Towards a Model Transformation Intent Catalog

Moussa AMRANI*, Jürgen DINGEL†, Leen LAMBERS‡, Levi LÚCIO§, Rick SALAY¶, Gehan SELIM†, Eugene SYRIANI‖ and Manuel WIMMER**

* University of Luxembourg (Luxembourg), Moussa.Amrani@uni.lu
† Queen's University (Canada), {Gehan, Dingel}@cs.queensu.ca
‡ Hasso Plattner Institute / Postdam University (Germany), Leen.Lambers@hpi.uni-potsdam.de
§ McGill University (Canada), Levi@cs.mcgill.ca
¶ University of Toronto (Canada), rsalay@cs.toronto.edu
‖ University of Alabama (USA), esyriani@cs.ua.edu
** University of Malaga (Spain), mw@lcc.uma.es

***Abstract*—We report on our ongoing effort to build a catalog of model transformation intents which describes common uses of model transformations in Model-Driven Engineering (MDE) and the properties they must or may possess. We present a preliminary list of intents and common properties. One intent (transformation for analysis) is described in more detail and the description is used to identify transformations with the same intent in a case study on the use of MDE techniques for the development of control software for a power window.**

I. INTRODUCTION

While most model transformation languages are Turing-complete and, as such, can be used to solve any computable problem, most were developed to support Model-Driven Engineering (MDE). We identify a limited set of *model transformation intents* that appear repeatedly in most MDE efforts. Awareness of these intents is useful for developers of model transformations and model transformation languages. For instance, the intent of a model transformation can be to extract different views from a model (query), add or remove detail (refinement or abstraction), translate the model to another modeling language (translation), execute the model (simulation), restructure the model to improve certain quality attributes (refactoring), compose models (composition), or reconcile the information in different models (synchronization).

Each of these intents has its own set of attributes and properties. The effectiveness of a transformation in realising an intent depends on how well it respects the intent's attributes and properties. For instance, queries should produce information contained in the model in some form, translations and refactorings should preserve model semantics, and refinements should add information.

This paper is influenced by the survey conducted in [1] We report on our ongoing work to build a *model transformation intent catalog* which identifies and describes intents and the properties that they may or must possess. There are several potential uses of this catalog:

1) Requirements analysis for transformations: The catalog facilitates the description of transformation requirements, i.e., of what a transformation is supposed to do. Improved requirements in turn can improve reuse, because they may make it easier to locate suitable transformations among a set of existing ones and reuse them.
2) Identification of properties, certification methods, and languages: The catalog may help transformation developers become aware of properties a transformation has to have, how these properties can be certified, and which transformation language is known to best support their needs (*i.e.,* if the used certification methods are language dependent).
3) Model transformation language design: The catalog may provide some useful input for designers of domain-specific, transformation languages. For instance, it may be appropriate to design dedicated languages for specific intents. The properties and certification methods associated with an intent may also provide useful information about requirements of a transformation language used for an intent.

Due to the space limitation, a subset of our current intent catalog and the supporting references is demonstrated in this paper. Besides illustrating our catalog and its uses with different transformation examples from the literature, a case study on the use of transformations for the development of a car's power window software will be used. This case study shows how important transformations are for the MDE of embedded software and how diverse their intents can be. In future work, we plan to complete the catalog and use it to classify and compare model transformation analysis approaches by extending the work in [1].

The remainder of the paper is structured as follows: Section II presents a schema to describe an intent catalog, identifies common transformation intents, and lists some of their common properties; Section III details the *transformation for analysis* intent; Section IV lists some of the matching transformations in the power window case study; Section V presents related work; and finally Section VI presents the conclusion.

II. A MODEL TRANSFORMATION INTENT CATALOG

In this section we propose a schema for a model transformation intent catalog, where each intent has a set of attributes and properties, where a transformation with this intent should

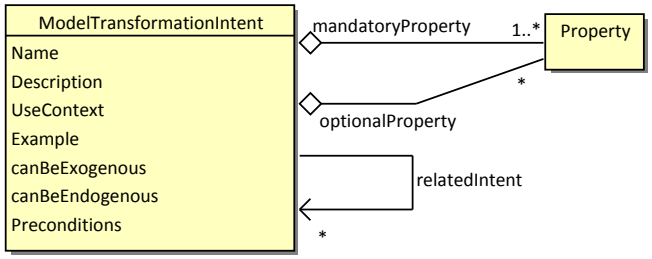

Fig. 1. The portion of the intent domain metamodel showing the key classes `ModelTransformationIntent` and `Property`.

demonstrate such properties to be able to achieve its underlying goal. We then list a set of common transformation intents and model transformation properties from the literature.

A. Transformation Intent Catalog Scheme

Our aim is to describe model transformation intents in a systematic way. Thus, we developed a schema for model transformation intents and their properties on which the intent catalog is based. Figure 1 shows a fragment of this schema. The root class in the metamodel is `ModelTransformationIntent` with attributes as shown in Table I. Adapted from the Gang Of Four [2], our definition of a transformation intent is as follows:

Definition 1. *A* model transformation intent *is a description of the goal behind the model transformation and the reason for using it.*

The class `Property` is used to specify the properties of transformations that fall under a specific intent. In this paper, we restrict our definition of a model transformation property to the following:

Definition 2. *A* model transformation property *is any verifiable characteristic inherent to a model transformation that depends on the internal details of its input/output types and/or the internal details of its implementation.*

Definition 2 is used to justify cases where it is not clear whether to use an attribute or a property to express a concept related to an intent. For example, the attribute `CanBeEndogenous` could potentially also be expressed as a transformation property called `IsEndogeneous` and then associating this as an optional property to each intent that can have endogenous transformations. In this case, the concept of being endogeneous does not satisfy Definition 2 since it is dependent only on the fact of whether the input/output types are the same or different and not on their internal details. Thus, we do not use a property to express this concept.

B. Common Transformation Intents

As defined in [3], "a model transformation is an automated manipulation of models according to a specific intent". The following list of model transformation intents extends previous work [3], [4], [5]. The proposed list is not meant to be complete, but it nevertheless covers a wide spectrum of common transformation intents. Moreover, transformation chains may combine multiple intents in separate phases.

Attributes	
Name	The name used to identify the intent.
Description	An informal description of the underlying goal of the intent.
UseContext	A description of when to use a transformation with this intent - i.e., what problems can it be used to solve?
Example	Examples of transformations that have this intent.
canBeExogenous	True iff it is possible for an exogeneous transformation to have this intent.
canBeEndogenous	True iff it is possible for an endogenous transformation to have this intent.
Preconditions	The conditions that must hold before this intent applies.
Associations	
mandatoryProperty	A property that a transformation must have in order to have this intent.
optionalProperty	A transformation property that is relevant for this intent.
relatedIntent	Another intent that is often associated with this intent.

TABLE I
ATTRIBUTES OF `ModelTransformationIntent`

1) Manipulation: Simple atomic or bulk operations on a model such as adding, removing, updating, accessing, or navigating through model elements is considered a model transformation when the system is completely and explicitly modeled.

2) Restrictive Query: It requests for some information about a model by a proper sub-model a.k.a. a *view*. We consider any subsequent aggregation or restructuring of the sub-model as an abstraction.

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) [6]. As defined in [7], a model m_1 refines another model m_2 if m_1 can answer all questions that m_2 can answer. For example, a non-deterministic finite state automaton (NFA) can be refined into a deterministic finite state automaton (DFA).

4) Abstraction: Abstraction is the inverse of refinement: if m_1 refines m_2 then m_2 is an abstraction of m_1. For example, an NFA is an abstraction of a DFA.

5) Synthesis: A model is synthesized into a well-defined language format that can be stored, such as in *serialization*. *Model-to-code generation* is the case where the target language is source code in a programming language. For example, Java code can be synthesized from a UML class diagram model.

6) Reverse engineering: Reverse engineering is the inverse of synthesis: it extracts higher level specifications from lower-level ones. For example, a UML class diagram model can be generated from Java code using Fujaba [8].

7) Approximation: As defined in [7], approximation is a refinement with respect to negated properties. That is m_1 approximates m_2 if m_1 negates the answer to all questions that m_1 negates. In practice, m_2 is an idealization of m_1 where an approximation is only extremely likely. For example, a Fast Fourier Transform is an approximation of a Fourier Transform

which is computationally very expensive.

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 that translates any of its instances to a valid instance of the reference formalism with well-defined semantics. For example, the meaning of a Causal Block Diagram is given by mapping it onto an Ordinary Differential Equation.

9) Analysis: A model transformation can be used to map a modeling language to a formalism that can be analyzed more appropriately than the original language. The target language is typically a formal language with known analysis techniques. For example, a Petri net model is transformed into a reachability graph on which liveness properties can be evaluated.

10) Simulation: A simulation is a model transformation that updates the state of the system modeled. A simulation defines the *operational semantics* of the modeling language. For example, a model transformation can simulate a Petri net model and produces a trace of the transition firing.

11) Normalization: Normalization aims to decrease the syntactic complexity of models by translating complex language constructs into more primitive constructs, which results in a canonical form of a model. For example, a Statechart model is normalized into its flattened form by removing OR- and AND-states.

12) Rendering: It is the assignment of a concrete representation to each abstract syntax elements or group of elements, as long as a meta-model of the concrete syntax is defined explicitly. Furthermore, multiple concrete syntaxes can be assigned to a single abstract syntax: for example a textual and a graphical representation of the same model.

13) Model Generation: The meta-model of a language can be defined by a graph grammar [9]. The grammar execution leads to model transformations able to generate all possible instances of the language such as in [10].

14) Migration: In [11], the authors define migration as a transformation from a software model written in one language or framework into another language, keeping the models at the same level of abstraction. When a language, *e.g.,* Enterprise Java Beans 2.0 (EJB2), *evolves* to a newer version, *e.g.,* Enterprise Java Beans 3.0 (EJB3), one must migrate all models conforming to the meta-model of EJB2 so that they conform to the new meta-model of EJB3.

15) Optimization: This model transformation aims at improving operational qualities of models such as scalability and efficiency. For example, replacing an n-ary association with a set of binary associations in a UML class diagram may optimize the subsequent code generation process.

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 [12]. For example, Zhang *et al.* [13] proposed a generic model transformation engine that can be used to specify refactorings for domain-specific models.

17) Composition: Model composition integrates models that have been produced in isolation into a compound model. Typically, each isolated model represents a concern which may overlap. On the one hand, *model merging* creates a new model such that every element from each model is present exactly once in the merged model. On the other hand, *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. It requires that changes are propagated to the models that are being integrated as done in triple graph grammars for example.

C. Common Transformation Properties

In the text that follows we identify a relevant set of model transformation properties for our purposes. Very little work exists in the literature for classifying model transformation properties. Due to that fact and due to the space limitation, the set of properties we present is tentative and incomplete. The properties we describe are based on the classification of transformation properties presented in [1] and on the literature survey conducted for this paper. We focus on transformation properties required for Section III where we investigate in detail the *model transformation analysis intent*.

1) Termination: A terminating transformation produces an output model from an input model in finite time. Termination of transformations directly refers to Turing's halting problem, which is known to be undecidable for Turing-complete, model transformation languages. Termination has been addressed extensively in several studies, including [14], [15], [16], [17];

2) Determinism: A deterministic transformation always produces the same output model for the same input model. Determinism has been addressed extensively in several studies, including [18], [15], [19];

3) Type Correctness: In a type correct transformation, the transformation's input and output models conform with their respective metamodels. Usually, *structural* conformance, involving only the metamodel, is distinguished from conformance w.r.t. additional *well-formedness rules* (*e.g.,*[20]);

4) Property preservation: A transformation can preserve the syntactic [21], [22] or semantic [23], [24] properties of a model. Since exogenous transformations are usually investigated, a formal property of the input model needs to be transformed into an equivalent property of the output model. Varro and Pataricza examine this problem in [24];

5) Traceability: Most transformation languages allow logging a transformation's traceability links between the transformation's input and output model elements. This mechanism is often used to alleviate the problem of tracing the analysis results from the transformation's input model to its output model as done in [21], [25]. In cases where the traceability of the analysis results is simple (e.g. in [26] where the termination of a transformation is decided by simulating a Petri Net that abstracts the transformation's semantics and that always runs out of tokens in finite time), less costly means may be employed.

6) Readability: A transformation is readable if it is comprehensible and amenable to be read by humans. Distinct parts in a transformation may be individually readable: the input model, the output model or the transformation specification itself. Mens mentions the readability property of model transformations in [4]. To the best of our knowledge there is little work done on the readability of software models, but metrics do exist for evaluating software readability [27]. Readability is a non-functional property of the transformation.

7) Mathematical underpinning: A transformation has a mathematical underpinning if the transformation language's semantics and/or the input and output metamodels' semantics are mathematically formalised. Mens refers to the mathematical properties of transformation languages in [4]. There is vast literature on the formalisation of programming and modeling languages, e.g. [28]. Mathematical underpinning is a non-functional property of the transformation.

III. Overview of the Analysis Intent (I_{Ana})

Due to space limitations, we demonstrate one intent, namely the *analysis intent*. First, we overview studies in the literature that fall under this intent. We then summarize the commonalities of these studies using the proposed intent scheme.

Several example transformations from the literature fall under the analysis intent. Kühne *et al.* [29] defined the semantics of Finite State Automata in terms of Petri Nets. de Lara and Taentzer [30] implemented a graph rewriting system to transform process interaction models to timed transition Petri Nets (TTPNs) for analysis. The graph rewriting system was proven to be terminating, deterministic, type correct and behaviour preserving (i.e., property preserving). Varró *et al.* [26] transformed graph rewriting systems into Petri Nets to analyze them for termination. König and Kozioura [31] proposed a tool, Augur2, that approximates graph rewriting systems as Petri Nets and analyzes them for property preservation. A property of interest is specified as a graph pattern which is transformed by Augur2 to an equivalent Petri Net marking. Accordingly, Augur2 either verifies that the property is satisfied or produces a counter example. Narayanan and Karsai [21] implemented a graph rewriting system in GREAT to transform UML activity diagrams to communicating sequential process models. The graph rewriting system was then checked for preserving structural correspondences between input and output models (property preservation). Narayanan and Karsai [23] implemented a graph rewriting system in GREAT to transform state charts to Extended Hybrid Automata (EHA) models for analysis. The graph rewriting system was then checked to preserve bisimilarity (i.e. property preservation) between input and output models. Rivera *et al.* [32] mapped graph rewriting systems to Maude and used reachability analysis and LTL model checking in Maude to analyze the graph rewriting system for property preservation. Properties were expressed as invariants, safety properties and liveness properties. Schätz *et al.* [33] formalized transformations using a relational, rule-based characterization of the transformation's constraints to analyse different possible mappings of the transformation.

Attributes	
Name	Analysis, I_{Ana}
Description	To indirectly analyse a property of the input model by running the analysis algorithm on the transformation's output model
UseContext	Need to analyse models that are not analysable in the transformation's input language, or are more efficiently analysable in the transformation's output language
Example	Transforming graph rewriting systems into Petri Nets to analyse them for termination [26]
canBeExogenous	True
canBeEndogenous	True (if transforming to a profile of the original language).
Preconditions	(1) Access to intended semantics, (2) The property of interest (that should be analysed) is defined, (3) There exists an exhaustive (up to a given abstraction) automated method to analyse the property of interest on the transformation's output language, (4) There exists a method to translate the property of interest onto the transformation's output language (if the transformation is exogenous)
Associations	
mandatoryProperty	(1) Termination, (2) Type correctness, (3) Preservation of the property of interest (specialises *Property preservation*), (4) Analysis result can be mapped back onto the input model (specialises *Traceability*)
optionalProperty	(1) Readability of the transformation's output for debugging purposes, (2) Semantics of the input language is formally defined (specialises *Mathematical underpinning*)
relatedIntent	Translational Semantics, Simulation

TABLE II
Analysis Intent, I_{Ana}

Models were represented as Prolog terms and transformation rules were represented as Prolog predicates. The state space of a transformation was formalized using a relational, rule-based description of design constraints in terms of the predicates representing the transformation rules. This formalization was then interpreted by Prolog as a non-confluent transformation.

Table II instantiates the intent metamodel (Figure 1) for the analysis intent, summarizing our findings in the literature as formerly described. Not all surveyed studies analyzed properties inherent to a transformation. Some studies analyzed properties of a transformation's output, only. For example, different design options (i.e. non-functional requirements) of the output models were explored in [33]. Such output model properties were not stated in Table II since they are not specific to a transformation per se, but only to its output models.

IV. Applying the Analysis Intent to the Power Window Case Study

The power window case study [34] is an industrially oriented study on the application of transformations to MDE of software. This study is of interest to us because it describes in terms of metamodels, model transformations and UML 2.0 activity diagrams chaining those transformations, the process of building the software for controlling an automobile's power window. A power window is basically an electrically powered window. The development of control software for such devices

is nowadays highly complex, due to the set of functionalities required for the comfort and security of the vehicle's passengers. The case study [34] is relevant to us because it exposes in a detailed fashion a large number of transformations that span many intents identified in Section II-B.

The power window case study's authors provide in their text, using varying degrees of detail, the context where their transformations occur and the properties those transformations should satisfy. We use this information as a means to validate our work. Several transformations from the power window case study apparently fall under the analysis intent. In table III we summarize these transformations according to the classification of the analysis intent in Table II. Due to space limitations we provide only very brief descriptions the transformations in the case study, which can be found in [34].

Model Transformation	**Description**	**Preconditions**	**Mand. Properties**	**Opt. Properties**
EnvToPN	Build a Petri net representation of a specialised model of the passenger's interactions with the powerwindow. Allows checking power window security requirements.	(1),(2),(3)	(1),(2),(3)	
PlantToPN	Build a Petri net representation of a specialised model of the powerwindow physical configuration. Allows checking power window security requirements.	(1),(2),(3)	(1),(2),(3)	
ScToPN	Build a Petri net representation of a specialised model of the powerwindow control software. Allows checking power window security requirements.	(1),(2),(3)	(1),(2),(3)	(1),(2)
ToBinPacking-Analysis	Build an equational algebraic representation of the dynamic behavior of the involved hardware components from an AUTOSAR [35] specification. Allows checking processor load distribution.	(1),(2),(3),(4)	(1),(2),(3),(4)	(1)
ToSchedulability-Analysis	Build an equational algebraic representation of the dynamic behavior of the involved hardware and software components from an AUTOSAR specification. Allows checking software response times.	(1),(2),(3),(4)	(1),(2),(3),(4)	(1)
ToDeployment-Simulation	Build a DEVS representation of the deployment solution to check for latency times, deadlocks and lost messages.	(1),(2),(4)	(1),(2),(3),(4)	(1)

TABLE III
MODEL TRANSFORMATION EXAMPLES FROM THE POWER WINDOW CASE STUDY THAT FALL UNDER I_{Ana}

Several interesting questions are raised by the transformations we describe in table III. First, only two of the transformations fulfill the four preconditions listed in table II. This may point to two issues: the transformation does indeed have the *analysis* intent, but has not been fully implemented; the transformation does not have the *analysis* intent. After looking at the detailed description of the transformations in table III, we found that transformations `EnvToPN`, `PlantToPN` and `ScToPN` are missing precondition (4) (property translation implementation) described in table II. Work to address that problem within the case study is foreseen. On the other hand, we found that the `ToDeploymentSimulation` transformation has the *simulation* rather than the *analysis* intent, which seems to be a good indicator of the discriminating power of table II.

Regarding the mandatory properties, the `EnvToPN`, `PlantToPN` and `ScToPN` transformations do not implement property (4) of table II. As previously, this is mainly due to the fact that the case study is not fully developed. In fact, traceability for interpreting the analysis result on the input model is yet to be implemented.

Finally, regarding the optional properties, the results in table III are to be expected. Some of the transformations do exhibit the optional properties while others do not. This indicates that our choice for such properties is indeed correct, although most likely not complete.

V. RELATED WORK

The notion of *intent* in the software engineering discipline is not new. Yu and Mylopoulos [36] realized in 1994 that current research in this area was more focused on design and implementation—the *what* and the how for developing software—rather than on the requirements necessary to understand the software to improve the underlying production processes—the *why*. To a certain extent, MDE is following the same path: historically, research was more dedicated towards the management of different modelling and transformation activities instead of exploring the intents behind them.

Three contributions [4], [5], [37] are related to our study; all aiming for a classification of different transformation aspects. Mens and Van Gorp [4] provide a multidimensional taxonomy of transformations. This taxonomy exhibits several syntactic classification dimensions with respect to the manipulated metamodels, e.g., if they are on the same abstraction level, and used transformation execution strategies, e.g., in-place and out-place transformations. While all these dimensions represent meta-information on syntactical aspects, the dimensions are illustrated on transformations related to our intents. However, our catalog aims at reflecting known, documented *uses* of transformations and proposes, in addition to the seven intents presented in [4], ten additional intents whereas one of them is characterised by its properties.

Tisi *et al.* [37] examined higher-order transformations, i.e., transformations manipulating transformations. They classify them based on whether source and/or target models are transformations or not. Our intents are more general in the sense that we do not distinguish between transformation and non-transformation models allowing for a wider applicability of the intent catalog.

The goal of Czarnecki and Helsen [5] was to classify the features of transformations languages by establishing a feature model. To do so, they introduced five intended applications of transformations which are also reflected by our transformation intent catalog.

A taxonomy of program transformations is presented by Visser [38]. Instead of proposing a taxonomy of multiple dimensions as in [4], Visser employs one discriminator for

the taxonomy: out-place vs. in-place transformations (named as *translations* and *rephrasing*). Some of the leaf nodes in the taxonomy are program-specific, *e.g.,* (*de-*)*compilation*, *inlining*, and *desugaring*. However, other nodes in the taxonomy are covered by our intent catalog. Moreover, we present several intents specifically tailored to transformations.

To sum up, the presented transformation intent catalog is more comprehensive than previous attempts. Besides providing a name and an example of each intent, comprehensive meta-information (e.g., the use context, preconditions, etc.) and properties of interest for the given intent are proposed. To the best of our knowledge, the later has not been subject of research in previous work.

VI. Conclusion and Discussion

In this paper, we have presented our ongoing work on using the notion of *intent* to help us understand the uses of model transformations in MDE and how they can be best supported. More concretely, we have listed some common transformation intents and properties, presented a schema to describe intents, and briefly illustrated its use on a case study.

Future work includes making our catalog more comprehensive and work on describing other intents has already begun. We also plan on identifying certification methods that allow a given transformation property to be analyzed, together with suitable references to corresponding research efforts on the analysis and verification of transformations.

On the more abstract level, we hope to gain a better understanding of how potential uses of the catalog outlined in the introduction can best be realized, if at all. For instance, it is currently unclear how "crisply" intents and their properties can be described and how useful our descriptions are in practice, as transformations in practice may have overlapping intents and properties that span too large a spectrum. Also, it may be more useful to think of intents as a form of "requirements patterns" [39] for transformations. To support formal transformation analysis, a formal framework for the description of properties would be useful. Finally, how intents and the certification of properties can be best supported by, possibly, dedicated transformation languages is another topic for future work.

References

[1] M. Amrani, L. Lúcio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J. R. Cordy, "A Tridimensional Approach for Studying the Formal Verification of Model Transformations," in VOLT *Workshop*, 2012.

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

[3] E. Syriani, "A Multi-Paradigm Foundation for Model Transformation Language Engineering," Ph.D. Thesis, McGill University, 2011.

[4] T. Mens, K. Czarnecki, and P. Van Gorp, "A Taxonomy Of Model Transformation," ENTCS, vol. 152, pp. 125–142, 2006.

[5] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," IBM *Systems J.*, vol. **45**(3), pp. 621–645, 2006.

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

[7] Holger Giese, Tihamer Levendovszky, and Hans Vangheluwe, "Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools," in *Models in Software Engineering*, vol. 4364, 2007.

[8] T. Fischer, J. Niere, L. Turunski, and A. Zündorf, "Story Diagrams: A New Graph Rewrite Language Based on UML and Java," in *Theory and Application of Graph Transformations*, 2000, Chapter, pp. 296–309.

[9] G. Viehstaedt and M. Minas, "DiaGen: A Generator for Diagram Editors Based on a Hypergraph Model," in *International Workshop on Next Generation Information Technologies and Systems*, 1995, pp. 155–162.

[10] J. Winkelmann, G. Taentzer, K. Ehrig, and J. Küster, "Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars," ENTCS, vol. 211, pp. 159–170, 2008.

[11] P. Mc Brien and A. Poulovassi, "Automatic Migration and Wrapping of Database Applications - A Schema Transformation Approach," in *Conceptual Modeling ER*, vol. 1782, 1999, pp. 99–114.

[12] W. G. Griswold, "Program Restructuring as an Aid to Software Maintenance," Ph.D. dissertation, University of Washington, August 1991.

[13] J. Zhang, Y. Lin, and J. Gray, "Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine," in *Research and Practice in Software Engineering (Vol. II)*, 2005, pp. 199–218.

[14] H.-K. Ehrig, G. Taentzer, J. de Lara, D. Varró, and S. Varró Gyapai, "Termination Criteria for Model Transformation," in FASE, 2005.

[15] J. M. Küster, "Definition and Validation of Model Transformations," SOSYM, vol. **5**(3), pp. 233–259, 2006.

[16] H. S. Bruggink, "Towards a Systematic Method for Proving Termination of Graph Transformation Systems," ENTCS, vol. **213**(1), 2008.

[17] F. Spoto, P. M. Hill, and E. Payet, "Path-Length Analysis of Object-Oriented Programs," in EAAI, 2006.

[18] R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems," in ICGT, 2002.

[19] L. Lambers, H. Ehrig, and F. Orejas, "Efficient Detection of Conflicts in Graph-based Model Transformation," ENTCS, vol. 152, 2006.

[20] A. Boronat, "MOMENT: A Formal Framework for MOdel manageMENT," Ph.D. dissertation, University of Valencia, 2007.

[21] A. Narayanan and G. Karsai, "Verifying Model Transformations by Structural Correspondence," ECEASST, vol. 10, 2008.

[22] L. Lúcio, B. Barroca, and V. Amaral, "A Technique for Automatic Validation of Model Transformations," in MO*DELS*, 2010, pp. 136–150.

[23] A. Narayanan and G. Karsai, "Towards Verifying Model Transformations," ENTCS, vol. 211, pp. 191–200, 2008.

[24] Dániel Varró and András Pataricza, "Automated Formal Verification of Model Transformations," in CSDUML *Workshop*, 2003, pp. 63–78.

[25] L. Lúcio, Q. Zhang, V. Sousa, and Y. Le Traon, "Verifying Access Control in Statecharts," ECEASST, 2012.

[26] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer, "Termination Analysis of Model Transformations by Petri Nets," *International Conference on Graph Transformations*, pp. 260–274, 2006.

[27] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of ISSTA '08*. NY, USA: ACM, 2008, pp. 121–130.

[28] D. Harel and B. Rumpe, "Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff," Israel, Tech. Rep., 2000.

[29] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Systematic Transformation Development," ECEASST, vol. 21, 2009.

[30] J. de Lara and G. Taentzer, "Automated Model Transformation and its Validation Using AToM3 and AGG," in *Diagrams*, 2004, pp. 182–198.

[31] B. König and V. Kozioura, "Augur 2–A New Version of a Tool for the Analysis of Graph Transformation Systems," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 211, pp. 201–210, 2008.

[32] J. Rivera, E. Guerra, J. de Lara, and A. Vallecillo, "Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude," *Software Language Engineering*, pp. 54–73, 2009.

[33] B. Schätz, F. Hölzl, and T. Lundkvist, "Design-Space Exploration Through Constraint-Based Model-Transformation," in *Engineering of Computer Based Systems Workshop (*ECBS*)*, 2010, pp. 173–182.

[34] L. Lúcio, J. Denil, and H. Vangheluwe, "An Overview of Model Transformations for a Simple Automotive Power Window," McGill University, Tech. Rep. SOCS-TR-2012.1, 2012.

[35] AUTOSAR, "Official webpage," http://www.autosar.org, 2010.

[36] E. S. Yu and J. Mylopoulos, "Understanding "Why" in Software Process Modelling, Analysis, and Design," in ICSE, 1994, pp. 159–168.

[37] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the Use of Higher-Order Model Transformations," in ECMDA-FA, 2009, pp. 18–33.

[38] Eelco Visser, "A survey of strategies in rule-based program transformation systems," *J. Symbolic Computation*, vol. **40**(1), pp. 831–873, 2005.

[39] S. Withall, *Software Requirement Patterns*. Microsoft Press, 2007.