Challenges for Addressing Quality Factors in Model Transformation

Eugene Syriani Jeff Gray

Department of Computer Science
University of Alabama
Tuscaloosa AL, U.S.A.
{esyriani,gray}@cs.ua.edu

***Abstract*—Designing a high quality model transformation is critical, because it is the pivotal mechanism in many mission applications for evolving the intellectual design described by models. This paper proposes solution ideas to assist modelers in developing high quality transformation models. We propose to initiate a design pattern movement in the context of model transformation. The resulting catalog of patterns shall satisfy quality attributes identified beforehand. Verification and validation of these patterns allow us to assess whether the cataloged design patterns are sound and complete with respect to the quality criteria. This will lead to techniques and tools that can detect bad designs and propose alternatives based on well-thought design patterns during the development or maintenance of model transformation.**

***Keywords*-model transformation; design patterns; software quality; validation and verification.**

I. Introduction

In today's practices, model transformations are deployed as software artifacts in a wide range of application contexts, ranging from prototypes in rapid development processes to large-scale industrial applications. Thus, critical requirements for model transformation are to (1) provide a sound, optimal and efficient solution to the problem to be solved, (2) demonstrate a high quality in their design and integration with other systems and technologies, and (3) hence systematically maintain the developed transformations.

The growing interest in model transformation has led to a plethora of model transformation languages expressed in different paradigms, supported by various implementations [1]. They each provide tremendous value for developers and most of them are involved in the development of large industrial applications [2]. However, despite a robust theoretical foundation based on the theory of graph transformation, this diversity lacks cohesive support for model transformation, as witnessed in [3]. Moreover, 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. We believe that, as similarly established in the object-oriented paradigm (GoF [4]), standardizing and codifying good practices in the form of *design patterns* of model transformation can solve these issues. We propose to solve these issues by (1) establishing quality criteria based on existing model transformations, (2) identify and classify well-founded model transformation design patterns with proven quality, and (3) support model transformation engineers with integrating these patterns in their design in an automated manner.

In this paper we describe the three primary research challenges to address quality factors for model transformation. Figure 1 illustrates the overall approach.

Section II proposes to first investigate and discover the commonalities across model transformation designs and partition the domain of transformations based on the results of this discovery. From these inferred commonalities, a set of well-defined representations of these components will be specified in terms of patterns. The identified patterns will be defined in an appropriate formalism to facilitate their verification and validation (V&V), as well as to support their classification.

Because our goal is to enhance the quality of transformation models, the quality criteria must be specified. Section III proposes to conduct a thorough analysis of existing model transformation languages and approaches to identify which criteria are met and which are not. This provides the initial requirements for the pattern catalog. Each pattern will then be verified against one or more of these requirements using V&V techniques applied to model transformation.

Section IV focuses on how to reduce the negative impact that model transformation development may have in complex projects. It will contribute toward a semi-automated development approach to model transformation by detecting design patterns from the catalog during the development of transformations.

II. Model Transformation Design Pattern Cataloging

The goal of this challenge is to build a repository of design patterns for model transformation development. To develop this catalog, there are two tasks that need to be achieved: identify recurrent patterns that appear when developing model transformations and define a formalism to consistently describe such patterns.

A. Pattern Identification

The task of identifying and discovering recurrent patterns in model transformation must be performed very meticu-

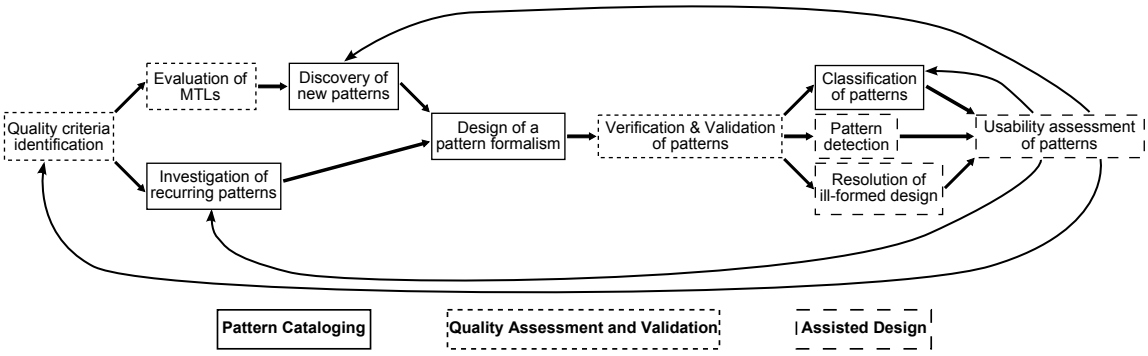

Figure 1. Overview of the proposed approach.

lously in order to cover most possible scenarios. We are aware that, just as in GoF, the collection of identified design patterns will not be complete due to the informal process of mining from existing examples as well as the relatively young field of MDE. To be as complete as possible, we propose two approaches to identify the patterns.

On the first hand, we examine a variety of existing transformations developed in industrial and academic contexts. For that, one can explore several sources such as model transformation repositories (ATL Transformation Zoo [5], ReMoDD [6]) as well as transformations developed in published case studies and examples appearing in journal and conference proceedings on modeling and model transformation (such as SoSym, MoDELS, ICMT, and ECMFA). For instance, we identify from [7] that computing the transitive closure of a hierarchical model is a recurrent task in model transformation. On the second hand, we attempt to map the well-known GoF patterns, which are defined at the code level, onto the model transformation paradigm. For example, Agrawal *et al.* [8] propose a visitor pattern for composite models. We do not expect to be able to map all 23 patterns, but examining them from a mindset geared towards their application in model transformation tasks may enlighten discoveries of new development patterns.

One shall distinguish between general-purpose and domain-specific (or problem-specific) model transformation patterns. The former would typically apply to general-purpose transformation languages (such as QVT [9]). Patterns, such as copying elements from source model to target model [10], can be applied to any model transformation in general. Domain-specific transformation languages restrict the transformation engineer to focus entirely on designing transformation models without added complexity that is irrelevant for the purpose of the transformation. Also, the transformation language has no more expressiveness than is needed as this may allow for better analysis of the transformation models. Some patterns only make sense in particular application domains and would typically apply to domain-specific transformation languages (such as transformations built in the T-Core framework [11]), for example the animation of a state-transition modeling language.

One of the main criteria of a design pattern catalog is to be language independent [12]. Hence the investigation task should covers transformations from some of the most widely used tools using different rule-based transformation techniques: declarative transformations (such as QVT-Relation and Triple Graph Grammars) and operational transformations (such as QVT-Operational Mappings, ATL and graph-based transformations). This set of tools covers the following characteristics [1], [13]: unidirectional and bidirectional, in-place and out-place, endogenous and exogenous, implicit and explicit rule scheduling, and incremental. The investigation will breadth its search in different application scenarios of model transformations: code generation from a model, synthesis and reverse engineering between models at different levels of abstraction, simulation of models by specifying the operational semantics of their language, meta-model instance generation, query and simple updates of models, model migration to adapt to evolved meta-models, model composition and synchronization.

Finally, another concern to consider when developing this catalog is to determine the granularity at which the design patterns shall be defined. For this project, we consider a transformation rule as the base level. Design patterns of individual rules of a transformation are specific to simple manipulations of model elements, *e.g.,* clone a model element. However, it is very often the case that multiple rules are needed to perform a single task. Design patterns of groups of rules then define common re-usable libraries of transformation snippets. One may also consider defining good design practices when chaining or composing multiple transformations for larger applications [14]. The latter may be automated through higher-order transformation [15].

B. Pattern Formalism

Once model transformation design patterns have been identified, it is crucial to define a formalism in which they will be defined. The benefits of having such a formalism are to facilitate understanding, documenting, communicating,

and reasoning about the patterns in a standard way. For example, GoF design patterns are described in UML class diagram. UML is independent from the object-oriented programming language used for the implementation of software. Similarly, the pattern formalism should be independent from the model transformation language in which patterns are realized. Possible candidates can be inspired from MOF-like languages (*e.g.,* Ecore) or a generic meta-model for model transformation described in UML. The advantage of the latter is that we use a domain-specific language for describing model transformations, only using the concepts related to a transformation. Ultimately, it should support higher-order transformation specification and therefore be a precisely modeled language such as in [16].

The pattern language must have a well-defined formal semantics in order to facilitate analysis (termination, confluence, property preservation). From a syntax point of view, the pattern formalism must allow model transformation patterns to be specified concisely in a canonical form.

C. Pattern Classification

The goal of the catalog is to guide model transformation designers with good practices. It is therefore important to clearly specify the context in which each pattern can be applied. For example, the Gang of Four classify object-oriented design patterns primarily according to their purpose (*i.e.,* creational, structural, and behavioral) and their scope (*i.e.,* class or object). Further, although patterns are independent from one another, they can be used in combination within a single transformation model. Thus, another way to classify patterns is via their relationships to each other. One shall investigate how to classify the design patterns according to, for example, the context, the quality attributes they are aimed to ensure, or the taxonomy category in which they fall [13].

D. Related Work

Design patterns have revolutionized the state of software practice. The most notable impact was in the object-oriented paradigm [4]. The activity of cataloging patterns has been performed for decades in many sub-fields of computer science, however very little work has been done on defining a pattern catalog in MDE and in particular in the area of model transformation (*e.g.,* [17] propose patterns for describing domain-specific languages).

Agrawal *et al.* [8] have proposed three transformation design patterns: a visitor for composite models, the computation the transitive closure in a hierarchy, and the creation of a proxy object. The former two are very relevant for the purpose of the project and will be considered in the cataloging phase. The latter has been defined in the context of a modeling a distributed system: this is an example of a domain-specific transformation pattern. However, the authors describe these patterns in terms of their tool (GReAT) and do not attempt it to abstract the tool specificity.

Iacob *et al.* [10] have proposed five transformation design patterns: a one-to-one mapping from the elements of the source model to elements of the target model, copying elements from source model to target model, the flattening of composite structures, the mapping of an element to two elements and a relation between them, and its inverse. The former three are very relevant for the purpose of the project and will be considered in the cataloging phase. The idea in the latter two can be used to produce a pattern for defining one-to-many mappings and many-to-one mappings. The authors describe their patterns in terms of QVT-Relation and do not attempt it to abstract the tool specificity.

III. Quality Assessment and Validation of Model Transformation Design

The second objective of this project is to define quality attributes for model transformation and propose a framework where transformation models are guaranteed to satisfy these criteria.

A. Quality Criteria Identification

Several catalogs of quality attributes in software engineering exist today [18]. The ISO 9126 enumerates the following quality characteristics for software: functionality, reliability, usability, maintainability, efficiency, and portability. We distinguish the following categories with respect to model transformation. Each category must be divided in specific quantifiable attributes and provide means to measure them in a reproducible way.

Correctness: the set of attributes that bear on the existence of a set of requirements and to which degree the transformation adheres to it. Correctness can be evaluated with validation and verification techniques (see Section III-B). Often, the transformation tool automatically generates traces linking the input model with the output model [19], [9]. This can then be traced transitively up to the requirements, given that they are appropriately modeled.

Re-usability: the set of attributes that bear on the ease to re-use transformation components. Re-usability can be achieved through modular composition of transformation units, rules, or complete transformations. In some approaches (*e.g.,* MoTif [20]), modularity is inherent in the transformation language itself where a transformation model consists of connecting independent building blocks. VIATRA [21] offers generic rules with parametrized types. Higher-order transformation can be used to copy parts or the whole transformation model [22].

Efficiency: the set of attributes that bear on the relationship between the performance of the execution of a transformation and the amount of resources used under stated conditions. Most model transformation tools, in particular those implementing the graph transformation paradigm,

suffer from the NP-Complete problem of pattern matching. Most efficient implementations rely on heuristics based on the input model. Caution should be taken into consideration the memory management with respect to traces and large models (consisting of over a million elements). Existing benchmarks [23], [24] may be used to measure the efficiency.

Reliability: the set of attributes that bear on the frequency and criticality of a transformation to behave in an unacceptable manner under permissible operating conditions. There are two types of reliability issues in model transformation. The pragmatic issues typically consist of error handling and security issues. Fault-tolerance techniques such as exception handling [25] address these issues. The usability issues consist of ensuring invariant properties on the transformations. For example, in a water tank simulation system, we require that the temperature of the water in a specific water tank be always between $45°$C and $90°$C. Therefore, each of the transformation rules must make sure that the temperature ranges of the water are not violated.

Maintainability: the set of attributes that bear on the effort needed to modify the transformation to satisfy new requirements or correct deficiencies. In other words, it measures the flexibility and ability to accommodate changes in a deployed transformation. One main advantage of the model-driven engineering philosophy is that models are self-documenting. Thus domain-specific transformations are readable directly by domain experts. Model and transformation evolution techniques may then be applied [26].

Interoperability: the set of attributes that bear on the cooperation between a given model transformation and other systems, such as transformation models, other software programs or technologies. Interoperability is realized through model composition. A simple solution is to conform to a standard import/export format for models, such as XMI or GXL [27]. In [28] transformations are treated as black-boxes complying to a specific interface. In [29], the composition is performed at the rule level, resulting in a new model transformation computed by transitivity.

The research challenge is to define a suite of quantitative metrics to measure these properties. A starting point can be those used in [30] and in [31].

B. Verification and Validation of Patterns

This activity verifies that the design patterns resulting from the first challenge (Section II) satisfy the set of quality criteria identified from the efforts described in Section III-A. V&V of a model transformation is the process of checking that the transformation definition meets specifications and that it fulfills its intended purpose. In this work, we primarily consider static V&V techniques: only the transformation model and the in/output meta-models are taken into account. In contrast, a dynamic approach is specific to the involved models manipulated by the transformation at run-time. The static technique is more general and poses complex challenges we plan to address as illustrated in Figure 2 and described below.

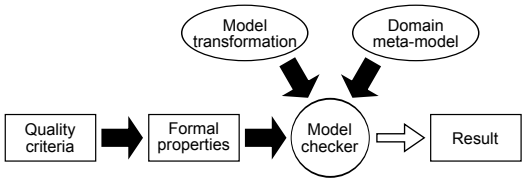

Figure 2. Verification of model transformation patterns against quality attributes.

We propose to use model checking techniques to verify properties on transformation design patterns rather than on complete transformation models. This requires to first express each quality requirement in terms of properties formally defined (*e.g.,* OCL, first-order logic). The properties must be understandable by the model checker which may require an additional transformation step. Then, the model transformation—or a part of it involving the design pattern to verify—and the involved meta-models are input to a model checker. The latter analyses them with respect to a specific property and outputs the result. The result may be a Boolean value, an instance model, or a path in the state space explored.

The choice of the language defining these properties highly depends on the model checker used. Since we are interested in analyzing transformation patterns and not a whole transformation model, the choice of the formalism used to define those patterns is also influenced by the model checking approach used. Several alternatives are to be considered. GROOVE [32] and CheckVML [33] exploit model checking analysis to prove the reachability properties on a set of rules modeled in the respective tool. In that case, graph models are interpreted as states and rule applications as transitions. For model checkers such as SPIN [34], the requirement property must be expressed in linear temporal logic (LTL). In AlPiNA [35], the properties are defined in a proprietary language that is equivalent to first-order logic with deadlock detection. he transformation model is mapped onto an algebraic Petri nets model whose behavior is represented as Kripke structures. However, AlPiNA's state space encoding does not support temporal logics as it does not encode transitions. Alloy [36] can be used to find instances and counter examples for whether a property is satisfied or not; however it relies on dynamic verification, requiring an instance model be present. Other dynamic techniques may be considered such as critical-pair analysis [37] or those based on design space exploration such as in [38].

The model checking approach does not need to be restricted to only one of the solutions enumerated. In fact, by mapping each requirement to a platform-independent language allows us to automatically generate the formal

models specific to each tool. We are aware that this may be quite complex in practice and are investigating in ways of addressing this issue.

C. Related Work

Several works have focused on defining quality attributes to models [39]. Cetinkaya *et al.* [40] have proposed a set of evaluation criteria for model transformation, but the enumeration is informal and does not suggest how to evaluate these criteria. Nevertheless these criteria are relevant and must be considered in the identification phase. Lately quality assurance of model transformation has gained interest, but is still in preliminary stage [30], [31].

The verification and validation of model transformation has become one of the main challenges in the last years. The authors in [41] propose to use SPIN in order to guarantee properties on the output model of a transformation. Several techniques map model transformations to Petri nets in order to verify model transformation [42]. The theory of Petri nets provides useful techniques to analyze transformations (*e.g.,* reachability, model-checking, boundedness and invariants) and to determine their confluence and termination given a starting model. However, these are only generic properties that are independent from the meta-models involved in the transformation. The authors in [43] suggest a manual method to analyze model transformation based on the algebraic theory of graph transformation.

IV. Assisted Design of Model Transformation

The development of complex model transformations often requires deep knowledge about the semantics of the transformation language used with respect to rule scheduling, attribute specification, and control logic. This may hinder domain experts as they are developing the transformation, which may induce project delivery delays, overspent budgets and reductions in the quality of the resulting software. The third goal of this project is to reduce the negative impact that model transformation development has in complex projects. We propose an automated development approach to model transformation by detecting design patterns based on the aforementioned catalog during the development of transformations. Also, given a model transformation design, we develop a tool to detect a non-exact match of a cataloged design pattern and propose a resolution to make it compatible with the catalog. To evaluate the impact on development productivity using this tool, we propose to conduct several controlled experiments.

A. Pattern Detection

As software products, model transformations evolve frequently. Such evolutions originate from changes in the involved meta-models or in the transformation model itself [44]. Because of their declarative expression, the evolution of a model transformation impacts its components

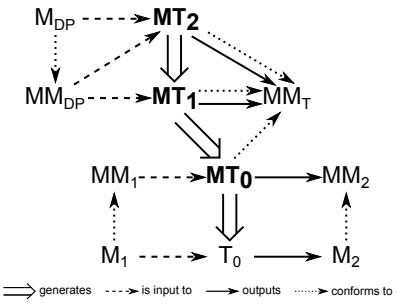

Figure 3. Transformation models generating model transformation.

(*i.e.,* rules, meta-model elements), its control logic (implicit and explicit rule scheduling), and its interaction with models (query, action, attribute specification). These hamper maintenance tasks on model transformations. Design patterns impose a structure in the transformation model due to the abstractions being used. Therefore, to better comprehend a transformation model, it would be useful to identify design patterns in the transformation directly. In this activity, we investigate techniques to automate the detection of design patterns in an existing transformation from the catalog.

Several stochastic solutions exist to this problem in the object-oriented domain [45]. We will investigate how these techniques can be applied in the domain of model transformation, based on design space exploration. Similar techniques as those implemented from the verification task (Section III-B) can be used to statically analyze the transformation and derive a structural and behavioral correspondence with a pattern from the catalog.

Another perspective is to use higher-order transformation techniques to detect the design patterns, as illustrated in Figure 3. In this scenario, the model transformation under consideration MT_0 depends on an input a meta-model MM_1 and an output a meta-model MM_2. The actual transformation T_0 is generated automatically from MT_0 to operate on models M_1 as input and M_2 as output, conforming to their meta-models MM_1 and MM_2, respectively. This technique requires that each design pattern M_{DP} be modeled in the appropriate formalism with a meta-model MM_{DP} (cf. Section II-B). The transformation under development MT_0 must conform to an explicit meta-model MM_T, as well. A first-order model transformation MT_1 can then be defined from the pattern formalism MM_{DP} to the transformation meta-model MM_T. Each rule of this transformation is read-only, consisting of a pre-condition pattern that corresponds to an individual pattern. In other words, MT_1 (the design pattern detector) finds possible occurrences of a design pattern model M_{DP} in the given transformation model MT_0. MT_0 can then be partially generated from MT_1. MT_1 can also be generated automatically , leading to a second-order transformation MT_2. In this case, MT_2 takes as input both a design pattern model M_{DP} and the design pattern

formalism MM_{DP} and outputs the transformation model MT_1. In other words, MT_2 automatically generates model transformations that detect design patterns, but tailored to a specific design pattern.

The practical outcome of this task is an enhanced IDE for model transformation that assists in detecting the use of design patterns in a transformation model under development (typically for maintenance). On the one hand, we prototype the stochastic solution in the Eclipse Modeling Framework (EMF) by developing an Eclipse plug-in to be used when developing a transformation in EMF. Both the detection of existing patterns and the suggestion of design pattern templates will be implemented in EMF. On the other hand, we prototype the higher-order transformation solution in the multi-paradigm modeling tool AToM^3 [46]. Since the IDE of AToM^3 itself is entirely modeled and transformations are also explicitly modeled with a distinct meta-model and a precise semantics, higher-order transformation can be achieved easily as in [16]. We also investigate how the second-order transformation can be implemented in ATL [19] which is integrated in EMF.

B. Resolution of Ill-Formed Design

Refactoring [47] is an important task when it comes to the maintenance of software artifacts such as model transformations. Refactoring alters the internal structure (*i.e.,* rule, scheduling, navigation, and control logic) of a model transformation without changing the domain and the purpose for which it is defined. The goal is to improve the non-functional properties of the transformation that are developed from Section III-A. This is achieved by detecting design patterns in a transformation that are almost similar to one from the catalog.

While the previous activity dealt with the automatic detection of design patterns, this activity addresses non-exact matches of designs used in a transformation under development. We focus on stochastic techniques for detecting similarities between fragments of a transformation model and a set of design patterns, for example search-based techniques [48]. This will be integrated in the EMF plug-in described above to enhance the IDE by augmenting the transformation model with state-of-the-art patterns at development-time. Once a fragment is identified, a set of matching design patterns will be proposed. Design patterns highly depend on the context in which they are used. We foresee that the patterns in the catalog we build should not be rigid, but allow for variations geared by the domain in which the transformation is applied.

C. Usability Assessment

Thus far, most empirical studies in MDE have looked at the development process, in general, compared to other development processes [2]. However, none have focused directly on the usability of model transformation and its impact on productivity. Although the verification and validation activity from Section III-B provides a formal theoretical assessment, we shall evaluate the usability of the design patterns in practice by conducting human-subject empirical studies.

In this activity, we are interested in validating the usability of the design pattern catalog. One effective way for gathering this type of information is through an observational study. In this kind of study, an experimental subject performs some task while an observer collects data about how the task is performed. Observational techniques can be used to obtain fine-grained understanding of how a new technique or tool is used. Compared to retrospective techniques such as interviews or questionnaires, collecting data by observation can be more time-consuming for the experimenter and less relaxed for the subject. However, an observational approach delivers more accurate qualitative results than retrospective methods [49]. When retrospective methods are used, subjects may find it difficult to reconstruct their own thought processes, or may (intentionally or accidentally) present their thought processes in a more structured or coherent way than actually occurred.

D. Related Work

The automatic detection of design patterns in object-oriented programming has been investigated considerably over the past decade [50], [51]. Previous approaches were focused on automatic application of design patterns through meta-programming [52] and even through model transformation [53]. However, these approaches often faced challenges related to the accuracy of the matching process. Tsantalis *et al.* [54] described an algorithm for similarity scoring between graph vertices that represent the patterns and developed system. A potential adaptation to their approach for the context of domain specific modeling languages and model transformation may be a potential area of investigation.

Some works have been done to realize automatic model completion features to create and modify the existing model elements automatically from an incomplete state to a complete state. Sen *et al.* [55] proposed to transform the meta-model and associated instance models to an Alloy specification, including static semantics. Then, the partial model can be completed automatically by applying a SAT solver. This approach provides guidance to end-users in the model editors, but the limitation is that the inferred complete models are mainly based on the input constraints, rather than end-user customizations.

Mazanek *et al.* [56] implemented an auto-completion feature for diagram editors based on graph grammars. Given an incomplete graph (model) in the editor, all possible graphs that can be generated using the grammar production rules will be suggested to users. Although this is a runtime and live suggestion feature, the suggestions are totally dependent on the grammar, which requires users to specify a number

to restrict the times of production and avoid infinite loops. Also, the graph grammar may not be fully compatible to process domain-specific modeling languages, and this approach cannot express user-customized editing activities (*e.g.,* the WCET must be greater than 300).

The concept of Model-Transformation by Demonstration allows end-users to demonstrate a model transformation within the modeling environment that is recorded and inferred for future use [57]. In this context, previously inferred model transformations are placed in a repository. From this repository, pattern matches are performed on the repository with a specific model that the user may be creating a new. This previous effort differs from the proposed work in that the prior effort was focused purely on transformation reuse of exact patterns, but the proposed work is concerned with improving the design of a model transformation by matching possible design patterns.

V. Conclusion

This paper addresses challenges of the very important problem of quality in model transformation. It proposes three main solutions: (1) the discovery and elaboration of a framework for good practices in model transformation development, (2) the formal analysis of this framework, and (3) its application in industrial settings.

The work proposed here is still in its preliminary stage. The next step is to work on the realization of each of these challenges in order to detect poor designs in models of complex systems that span a broad collection of domains. These objectives will be realized to apply well-thought patterns to frequently occurring modeling contexts, with the assistance of tools that provide feedback to a model engineer regarding the quality of the design represented in their models.

References

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

[2] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *ICSE*. Waikiki HI: ACM, May 2011, pp. 471–480.

[3] E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos, "Engineering model transformations with transML," *Software and Systems Modeling*, vol. in press, pp. 1–23, 2011.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, November 1994.

[5] "ATL Transformation Zoo," http://www.eclipse.org/m2m/atl/atlTransformations/, 2012.

[6] R. France, J. Bieman, and B. H. Cheng, "Repository for Model-Driven Development (ReMoDD)," in *Models in Software Engineering*, ser. LNCS, vol. 4364. Springer, 2007, pp. 311–317. [Online]. Available: \url{www.cs.colostate.edu/remodd/}

[7] J. Bézivin, B. Rumpe, and L. Tratt, "Model Transformation in Practice Workshop Announcement," 2005.

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

[9] Object Management Group, *Meta Object Facility 2.0 Query/View/Transformation Specification*, April 2008.

[10] M.-E. Iacob, M. W. A. Steen, and L. Heerink, "Reusable Model Transformation Patterns," in *EDOC Workshops*. Munich: IEEE Computer Society, Setpember 2008, pp. 1–10.

[11] E. Syriani and H. Vangheluwe, "De-/Re-constructing Model Transformation Languages," *ECEASST*, vol. 29, March 2010.

[12] E. Agerbo and A. Cornils, "How to preserve the benefits of Design Patterns," in *OOPSLA*, ser. ACM SIGPLAN Notices. Vancouver: ACM, October 1998, pp. 134–143.

[13] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," in *International Workshop on Graph and Model Transformation*, ser. ENTCS, vol. 152, 2006, pp. 125–142.

[14] J. von Pilgrim, B. Vanhooff, I. Schulz Gerlach, and Y. Berbers, "Constructing and Visualizing Transformation Chains," in *ECMDA-FA*, ser. LNCS. Springer, 2008, vol. 5095, pp. 17–32.

[15] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the Use of Higher-Order Model Transformations," in *ECMDA-FA*, ser. LNCS, vol. 5562. Enschede: Springer-Verlag, June 2009, pp. 18–33.

[16] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Explicit Transformation Modeling," in *MODELS Workshops*, ser. LNCS, vol. 6002. Springer, 2010, pp. 240–255.

[17] Tihamer Levendovszky and Gabor Karsai, "An Active Pattern Infrastructure for Domain-Specific Languages," *ECEASST*, vol. 25, 2010.

[18] Firebrand Architect, "Quality Attribute Scenarios Catalog," http://www.firebrandarchitect.com/quality-attribute-scenarios.html.

[19] 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.

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

[21] 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.

[22] P. Van Gorp, H. Schippers, and D. Janssens, "Copying Subgraphs Within Model Repositories," in *GT-VMT*, ser. ENTCS, vol. 211, 2008, pp. 133–145.

[23] G. Varró, A. Schürr, and D. Varró, "Benchmarking for Graph Transformation," in *IEEE Symposium on VLHCC*. Dallas: IEEE Press, September 2005, pp. 79–88.

[24] A. Zündorf, "The AntWorld Simulation Tool Case," www.se.eecs.uni-kassel.de/~fujabawiki/index.php/AntWorld, May 2008.

[25] E. Syriani, J. Kienzle, and H. Vangheluwe, "Exceptional Transformations," in *ICMT*, ser. LNCS, vol. 6142. Springer, July 2010, pp. 199–214.

[26] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*. IEEE Computer Society, 2008, pp. 222–231.

[27] M. S. Sarkar, D. Blostein, and J. R. Cordy, "GXL - A Graph Transformation Language with Scoping and Graph Parameters," in *Workshop on Theory and Application of Graph Transformations*. Springer, 1998.

[28] F. Heidenreich, J. Kopcsek, and U. Assmann, "Safe Composition of Transformation," in *ICMT*, ser. LNCS, vol. 6142. Springer, July 2010, pp. 108–122.

[29] M. Asztalos, E. Syriani, M. Wimmer, and M. Kessentini, "Towards Rule Composition," *ECEASST*, vol. 42, October 2010.

[30] P. Mohagheghi and V. Dehlen, "Developing a Quality Framework for Model-Driven Engineering," in *MoDELS Workshops*, ser. LNCS, vol. 5002. Springer, 2008, pp. 275–286.

[31] M. van Amstel, C. Lange, and M. van den Brand, "Metrics for Analyzing the Quality of Model Transformations," in *Workshop on Quantitative Approaches on Object Oriented Software Engineering*, Paphos, July 2008.

[32] A. Rensink, "The GROOVE Simulator: A Tool for State Space Generation," in *AGTIVE*, ser. LNCS, vol. 3062. Springer, 2004, pp. 479–485.

[33] Á. Schmidt and D. Varró, "CheckVML: A Tool for Model Checking Visual Modeling Languages," in *UML Conference series*, ser. LNCS, vol. 2863. Springer, 2003, pp. 92–95.

[34] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[35] S. P. Hostettler, A. A. Marechal Marin, A. Linard, M. Risoldi, and D. Buchs, "High-Level Petri Net Model Checking with AlPiNA," *Fundamenta Informaticae*, vol. 113, no. 3-4, pp. 229–264, February 2011.

[36] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[37] T. Mens, G. Taentzer, and O. Runge, "Detecting Structural Refactoring Conflicts Using Critical Pair Analysis," in *Software Evolution through Transformations: Model-based vs. Implementation-level Solutions*, ser. ENTCS, vol. 127. Elsevier, 2005, pp. 113–128.

[38] Á. Hegedüs, Á. Horváth, I. Ráth, and D. Varró, "A Model-driven Framework for Guided Design Space Exploration," in *Automated Software Engineering*. IEEE Computer, 2011.

[39] J. Rech and C. Bunse, Eds., *Model-Driven Software Development: Integrating Quality Assurance*. IGI Global, December 2009.

[40] D. Cetinkaya and A. Verbraeck, "Metamodeling and Model Transformations in Modeling and Simulation," in *Winter Simulation Conference*, Phoenix AZ, December 2011.

[41] A. Narayanan and G. Karsai, "Towards Verifying Model Transformations," *Electronic Notes in Theoretical Computuer Science*, vol. 211, pp. 191–200, April 2008.

[42] D. Varró, S. Varró Gyapay, H. Ehrig, U. Prange, and G. Taentzer, "Termination Analysis of Model Transformations by Petri Nets," in *Graph Transformations*, ser. LNCS, vol. 4178. Springer, 2006, pp. 260–274.

[43] M. Asztalos, I. Madari, and L. Lengyel, "Towards Formal Analysis of Multi-paradigm Model Transformations," *SIMULATION*, vol. 86, no. 7, pp. 429–452, July 2010.

[44] B. Meyers and H. Vangheluwe, "A Framework for Evolution of Modelling Languages," *Science of Computer Programming*, vol. 76, no. 12, pp. 1223–1246, 2011.

[45] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design Pattern Recovery in Object-Oriented Software," in *Proc. of the Int'l Wksp. on Program Comprehension*, 1998.

[46] J. de Lara and H. Vangheluwe, "AToM^3: A Tool for Multi-formalism and Meta-Modelling," in *Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 2306. Springer, 2002, pp. 174–188.

[47] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[48] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-Based Design Defects Detection by Example," in *Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 6603. Springer, 2011, pp. 401–415.

[49] F. Shull, J. C. Carver, and G. Travassos, "An Empirical Methodology for Introducing Software Processes," in *Symposium on the Foundations of Software Engineering*, 2001, pp. 288–296.

[50] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An approach for reverse engineeering of design patterns," *Software and Systems Modeling*, vol. 4, no. 1, pp. 55–70, 2005.

[51] Y. Gueheneuc, J. Guyomarc'h, and H. Sahraoui, "Improving design pattern identification: a new approach and an exploratory study," *Software Quality Journal*, vol. 18, no. 1, pp. 145–174, 2010.

[52] A. H. Eden, A. Yehudai, and J. Gil, "Precise specification and automatic application of design patterns," in *Automated Software Engineering*. Lake Tahoe: IEEE Computer Society, November 1997, pp. 143–152.

[53] J. Dong, Y. Zhao, and Y. Sun, "Design Pattern Evolutions In QVT," *Software Quality Journal*, vol. 18, no. 2, pp. 269–297, June 2010.

[54] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 869–909, November 2006.

[55] S. Sen, B. Baudry, and H. Vandheluwe, "Towards Domain-specific Model Editors with Automatic Model Completion," *SIMULATION*, vol. 86, no. 2, pp. 109–126, 2010.

[56] S. Mazanek and M. Minas, "Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors," in *MODELS*, ser. LNCS, vol. 5795. Springer, 2009, pp. 322–336.

[57] Y. Sun, J. White, and J. Gray, "Model Transformation by Demonstration," in *Model Driven Engineering Languages and Systems*, ser. LNCS, A. Schürr and B. Selic, Eds., vol. 5795. Denver: Springer, October 2009, pp. 712–726.