specification language, graph transformation language and control flow language; FUJABA [16] from Klein *et al.*, which is one of the first tools to do code generation from UML models and UML model generation from code; VIATRA2 [17] from Varro and Balogh, which provides a rule and pattern-based language for manipulating graph models by using graph transformation and abstract state machines; and AGG [18] from Taentzer, which lets graphs to be attributed by Java objects and equips graph transformation with computations on these objects.

Each of these languages have a unique set of structure combinations (e.g., different rule and scheduling structure, different directionality). Jouault and Kurtev [19] compared a number of model transformation languages in terms of transformation scenarios, paradigm, directionality, cardinality, traceability, query language, rule scheduling, rule organization and reflection.

In this section, some common model transformation languages are described in details. These are QVT [11] from Kurtev, ATL [4] from Jouault *et al.* and MoTiF [12] from Syriani and Vangheluwe. All three languages work on metamodels eventually created in MOF-like languages.

1) QVT: QVT is an acronym for "Query, View, Transformation". QVT is the OMG standard language for specifying

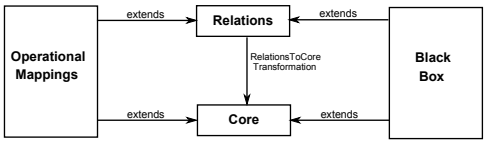

Fig. 3. QVT Languages layered architecture [11]

model transformations. The OMG defined the terms in QVT as follows in [20]:

- "*Query* is an expression that is evaluated over a model. The result of the query is one or more instances of types defined in the source model, or defined by the query language".
- "*View* is a model which is completely derived from another model which may be base model. There is a live connection between the view and the base model."
- "*Model Transformation* is a process of automatic generation of a target model from a source model, according to a transformation definition."

QVT abstract syntax is defined as a metamodel and QVT transformations are models conforming to the metamodel. QVT metamodel defines three sublanguages for model transformation. These are QVT Relations (QVT-R), QVT Core (QVT-C) and QVT Operational Mappings (QVT-OM). The layered architecture of these three languages is depicted in Fig. 3.

QVT-R provides capabilities for specifying transformations as a set of relations among models. It is based on arbitrary number of domains, but there must be at least two domains, as well as pre- and post-conditions. QVT-R can be used in these scenarios: *check-only* verifies that the given models satisfy the relations with a boolean output; *unidirectional transformation* creates a target model from a source model according to some relations; *model synchronization* verifies and makes sure that a set of models satisfy the relations of the transformation, *in-place update* modifies a model as specified by the relations.

Every relation has a set of object patterns. These patterns can be matched in the source model and can be produced in the target model. QVT-R handles the traceability links automatically and hides related details from developer. QVT-R is designed to be multi-directional and support both inplace and outplace transformations. The code sample from [11] represents a portion of a model transformation transforming UML class diagram into relational database schemas. The relation transforms attributes of classes to columns of relational database tables.

```
1 relation AttributeToColumn {
2     checkonly domain uml c:Class {};
3     enforce domain rdbms t:Table {};
4     primitive domain prefix:String;
5
6     where {
7         PrimitiveAttributeToColumn(c, t, prefix);
8         ComplexAttributeToColumn(c, t, prefix);
9         SuperAttributeToColumn(c, t, prefix);
10    }
11 }
```

The keywords are important for the semantics of the transformation. *Checkonly* means domain elements cannot be

changed, so the source domain is read-only. *Enforce* indicates that elements of the domain are subject to change in order to ensure the relation holds, so the target domain is writable. *Primitive* is used for passing parameters. *Where* clause provides the possibility of invoking other relations, which must be evaluated to true for the invoking relation to hold. Different combinations of these keywords change scenarios of QVT-R mentioned above.

QVT-C is also a declarative language. It is however simpler than QVT-R, being defined at the level of mappings. These two languages can handle the same transformation scenarios. Unlike QVT-R, the traceability links must be defined explicitly in QVT-C. A QVT-R relation can be translated into a semantically equivalent QVT-C mapping. This transformation is defined in QVT-R [20] and is therefore an instance of a *higher-order transformation*.

QVT-OM language extends QVT-R with imperative constructs and OCL constructs. The basic idea of the language is that object patterns specified in the relations are instantiated by using these constructs. Therefore, the structure of the language provides imperative language constructs such as loops, conditions, etc. QVT-OM transformations are unidirectional.

Black box mechanism in the architecture allows plugging in and execution of external code during the transformation. It allows complex algorithms in any language and enables use of existing libraries.

2) ATL: ATL is an acronym for "Atlas Transformation Language". It also provides a set of languages as in QVT. Atlas Model Weaving (AMW) is a higher abstraction level specification. The ATL Virtual Machine executes the compiled ATL programs.

ATL transformations are unidirectional. The transformation operates on read-only source models and produces write-only target models. Since the source model is read-only, the transformation can only be out-place. Source model can be navigated during transformation, but target model can't be navigated.

The declarative part of ATL uses *matched rules* consisting of source and target patterns. Source pattern is matched in the source model and target pattern is created in target model for every match. ATL provides automatic traceability links between target and source elements. The *called rules* are declarative matched rules that must be invoked explicitly. ATL also has an imperative part, *action blocks*, that consist of sequences of imperative instruction that can be used in both rules.

There are two modes that ATL programs can work. When executed in *standard mode*, the ATL transformation creates elements only if they are matched. Therefore, unmatched elements are not created in target model. In *refining mode*, unmatched elements are automatically created in target model without need of a rule.

The following code snippet depicts a matched rule in ATL language and from [4]. The rule is part of a transformation from UML class diagram to relational database tables.

```
1 rule Class2Table {
2     from
3         c : Class!Class
4     to
5         out : Relational!Table (
6             name <- c.name,
7             col <- Sequence {key}->union(c.attr->
                  select(e | not e.multiValued)),
8             key <- Set {key}
9         ),
10    key : Relational!Column (
11        name <- objectId
12    )
13}
```

The *from* clause defines a read-only source pattern that is used only for matching. The *to* clause defines a target pattern that is created in the target domain. The rest of the target pattern is the mapping of the attributes from classes to tables.

There are three types of matched rules in ATL according to the way they are triggered.

- *Standard rules* are applied once in every match on source models.
- *Lazy rules* are called by other rules. They can be applied multiple times which produces a different set of target elements at the end of each application.
- *Unique lazy rules* are like lazy rules. The difference is they use existing target elements instead of creating new ones.

Rule scheduling is implicit and not an issue in ATL. Since the source domain is read-only, it is not modified during execution and transformation always results in same target model. ATL executions result in a deterministic target model unless lazy rules are used.

3) MoTif: MoTif is a short name for "Modular Timed Graph transformation". MoTif and its semantics is based on the Discrete Event System Specification (DEVS) formlism [21]. It also introduces an explicit notion of time because of DEVS and it allows to model the interruption for every rule in the execution. Under MoTif, there is T-Core [22], which stands for "Transformation Core". It is a collection of primitive operators for model transformation. T-Core offers the following eight primitives:

- *Matcher* finds all possible matches of the condition on the graph embedded. After matching, it stores all the matches in the packet.
- *Rewriter* applies the required transformation on the match specified in the packet it received.
- *Iterator* chooses a match among the set of matches of current condition of the packet. The match is chosen randomly and choosing the match continues until a maximum number is achieved.
- *Resolver* resolves a potential conflict between matches and rewritings by prohibiting any changes to other matches in the packet.
- *Rollbacker* is used as a recovery point that allows backward recovery of packets.
- *Selector* is used when a choice needs to be made between multiple packets processed concurrently by different constructs.
- *Synchronizer* is used when multiple packets processed in parallel need to be synchronized.

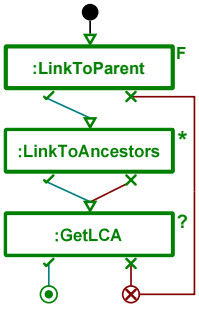

Fig. 4. Scheduling of MoTif Rules.

- *Composer* is a modular encapsulation of the elements and any other primitives can be added to the encapsulation.

Fig. 2 depicts an example rule in MoTif. The rule is part of another study [23] and basically add some new petrinet elements to UML activity diagram action nodes while preventing the rule to be applied more than once by adding a NAC and maintaining traceability links to the original nodes. The rule consists of three parts. First part is NAC and separated with dashed line from other parts. Many number of NACs can be added to a rule. Second part is LHS on the left of big arrow. That is the precondition pattern to be found in the input model. Third part is RHS on the right of big arrow and the post-condition pattern to be applied to the model. As one can easily realize, concrete syntax can be used in rule designing phase.

Rule scheduling in MoTif is explicitly defined by another structure, which is also modeled. The structure allows to define what happens when the rule is matched or not. A sample scheduling sequence is depicted in Fig. 4.

The rules are single lined green boxes and they have input ports and output ports for success and failure. Success case is finding the match in the input. All output ports can be connected to any other rule's input ports or the output ports of the block.

MoTif consists of rule blocks [24]. Each rule block can be either atomic or composite. Some of the atomic rule block can be found in Fig. 4 and are listed below. The content of these rule blocks appears in Fig. 11.

- *ARule:* means a regular atomic rule. It is a simple rule that is executed only once.
- *FRule:* is 'For all Rule'. The matches are found for the input model and this rule is applied to all found matches. For example in Fig. 4, rule LinkToParent is an FRule.
- *SRule:* means 'Star Rule' and applies the transformation to all matches as long as the rule is applicable. Therefore it is applied to the resulting model cumulatively after each application. For example in Fig. 4, rule LinkToAncestors is an SRule.
- *QRule:* means 'Query Rule' and mostly consists of only LHS and NACs For example in Fig. 4, rule GetLCA is an QRule.

Composite rule blocks allow one to encapsulate the composition of rule blocks. Some of them express flow structures, such as branching and looping.

III. Quality in Model Transformations

Some studies in the literature attempt to identify quality criteria with respect to model transformation ([25], [26], [27]) and metrics ([28], [29], [30], [31]) to measure these criteria.

This section, focuses on Mohaghehi and Dehlen's quality framework for MDE [25] and Insfran *et al.*design guidelines for the development of quality-driven model transformations [32]. The analysis of these two papers is extended with quality criteria and metrics.

A. Mohagheghi and Dehlen's Quality Framework for MDE

Mohagheghi and Dehlen [25] propose a framework for evaluating MDE projects in general. Inspired by the ISO-9126 recommendation, they adapt the activities carried to MDE. They identify following steps:

1) Identify quality criteria, such as maintainability and re-usability, which are discussed in Section III-A1.
2) Identify target objects that have an impact on quality criteria. These objects are metamodels, models, languages, transformations; *i.e.,* all concepts related to model transformation.
3) Identify the properties of target objects that have an impact on quality criteria. Identifying these properties are based on: purpose of the target objects, life cycle, relation with other objects, scale of the project, specificity of the project, and lifetime. Following the example objects in the previous step, these properties can be elements, relations, and constraints for metamodels, size, and modularity for models, scheduling structure, and rule structure of languages.
4) Specify how to evaluate the quality properties. This includes the metrics to be measured quantitatively or subjective evaluation of the transformation. These metrics are discussed in Section III-A2. Other approaches may be empirical evaluation by interviewing the users or inspections using checklists.
5) Specify traceability links between quality properties and quality criteria.
6) Execution is the implementation of quality properties and evaluation of metrics identified in step 4.

Fig. 5 illustrates these steps.

1) Quality Criteria in Model Transformation: Several works have focused on defining quality attributes to models [26]. Mohagheghi and Dehlen [25] propose quality criteria with respect to model transformation. A more detailed quality criteria listing is done in [27] as follows:

- *Correctness:* Correctness is defined as including the right elements and correct relations between them [26]. It also means not violating rules and conventions. This includes sticking to the transformation metamodel for the model transformation.
- *Re-usability:* Re-usability is the most important criteria in design pattern. Actually it is the main purpose of creating/trying to find a design pattern. Re-usability can also be satisfied by having smaller generic pieces in the transformation language, such as unit rules.
- *Efficiency:* In general, efficiency means fulfilling the purpose without wasting so much resource [33]. Model

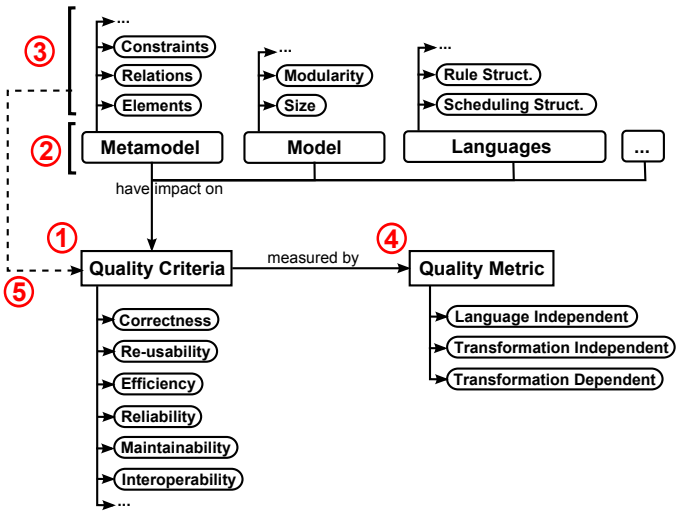

Fig. 5. Quality Framework for MDE (adapted to Model Transformation)

transformation tools mostly suffer from NP-Complete pattern matching problem [27]. Therefore, efficiency is an important criteria in model transformation when it comes to huge inputs.

- *Reliability:* Reliability is the characteristic of the code that it is expected to perform its intended functions satisfactorily [33]. In terms of model transformation, any rule must satisfy the condition of the domain as intended.
- *Maintainability:* The model transformation must be maintainable to fit to the new requirements or modifications. That means the transformation must be understandable, testable and modifiable.
- *Interoperability:* Interoperability means a model transformation must be able to work with outer systems, which may be another transformation model, software or technology.

This is not a complete list of quality criteria with respect to model transformation, but the most relevant ones are listed.

2) Metrics for Transformation Languages: Metrics are essential for assessing the quality of model transformations. In [34], the authors distinguished three different categories of transformation quality metrics: language independent, transformation independent and transformation dependent.

Language independent metrics are about the general structure of the model transformation languages which are mentioned in Section II. Size of rules, number of matches, number of rule applications can be in this category. General software metrics can also be counted such as execution time and memory constraint.

Transformation independent metrics focus on specific model transformation languages. The studies mainly focus on ATL and QVT languages, which are briefly explained in Section II-C. In this category, Vignaga [28] proposed 81 metrics for ATL such as number of clones of a piece of code, number of received calls, and number of keywords. Kapova *et al.* [31] proposed a set of metrics for QVT Relations like declarative languages such as number of relations, number of enforced domains, average number of local variables per relation, and number of 'when' predicates.

Transformation dependent metrics are specific to particular model transformation problems. These rely on some specific properties of the model transformation. For example a model transformation that flattens a state chart model must make sure that there is a bisimulation between the original and the flat version.

Relating the metrics with the appropriate criteria is also important. Vignaga [28] related the metrics he proposed for ATL with the quality criteria like understandability, modifiability, re-usability etc.

B. Insfran et al.*'s Design Guidelines for the Development of Quality-driven Model Transformations*

Insfran *et al.* propose a guideline for the development of quality-driven model transformation in the case where alternative solutions exist. The proposed guidelines lead the designer to select appropriate alternative transformation that satisfies certain quality criteria. They offer three artifacts to be used while building the guidelines:

- *The Quality Model:* this model lets the transformation designer build a set of quality criteria and related metrics with them.
- *The Transformation Model:* this model is basically the transformation language that has rules, characteristics, and structural elements that are transformed into some other structural elements.
- *The Active Rules Model:* the active rules are transformation rules that are selected among alternatives.

The steps of guidelines are:

- *Identifying and Selecting Alternative Transformation Rules:* the aim of this step is to find alternative transformation rules that can transform a structure in the source model into a different structure in the target model. This helps to have different quality transformation rules in model transformation.
- *Refactoring the Transformation Rules:* refactoring is needed to make the rules more concise so that cohesion is maximized and coupling is minimized. The authors claim that large and complex rules have less flexibility and reusability. Large and complex rules also mean a complex relation with quality criteria.
- *Defining Transformation Rules:* the authors mention two kinds of rules in this step. Top-level rules represent the alternative transformations of a structure and each regular creation/modification rule becomes a non top-level rule. Top-level rules set the selected alternatives as active rules.
- *Avoiding Conflicts among Rules:* a conflict occurs if a pre-condition or LHS of multiple rules overlap. The granularity of a transformation rule is the size of LHS. The larger the rule LHS covers in the input model, the larger its granularity is. The goal is to minimize granularity.
- *Building the Transformation Model:* the transformation model consists of associations among structures with alternative transformations. It also covers impacts on different quality attributes. Trade-off analysis among quality attributes are used to build the transformation model.

Guidelines introduced in the study look appropriate and reasonable for creating a model transformation. However, the

study does not provide any concrete materials about quality in model transformation. Also how to identify quality differences between alternative transformations is left to domain experts.

IV. Design Patterns

Design patterns are reusable structures that can help to overcome any problem to be solved again and again. Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, is such a way that you can use this solution a million times over, without ever doing it the same way twice [6]. There are many studies involving hundreds of design patterns in the literature. Each of them describe a solution to some kind of a problem. The use of design patterns leads to the construction of well-structured, maintainable and reusable software systems [35].

Pattern cataloging process began as a part of Erich Gamma's PhD thesis [6]. Then the other authors joined the process and design patterns found the last appropriate structure to be published as a book. There are now 23 standard object oriented design patterns in the book. Actually before Gamma *et al.* [6]'s work, there were still programming languages that use design patterns without mentioning the strict name. For example model-view-controller structure in Smalltalk-80 is an earlier example of a design pattern [36].

A. Structure

The essential elements of design patterns are explained in Gamma *et al.* [6]. These are the four main primitive elements under a design pattern and basically used to express a design pattern's purpose and results. They are listed as follows:

- The **pattern name** is actually a handle to summarize all other fields in the design patterns essential elements. It lets developers to freely talk and understand the design as an abstraction. Finding a good name is one of the hard parts of developing a pattern.
- The **problem** describes when to apply the pattern. Mostly the problem and its context are explained in this field. These may include specific design problems, class or object structures and a list of conditions that must be met before application.
- The **solution** describes the elements that are parts of the design, their relationships, responsibilities and collaborations. Since a pattern is like a template, it has to provide a solution to be applied in many different situations. The solution is generally given with UML class diagrams.
- The **consequences** are the results and trade-offs of applying the pattern. These are critical for evaluating design alternatives and for understanding the costs and benefits before applying the pattern. Language and implementation issues, impacts on a system's flexibility, extensibility and portability may also help users to understand and evaluate design patterns.

Other than these elements, there are more fields in [6] to describe a design pattern. These are; pattern name and classification, intent, also-know-as, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, related patterns. *Intent* is an important field both in design patterns and model transformations that needs to be mentioned more. This field is the key to select the right design pattern for a problem and the right constructs for a model transformations. Although the rest of the fields are not crucial for a design pattern, they help to explain it better and precise.

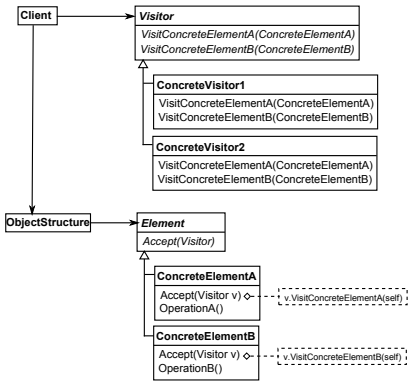

Fig. 6. Structure of Visitor Pattern [6]

Hasheminejad and Jalili [5] also introduce two main sections to categorize these fields: problem domain and solution domain. The *problem domain* describes the problem context where the pattern can be applied and has these fields: intent, motivation and applicability. The *solution domain* describes the structure and collaborations of the pattern solution being applied to the problem and has these fields: structure, participants, collaborations, consequences, implementation and related patterns.

B. Example

In this subsection, one of the design patterns from [6] are shown to make how a design pattern looks like more concrete. For this purpose, the visitor pattern is arbitrarily chosen.

1) **Visitor Pattern:** Visitor pattern is an object behavioral pattern. Object behavioral patterns deal with the interaction and responsibility of objects.
   - *Intent:* It allows one to add a new operation to a class structure without altering the structure.
   - *Applicability:* It helps keeping similar operations together. Therefore, a new operation can be added easily without adding so many lines or changing the structure of target classes.
   - *Structure:* The structure of Visitor pattern is depicted in Fig. 6.
   - *Participants*
     - *Visitor* defines the interface that has a seperate method to access and process each ConcreteElement.
     - *ConcreteVisitor* implements the Visitor interface and provides another set of features about what will happen when the element is visited.
     - *Element* defines the interface which has an Accept method for a visitor.
     - *ConcreteElement* implements the Element interface and also provides necessary methods to be used when a Visitor visits.

- *ObjectStructure* represents the context and data structure of the elements in the program.

- *Collaborations:* A client must create a ConcreteVisitor to traverse ConcreteElements in the program. An element calls related visit method in the ConcreteVisitor when it is traversed.
- *Consequences:* It is easy to add new ConcreteVisitors, since each visitor will already be visited by the ConcreteElement. However adding new ConcreteElements is hard and needs to add a visit operation in each ConcreteVisitor.

C. Limits of Design Patterns

Design patterns are accepted to be a useful structure in the aim of re-usability and readability. They have some advantages and disadvantages of course. A design pattern means a set of high-level documentation of the design. Design pattern abstracts presentation from the implementation.

The goal of design patterns is to increase quality metrics to satisfy some quality criteria. This often comes with a trade-off. In design patterns, re-usability criteria mostly conflicts with efficiency criteria. For example, visitor pattern lets people traverse class structure and process on them in an efficient way, while it requires nearly double number of new classes created. Also, it is not clear how many design patterns are enough in a project. There is a probability that you mess up your code by applying too many unnecessary design patterns.

Another point, applying design pattern is not automated yet and it still depends on the manual decision of the designer. Hasheminejad and Jalili [5] proposed an automatic two-phase method for design pattern selection, but this does not reduce the impact of the designer in selection process.

However, re-usability, readability and maintainability are so important criteria that this makes design patterns always popular.

V. Model Transformation Design Patterns

Solving a model transformation problem is exactly analogous to solving other software problems. One can generate a naive model transformation solution for a specific problem, while another can generate a highly optimized and efficient solution. Design patterns help to create optimized and efficient solutions for problems by providing necessary steps or structures as discussed in Section IV.

In this section, existing model transformation design pattern studies in the literature are reviewed. Agrawal *et al.* [37] name design patterns as "Reusable Idioms and Patterns" and introduce three design patterns in graph transformation languages. Iacob *et al.* [38] prefer "Reusable Model Transformation Patterns" as name and introduce five design patterns. These are the first studies in the literature that tries to create a design pattern catalogue. The main problems with these studies are: 1) The fields of a design pattern *i.e.,* benefits, applicability, motivation etc. are not handled correctly. They can sometimes have same and redundant sentences. 2) They do not provide a generic solution for the design patterns in terms of a design pattern formalism. They provide only a specific solution in a specific model transformation language. This problem is discussed more as a challenge in Section VI-A. 3) Any of design patterns introduced in these studies are not evaluated in terms of quality criteria. In the last subsection, a design pattern is identified by focusing on the lowest common ancestor (LCA) problem [39]. The relation between model transformation quality and design patterns are also depicted with this problem and identified design pattern. Design patterns can be identified in different ways. One way is focusing on one problem and trying to find different solutions to that problem. Starting with naive solutions to the more efficient solutions yield us to a reusable method for solving that problem. At the end, a design pattern can be identified with a high probability.

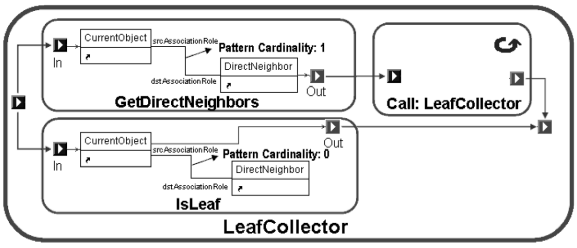

Fig. 7. Structure of Leaf Collector Pattern [37]

Nonetheless, these are the first studies and need to be inspired and improved in terms of productivity.

A. Reusable Idioms and Patterns in Agrawal et al.

Agrawal *et al.* [37] used GReAT model transformation language [15] to provide three design patterns. In this study, each design pattern has *motivation*, *applicability*, *structure* and *known uses* fields as introduced in Section IV-A. They also added *limitation* to extend consequences and *benefits* to promote the advantages of the pattern. They worked on a concrete problem which is flattening of a hierarchical dataflow to a flat dataflow representation. The design patterns introduced in this study are:

1) *The Leaf Collector Pattern:*
   - *Motivation:* This pattern aims collecting all leaf nodes in a hierarchy.
   - *Applicability:* The leaf collector pattern starts from a root node, traverses all nodes and tries to collect leaf primitives. A leaf primitive is a node where one can't traverse further. It can be applied to the problems that fit this purpose.
   - *Structure:* The structure of the pattern is depicted in Fig. 7. The structure is more like a recursive step in the pattern. GetDirectNeighbors rule collects all direct neighbors of the input node and call Leaf Collector again. IsLeaf rule is deciding if the input node is a leaf or not by checking direct neighbor count further that node.
   - *Benefits:* Getting the neighbors and leaf recognition are independent. Therefore, it can be modified according to other needs.
   - *Known Uses:* The transformation from hierarchical state machine to finite state machine can use this pattern.
   - *Limitations:* This pattern is suitable for only acyclic graphs, since it is traversing the nodes recursively and without marking any visit.

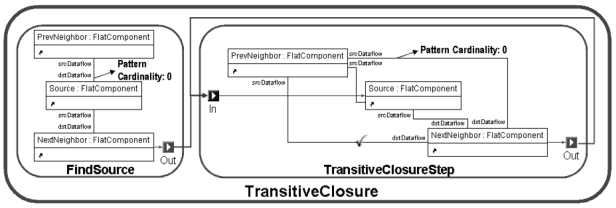

Fig. 8. Rules of Transitive Closure [37]

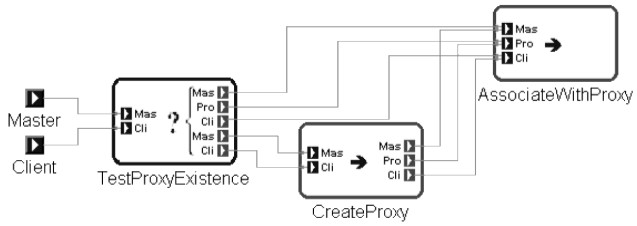

Fig. 9. Rules of Proxy Generator Idiom [37]

2) *Transitive Closure:*
   - *Motivation:* Some problems need a data dependency analysis. For these kinds of analyses, transitive closure of the data needs to be found. This pattern can be used for these cases.
   - *Applicability:* This pattern can be applied to compute the transitive closure of a graph.
   - *Structure:* In Fig. 8, the necessary rules are depicted to compute the transitive closure. First, next neighbor is found in FindSource rule. Then, the found next neighbors are used as an initial starting point for the TransitiveClosureStep rule.
   - *Known Uses:* The reachability analysis on ESML (Embedded Systems Modeling Language) is known to use this pattern.
   - *Limitations:* This pattern is suitable for only acyclic directed graphs.
3) *The Proxy Generator Idiom:* This is a reusable but domain specific design pattern.
   - *Motivation:* This pattern is used to generate proxies in a distributed system. A proxy is a placeholder for another object and provides access control to the main object.
   - *Applicability:* This pattern can be applied in any distributed system which has a structure like server/client, service/request or source/sink.
   - *Structure:* The structure of the pattern is depicted in Fig. 9. First, the system is checked for an already existing proxy with TestProxyExistence rule. If there is a proxy, it is associated with the client by applying AssociateWithProxy rule. If the proxy can't be found in the system, a new one is created with CreateProxy rule and the AssociateWithProxy rule is called again.
   - *Benefits:* The pattern has independent steps for check, create and associate steps. Therefore, each step can be modified according to needs.
   - *Known Uses:* Embedded System Modeling Language uses this pattern.
   - *Limitations:* The pattern is applicable with a client and a master pair and a request between these two.

B. Reusable Model Transformation Patterns in Iacob et al.

Iacob *et al.* [38] used QVT-R language [11] to introduce five model transformation design patterns. They fixed the pattern structure to have *goal*, *motivation*, *specification*, *example* and *applicability* fields. *Motivation* field describes the problem kinds that the pattern can solve. *Specification* field gives the solution using QVT-R language. *Applicability* field hints the places where this pattern can be applied. The introduced design patterns are:

1) *The Mapping Pattern:*
   - *Goal:* The mapping pattern creates one-to-one relations between a source metamodel and a target metamodel.
   - *Motivation:* Mapping is always an issue in transformation and is used in many places. It is most useful when source and target models use different languages or syntax, but have the same meaning.
   - *Specification:*

```
top relation XYMapping {
    nm: String;
    enforce domain left x: X {
        context = c1 : XContext {},
        name = nm };
    enforce domain right y: Y {
        context = c2 : YContext {},
        name = nm    };
    when {
        ContextMapping(c1,c2); }}
```

     The relation above from [38] states that an element x of type X must be related with an element y of type Y. Also x and y have the same name and the same context, which is ensured in ContextMapping function in the *when* part of the relation.
   - *Applicability:* This pattern can be applied to translate a model from one syntax to another *e.g.,* from UML to Java.
2) *The Refinement Pattern:*
   - *Goal:* The pattern provides a more detailed target model by refining an edge/node in source model to multiple edges/nodes.
   - *Motivation:* Refinement pattern is useful when detail steps of an existing model are requested.
   - *Specification:*

```
top relation RelationRefinementMapping {
    n : String;
    enforce domain left e1 : Edge {
        name = n,
        ... };
    enforce domain right im_node {
        context = c2 : Context {} };
    enforce domain right e2 : Edge {
        ... };
    enforce domain right e3 : Edge {
        target = t_right : Node {},
        ... };
    when {
        ContextMapping(c1,c2);
        ElementMapping(s_left,s_right);
        ElementMapping(t_left,t_right); }}
```

     In the relation above from [38], the edge from the left domain is refined into a node with two edges.
3) *The Node Abstraction Pattern:*
   - *Goal:* The pattern provides an abstraction mechanism for any node while keeping the relations of the node.

- *Motivation:* This pattern can be used to eliminate some specific nodes from the model. It is also assumed that source and target have same metamodels.
- *Specification:*

```
top relation Node_X_Abstraction {
    enforce domain left s1 : X {
        inEdge = e_in : Edge {
            name = na_in : String,
            source = ss1 : Node {}},
        outEdge = a_out : Edge {
            name = na_out,
            target = tt1 : Node{}} };
    enforce domain right a : Node {
        name = na_in + na_out,
        source = ss2 : Node {},
        target = tt2 : Node {} };
    when {
        NodeMapping(ss1,ss2);
        NodeMapping(tt1,tt2); }}
```

In the relation above from [38], s1 of type X is abstracted from the target model while preserving its adjancent edges.

- *Applicability:* This pattern can be applied when certain elements from models are to be removed.

4) *The Duality Pattern:*
- *Goal:* This pattern generates a semantic dual of the given model.
- *Motivation:* There are two main categories to represent dynamic behavior of a system. First type focuses on the procedural flow of activities which lead to a larger activity. Second type focuses on the flow of control from state to state. One may want to translate between these two class of semantic or want to assign our model a semantic purpose according to these two.
- *Specification:*

```
top relation ArrowNodeMapping {
    nm: String;
    enforce domain left a: Arrow {
        context = c1: AContext {},
        name=nm };
    enforce domain right v: Vertex {
        context = c2: VContext {},
        name=nm };
    when {
        ContextMapping(c1, c2); }}

top relation NodeArrowMapping {
    nm: String;
    enforce domain left v:Vertex {
        context = c1: NContext{},
        ... };
    enforce domain right a:Arrow {
        context = c2: AContext {},
        ... };
    when {
        ContextMapping(c1, c2);
        v.outgoing->size()=1;
        v.incoming->size()=1;
        ArrowNodeMapping(e1, v1);
        ArrowNodeMapping(e2, v2); }}
```

The two relations above from [38] first create mappings between arrows in source model and nodes in target model. Then, each node's arrows are matched to another set of nodes again.

- *Applicability:* This pattern can be used to relate models expressed in different languages but have a duality relationship. An example may be transformation from UML activity diagram to UML state chart diagrams.

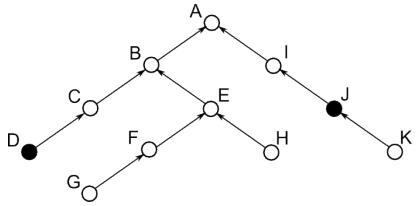

Fig. 10. The input model to an LCA problem.

5) *The Flattening Pattern:*
- *Goal:* This pattern aims to remove hierarchy from source model and generates flattened version.
- *Motivation:* Hierarchical structure mostly aims to make the models easier to understand. But a flattened version may be needed to analyze them.
- *Specification:*

```
top relation CompositeFlattening {
    checkonly domain left c: Composite {
        context = c1 : CompositeContext {} };
    enforce domain right r: RootElement{};
    when {
        RootMapping(c1,r) or
        CompositeFlattening(c1,r); }}

relation ElementMapping {
    nm: String;
    enforce domain left x: Element {
        name = nm,
        context = c1 : Context {} };
    enforce domain right y: Element {
        name = nm,
        context = c2 : Context {} };
    when {
        RootMapping(c1,c2) or
        CompositeFlattening(c1,c2); }}
```

In the two relations above from [38], the transformation is endogenous. Note also that a unique RootElement is assumed. All elements are belong to the RootElement or a Composite element, which represents a hierarchy. Then each element is either applied RootMapping or CompositeFlattening recursively.

- *Applicability:* This pattern can be used to remove hierarchy from the source model.

C. Case Study: Lowest Common Ancestor

In this section, a design pattern is identified from the well-known problem of computing the LCA. Aim of this case study is relating quality criteria and design patterns. First, LCA problem is solved naively. Then, an alternative solution that improves the quality of the model transformation is introduced. Finally, the solution is generalized to a design pattern that can be applied to similar problems.

LCA is defined between two input parameter nodes and tries to find the lowest shared ancestor between them. In Fig. 10, an input model to LCA problem is depicted. In this input, the LCA of the nodes D and J can be computed and found to be the node A.

In the naive approach for solving this model transformation problem, new temporary links are created to all ancestors of all

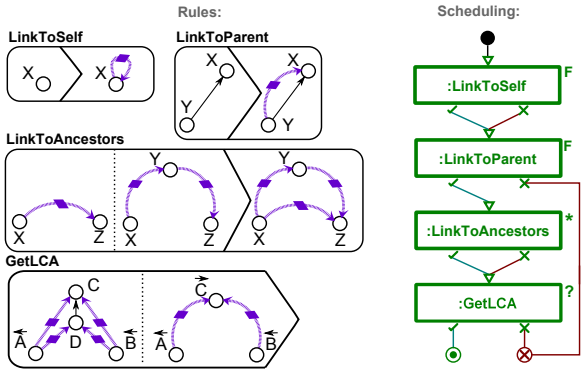

Fig. 11. The first set of rules to solve LCA problem.

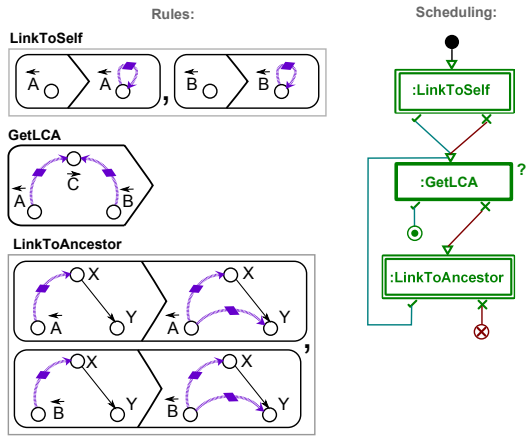

Fig. 12. The second set of rules to solve LCA problem.

nodes. Then input parameter nodes are matched if they have a common ancestor. The necessary three rules and the scheduling of them are depicted in Fig. 11. The LinkToSelf rule creates self-ancestor links. The LinkToParent rule creates the first level of ancestor links between each node and their parents. If this rule fails, that means the input model doesn't have any connected edges. Then, LinkToAncestors rule creates the ancestor links for all ancestors of nodes. As a result of these two rules, all nodes in the input model have ancestor links to both parents and all ancestors until root node. Finally, GetLCA rule matches the lowest common ancestor of the input parameter nodes. This rule has a NAC which guarantees the matched node is the lowest among the other matched ones.

The problem with the naive approach is to deal with too many unnecessary ancestor links. The solution creates ancestor links for all other nodes even though they are not the subject of the problem. This results in an inefficient solution of the LCA problem.

For a more efficient solution, only the input parameter nodes are focused. Ancestor links are created one by one to each ancestor of the input parameter nodes and check the lowest common ancestor after each new ancestor link creation. The necessary rules and the scheduling of these rules are depicted in Fig. 12. LinkToSelf rule creates the first set of ancestor links of the input parameter nodes, which are self-links. Therefore it is executed only once. Then, GetLCA rule tries to match a common ancestor between input parameter nodes. If GetLCA rule fails, this means computing one more level of ancestor links and executing GetLCA again. Execution of GetLCA rule successfully means finding a lowest common ancestor between input parameter nodes, so the model transformation should stop and return this node. LinkToAncestor rule creates one more level of ancestor links and is executed only once. If this rule is successful, then GetLCA rule can be executed to try for a common ancestor again. But the failure of this rule means there is no need for another check and the input model doesn't have a lowest common ancestor for the input parameter nodes.

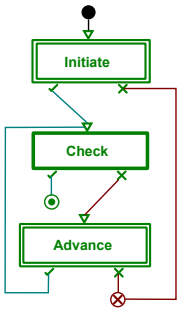

Fig. 13. Initiate-Check-Advance design pattern.

TABLE I. METRICS FOR LCA PROBLEM

Metric	Details	Sol. 1	Sol. 2
Number of rule applications	LinkToSelf	11	2
	LinkToParent	10	-
	LinkToAncestors	14	3
	GetLCA	1	4
	Total	**36**	**9**
Size of rules	LinkToSelf	3	6
	LinkToParent	7	-
	LinkToAncestors	14	22
	GetLCA	14	5
	Total	**38**	**33**
Number of auxiliary elements created	**Total**	**35**	**7**

As of this example, a design pattern can be identified from the second solution applied. First, a first step is initiated in direction of a solution. This can be a starting point, a first guess or even nothing if the problem is suitable. Then, the requested solution is queried in the intermediate form of the problem. Finally, if a solution can't be found, rules must be executed one step further and advance the intermediate form closer to find a solution. In this step, only the necessary steps that yield us a closer to the solution must be done, nothing more. The general structure of the identified design pattern is depicted in Fig. 13 as a summary.

Also, the two solutions (naive solution and more efficient solution that design pattern is applied) are compared in terms of *efficiency* quality criteria. For this reason, three metrics that are related to the *efficiency* criteria are identified: 1) Number of rule applications 2) Size of rules 3) Number of auxiliary elements created. The metrics are measured for the input model in Fig. 10 and input parameter nodes D and J. Results are depicted in Table I.

Number of rule applications metric is basically the execution number of each rule. In the first solution, LinkToSelf rule works for each node and LinkToParent rule works for each

node that has a parent. LinkToAncestors rule works again for each node that has an ancestor older than parents and for each of these ancestors. GetLCA rule works only once, since all nodes have ancestor links between each other. In the second solution, LinkToSelf rule only works for the input parameter nodes. LinkToAncestors rule works for the input parameter nodes' ancestor at each step after GetLCA rule fails to find the solution for the problem. Therefore, there are three steps until a solution is found and it means GetLCA rule works for four times and LinkToAncestors rule works for three times. In the second solution, LinkToSelf and LinkToAncestors is composite and have two rules inside which increases the number of rule applications count.

The size of rules metric counts the number of elements in the rule, such as nodes, parent links, ancestor links. The number of auxiliary elements created metric counts the number of ancestor links that are shown in purple and a diamond in the middle. The first solution generates ancestor links for all nodes even though they are not used in the solution. Therefore, it has a higher number of this metric. The second solution generates ancestor links only for the input parameter nodes through their parents and ancestors to the root.

With respect to these metrics, solution 2 has a better space and time efficiency of 55% than solution 1.

VI. Identified Challenges

In this section, the challenges in this research are listed.

A. Formalism for Model Transformation Design Patterns

One of the most important challenge is to define the most appropriate formalism for representing model transformation design patterns. In object-oriented design patterns, the community has agreed to provide design pattern solutions in UML class diagrams. Due to the few works in the literature, there is no common language or standard for model transformation area. Iacob *et al.* [38] used QVT-R to show the structure of their design patterns, whereas Agrawal *et al.* [37] introduced their own graph transformation language. The benefits of having a formalism show improvements in understanding, documenting, communicating and reasoning about the patterns in a standard way [27]. Generic higher-order transformations can be considered for describing model transformation design patterns. Tisi *et al.* [40] defines HOT as a model transformation such that its input and/or output models are themselves transformation models. They can be studied as a formalism candidate for design patterns. Syriani and Gray [27] identified two other candidates. These are: a MOF-like language, therefore defined at the metametamodel level, or a generic metamodel for model transformation described in UML.

In Section V-C, a concrete example of a model transformation design pattern is identified. In this example, graphical rule and scheduling structure of MoTif [12] is used to express the design pattern, which provides a more generic way.

B. Identifying Model Transformation Design Patterns

Identifying new model transformation design patterns is another challenge. Since the purpose of design patterns is to overcome any recurring problem to be solved, the problems in the model transformation design process have to be found. Model transformation is a rather new research field, so there is a lack of good and efficient model transformations in the literature. Therefore, more examples must be investigated.

Another way of identifying a model transformation design pattern can be what is done in Section V-C. In this section, one model transformation problem is focused and more efficient solution is generated to the same problem in different ways. Finally, a reusable design pattern is identified that is applicable to some problems starting from that one.

C. Quality Criteria and Related Metrics in Model Transformation

The quality framework of Mohagheghi and Dehlen [25] focuses on MDE, but special concern is missing for model transformation. A first step towards the identification of quality criteria for model transformation has been proposed by Syriani and Gray [27]. However this study needs to be extended by introducing metrics as done by Amstel for ATL ([29], [30]).

D. Evaluation of Model Transformation Design Patterns

Model transformation design patterns are mostly evaluated with (language independent) metrics which is not enough for a complete evaluation. Syriani and Gray [27] propose model checking techniques. This subject is need to be focused also, since identification of the model transformation design patterns must be supported by an evaluation and their advantages and disadvantages in terms of quality criteria must be listed.

VII. Conclusion

This study aims to find a starting point for creating a model transformation design pattern repository. Design patterns are important to improve quality criteria of model transformation. Therefore, model transformation languages, structure of them and model transformation intents are reviewed. Understanding the model transformation in detail helps us identify design patterns. The structure of object-oriented design patterns are also reviewed to inspire our design pattern effort from an existing and successful study. Some existing model transformation design patterns are reviewed to build a literature study. Also, a design pattern is identified by focusing on one problem and measuring quality metrics of the problem before and after the design pattern is applied.

As future work, the challenges identified in Section VI will be focused. At the end, it is aimed to have a model transformation design pattern catalog where each design pattern is precisely described, evaluated in terms of model transformation quality and ready to be applied with example problems.

References

[1] L. Rose, E. Guerra, J. de Lara, A. Etien, D. Kolovos, and R. Paige, "Genericity for model management operations," *Software and Systems Modeling*, vol. 1, pp. 1–19, 2011.

[2] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[3] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained. The Model Driven Architecture: Practice And Promise*. Addison-Wesley, 2003.

[4] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, June 2008.

[5] S. M. H. Hasheminejad and S. Jalili, "Design patterns selection: An automatic two-phase method," *Journal of Systems and Software*, vol. 85, no. 2, pp. 408–424, February 2012.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994.

[7] M. Amrani, J. Dingel, L. Lambers, L. Lucio, R. Salay, G. Selim, E. Syriani, and M. Wimmer, "Towards a Model Transformation Intent Catalog," in *MoDELS workshop on Analysis of model Transformation*. IEEE, 2012.

[8] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal, special issue on Model-Driven Software Development*, vol. 45, no. 3, pp. 621–645, July 2006.

[9] Object Management Group, *Object Constraint Language*, feb 2010.

[10] E. Syriani and H. Ergin, "Operational Semantics of UML Activity Diagram: An Application in Project Management," in *RE 2012 Workshops*. Chicago: IEEE, sep 2012.

[11] I. Kurtev, "State of the Art of QVT: A Model Transformation Language Standard," in *Applications of Graph Transformations with Industrial Relevance*, ser. Lecture Notes in Computer Science, A. Schurr, M. Nagl, and A. Zundorf, Eds. Springer Berlin / Heidelberg, 2008, vol. 5088, pp. 377–393.

[12] E. Syriani and H. Vangheluwe, "A Modular Timed Model Transformation Language," *Journal on Software and Systems Modeling*, vol. 11, pp. 1–28, June 2011.

[13] Tom Mens and Pieter Van Gorp, "A Taxonomy of Model Transformation," *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, vol. 152, no. 0, pp. 125 – 142, 2006.

[14] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations," in *Model Driven Engineering Languages and Systems*, ser. LNCS, vol. 6394. Springer, 2010, pp. 121–135.

[15] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo, "The Design of a Language for Model Transformations," *Journal on Software and Systems Modeling*, vol. 5, no. 3, pp. 261–288, September 2006.

[16] T. Klein, U. Nickel, J. Niere, and A. Zündorf, "From UML to Java And Back Again," University of Paderborn, Paderborn, Tech. Rep. tr-ri-00-216, September 1999.

[17] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214–234, 2007.

[18] G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," in *Applications of Graph Transformations with Industrial Relevance*, ser. Lecture Notes in Computer Science, J. Pfaltz, M. Nagl, and B. Bahlen, Eds. Springer Berlin / Heidelberg, 2004, vol. 3062, pp. 446–453.

[19] F. Jouault and I. Kurtev, "On the interoperability of model-to-model transformation languages," *Science of Computer Programming, Special Issue on Model Transformation*, vol. 68, no. 3, pp. 114–137, October 2007.

[20] Object Management Group, *Meta Object Facility 2.0 Query/View/Transformation Specification*, jan 2011.

[21] F. J. Barros, "The dynamic structure discrete event system specification formalism," *Trans. Soc. Comput. Simul. Int.*, vol. 13, no. 1, pp. 35–46, Mar. 1996.

[22] E. Syriani and H. Vangheluwe, "De-/Re-constructing Model Transformation Languages," in *Procedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques*, 2010.

[23] E. Syriani and H. Ergin, "Operational Semantics of UML Activity Diagram: An Application in Project Management," in *RE 2012 Workshops, IEEE, Chicago*, 2012.

[24] E. Syriani, J. Gray, and H. Vangheluwe, "Modeling a Model Transformation Language. Domain Engineering: Product Lines, Conceptual Models, and Languages," [inprint].

[25] P. Mohagheghi and V. Dehlen, "Developing a Quality Framework for Model-Driven Engineering," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, H. Giese, Ed. Springer Berlin Heidelberg, 2008, vol. 5002, pp. 275–286.

[26] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple, "Definitions and approaches to model quality in model-based software development A review of literature," *Information and Software Technology*, vol. 51, no. 12, pp. 1646 – 1669, 2009.

[27] E. Syriani and J. Gray, "Challenges for Addressing Quality Factors in Model Transformation," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, april 2012, pp. 929 –937.

[28] A. Vignaga, "Metrics for measuring ATL model transformations," *MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep*, 2009.

[29] M. F. Amstel, C. F. Lange, and M. G. Brand, "Using Metrics for Assessing the Quality of ASF+SDF Model Transformations," in *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ser. ICMT '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 239–248.

[30] M. Van Amstel, S. Bosems, I. Kurtev, and L. F. Pires, "Performance in model transformations: experiments with ATL and QVT," in *Proceedings of the 4th international conference on Theory and practice of model transformations*, ser. ICMT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 198–212.

[31] L. Kapová, T. Goldschmidt, S. Becker, and J. Henss, "Evaluating maintainability with code metrics for model-to-model transformations," in *Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice - Reality and Gaps*, ser. QoSA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 151–166.

[32] E. Insfran, J. Gonzalez Huerta, and S. Abrah ao, "Design guidelines for the development of quality-driven model transformations," in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 288–302.

[33] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 592–605.

[34] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J. Cordy, "A Tridimensional Approach for Studying the Formal Verification of Model Transformations," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, april 2012, pp. 921 –928.

[35] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 896 –909, nov. 2006.

[36] G. Krasner, S. Pope *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[37] A. Agrawal, "Reusable Idioms and Patterns in Graph Transformation Languages," in *International Workshop on Graph-Based Tools*, ser. ENTCS, vol. 127. Rome: Elsevier, March 2005, pp. 181–192.

[38] M.-E. Iacob, M. W. A. Steen, and L. Heerink, "Reusable Model Transformation Patterns," in *Proceedings of the Enterprise Distributed Object Computing Conference Workshops*. Munich: IEEE Computer Society, Setpember 2008, pp. 1–10.

[39] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On finding lowest common ancestors in trees," in *Proceedings of the fifth annual ACM symposium on Theory of computing*, ser. STOC '73. New York, NY, USA: ACM, 1973, pp. 253–265.

[40] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bezivin, "On the Use of Higher-Order Model Transformations," in *Model Driven Architecture - Foundations and Applications*, ser. Lecture Notes in Computer Science, R. Paige, A. Hartman, and A. Rensink, Eds. Springer Berlin / Heidelberg, 2009, vol. 5562, pp. 18–33.