Softw Syst Model (2016) 15:647–684
DOI 10.1007/s10270-014-0429-x

REGULAR PAPER

Model transformation intents and their properties

Levi Lúcio · Moussa Amrani · Juergen Dingel · Leen Lambers · Rick Salay · Gehan M. K. Selim · Eugene Syriani · Manuel Wimmer

Received: 24 September 2013 / Revised: 6 June 2014 / Accepted: 24 June 2014 / Published online: 18 July 2014

Abstract The notion of model transformation intent is proposed to capture the purpose of a transformation. In this paper, a framework for the description of model transformation intents is defined, which includes, for instance, a description of properties a model transformation has to satisfy to qualify as a suitable realization of an intent. Several common model transformation intents are identified, and the framework is used to describe six of them in detail. A case study from the automotive industry is used to demonstrate the usefulness of the proposed framework for identifying crucial properties of model transformations with different intents and to illustrate the wide variety of model transformation intents that an industrial model-driven software development process typically encompasses.

Keywords Model transformation · Intent · Property · Verification · Description framework

Communicated by Prof. Dragan Milicev.

L. Lúcio (✉)
McGill University, Montreal, Canada
e-mail: Levi@cs.mcgill.ca

M. Amrani
University of Luxembourg, Luxembourg, Luxembourg
e-mail: Moussa.Amrani@uni.lu

J. Dingel · G. M. K. Selim
Queen's University, Kingston, Canada
e-mail: Dingel@cs.queensu.ca

G. M. K. Selim
e-mail: Gehan@cs.queensu.ca

L. Lambers
Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
e-mail: Leen.Lambers@hpi.uni-potsdam.de

R. Salay
University of Toronto, Toronto, Canada
e-mail: rsalay@cs.toronto.edu

E. Syriani
University of Alabama, Tuscaloosa, AL, USA
e-mail: esyriani@cs.ua.edu

M. Wimmer
Vienna University of Technology, Vienna, Austria
e-mail: wimmer@big.tuwien.ac.at

1 Introduction

In model-driven engineering (MDE), *models* or software abstractions comprise the basic building blocks in the software development process, and such models are manipulated by model transformations. Thus, model transformations are considered the *heart and soul* of MDE [108] and can be used for a variety of purposes, such as the generation or synchronization of models on different levels of abstraction, the creation of different views on a system, and the automation of model evolution tasks [26].

Although several aspects of model transformations have been thoroughly investigated in the literature (such as model transformation languages and applications of model transformations), minimal research has been conducted on requirements and specifications for model transformations in general, and on the different *intents* or purposes that model transformations can typically serve in MDE and how they can be leveraged for development and validation activities.

This paper proposes the notion of *model transformation intent* to capture the purpose of a transformation and the expected goals to be achieved by using it. As illustrated in Fig. 1, intents are used to group transformations with the same goal and to associate the so-called *intent properties* with them, such as termination, type correctness, traceabil-

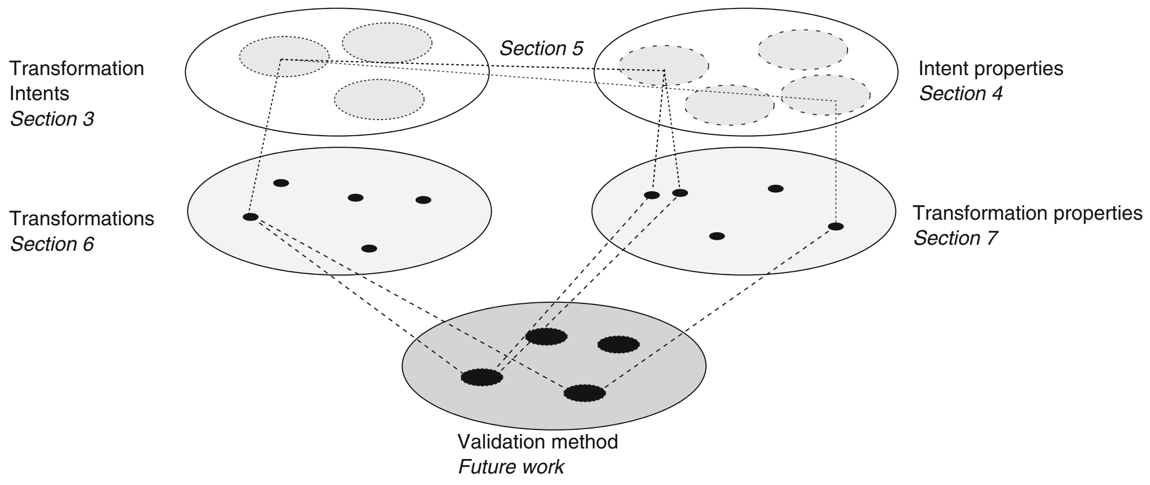


Fig. 1 Intents as a classification mechanism for model transformations

ity, or the preservation of structural or semantic aspects. An intent property can be thought of as a template that can be concretized into a *transformation property*, i.e., a concrete property pertaining to a specific transformation. The resulting link between transformations and transformation properties then facilitates validation of transformations via appropriate validation methods.

We present a description framework for model transformation intents. The framework allows the construction of a model transformation intent catalog through the identification of properties that an intent must or may possess, and any conditions that support or conflict with an intent. For instance, a translation model transformation intent can describe a model transformation whose purpose it is to prepare a model M_1 for some kind of analysis. Thus, for a model transformation to be considered a valid realization of the translation intent for analysis, it should produce an output model M_2 that, when analyzed, yields analysis results that "carry over" to M_1. High-level formalizations of key concepts in the framework are given.

The use of the framework is illustrated by presenting an initial catalog of 21 common model transformation intents and discussing six of them (query, refinement, translational semantics, translation, analysis, and simulation) in more detail. Moreover, a case study involving the use of model transformations for the development of the control software for a power window in the automotive industry is described, and for some of these transformations, their intents and transformation properties are identified.

We expect our work on model transformation intents to be useful to MDE practitioners and researchers. For instance, it would help engineers identify the model transformation intent that best matches a particular MDE development goal and facilitates the subsequent model transformation development or reuse by explicating the properties that a model transformation has to satisfy. Moreover, the notion of model transformation intent can also provide useful input for researchers interested in the specification and analysis of model transformations by clarifying how to best describe what a transformation is doing and which kinds of model transformation analyses might be most useful. Finally, the notion of model transformation intent can be used to classify model transformations into different *domains* that can be leveraged for the development of domain-specific model transformation languages and tools dedicated to express transformations of specific intents due to the language features or the kinds of analyses that they support.

This paper is a continuation of our work on model transformation verification [5] which identifies three aspects influencing the verification of model transformations (i.e., transformations, properties, and verification techniques) and uses them to survey formal verification approaches for model transformations. Transformation intents were first proposed in [4] which contains preliminary versions of the description framework and the intent catalog, together with a short description of the power window case study. This paper extends [4] significantly: We rebuild and structure the catalog, propose a formal description of the properties of intents, add a thorough description of six intents to the catalog, and exemplify the instantiation of properties for two of the thoroughly described intents.

In the next section, we will present the framework for the description of model transformation intents. An overview of the structure of the remainder of the paper will be given at the end of that section.

2 Description framework for model transformation intents

Our description framework consists of the metamodels shown in Figs. 2 and 3.

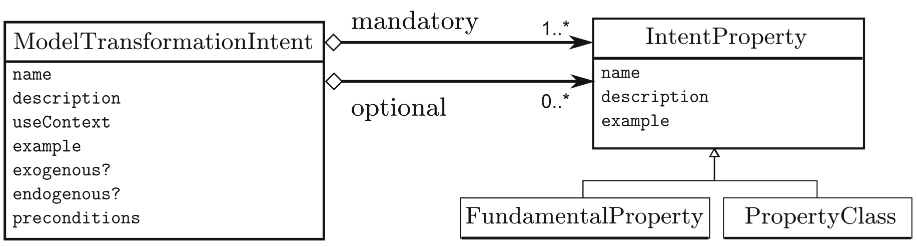


Fig. 2 Metamodel for describing model transformation intents

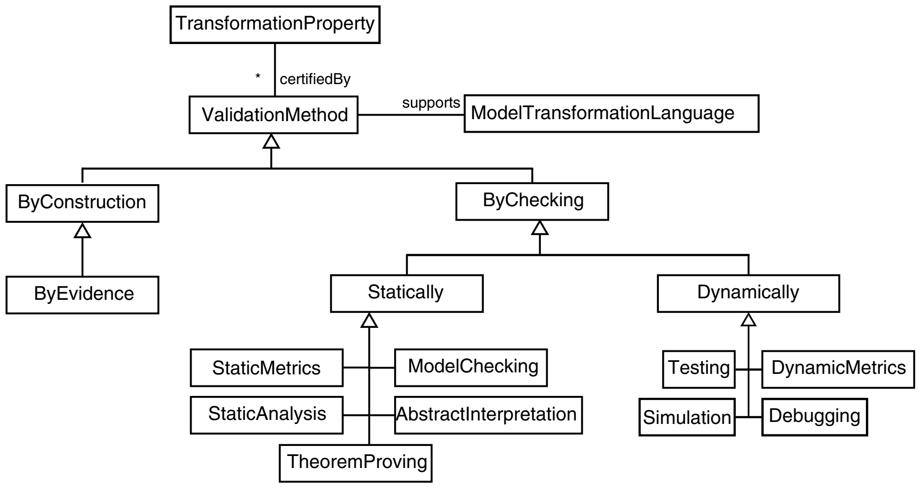


Fig. 3 Methods for validating model transformations

2.1 A metamodel for intents and their properties

In Fig. 2a, ModelTransformationIntent is described in a manner similar to object-oriented design patterns [42]. An intent has a name and is more precisely described using description and useContext. The description informally conveys the general idea behind the intent, whereas the useContext presents precise scenarios where the intent is used. One or several examples refer to sample transformations, possibly from the literature, having this intent. A set of preconditions describes any necessary conditions that need to be satisfied for transformations with this intent to be possible. Boolean attributes is_exogenous and is_endogenous indicate whether transformations with this intent can have different or the same metamodel.

An IntentProperty is a property common to all transformations with that intent. Intent properties can be seen as templates with "holes" for either the specifics of a transformation (e.g., its specification or just aspects of it, e.g., the target metamodel) or of the property to be expressed (e.g., a postcondition the output model has to satisfy). Intent properties can thus refer to aspects of the execution of the transformation, or to the result produced. The size and number of holes make some intent properties more abstract than others. Section 4 presents several intent properties including "termination," "type correctness," and "determinism" which are relatively concrete; more abstract intent properties include the "Structural Relation Property" which allows the expression of conditions over pairs of input and output models; intent property "Semantic Relational" additionally considers their semantics; properties requiring the preservation of aspects of structure or semantics arise as special cases of these two.

The mapping between ModelTransformationIntent and IntentProperty is split into two different parts: mandatory and optional properties. The mandatory property set describes *necessary* properties for a transformation to have a particular intent. Note, however, that this set is *not sufficient*, i.e., it is very common that related intents share their mandatory properties. In such cases, the intents' remaining attributes have to be consulted for disambiguation. The optional property set collects properties that transformations with a specific intent may, but do not need to, have.

2.2 A metamodel for model transformation validation methods

If the transformation is part of the development of a safety-critical application, validation[1] or even formal verification may be desired.

Partial classifications of formal verification techniques for model transformations have already been proposed in [5,20]

[1] We use the term *validation* to refer to all formal, semi-formal, and informal activities aimed at collecting evidence for the correctness of a model transformation with, e.g., testing and formal verification as prominent special cases.

where the impact of the *model transformation language paradigm* (i.e., if the model transformation language is, e.g., declarative, meta-programmed, or hybrid [26]) and the *model transformation form* (i.e., how the transformation is syntactically specified [26]) on the suitability of a given verification technique is also highlighted.

The process of filling the holes of an intent property is called *concretization* and yields a TransformationProperty, i.e., a fully fleshed out property pertaining to a specific transformation which can be used for transformation validation. In comparison with [5,20], Fig. 3 collects and organizes ValidationMethods (extracted from [1,24,32]) for validating a transformation with respect to a transformation property. We distinguish between two validation categories: ByConstruction and ByChecking. ByConstruction means that the property is implied by the way the transformation language is constructed and operates. Techniques that allow transformation-independent and input-independent validation of transformations, i.e., properties are shown to hold for all transformations of the language and for all input models, are often ByConstruction; for instance, using a mathematical proof one might be able to show termination or determinism for a model transformation expressed as a graph rewrite system for all transformations and inputs (see [5] for details). Other formal properties are either Statically or Dynamically validated with formal techniques. *Dynamic* techniques require executing the transformation being validated (e.g., Testing or DynamicMetrics), whereas *static* techniques include abstraction-based techniques such as AbstractInterpretation, TheoremProving, ModelChecking, or any StaticAnalysis with a specific scope (e.g., identifying unfireable rules). For many of these categories, concrete examples of approaches from the research literature can be found in [5].

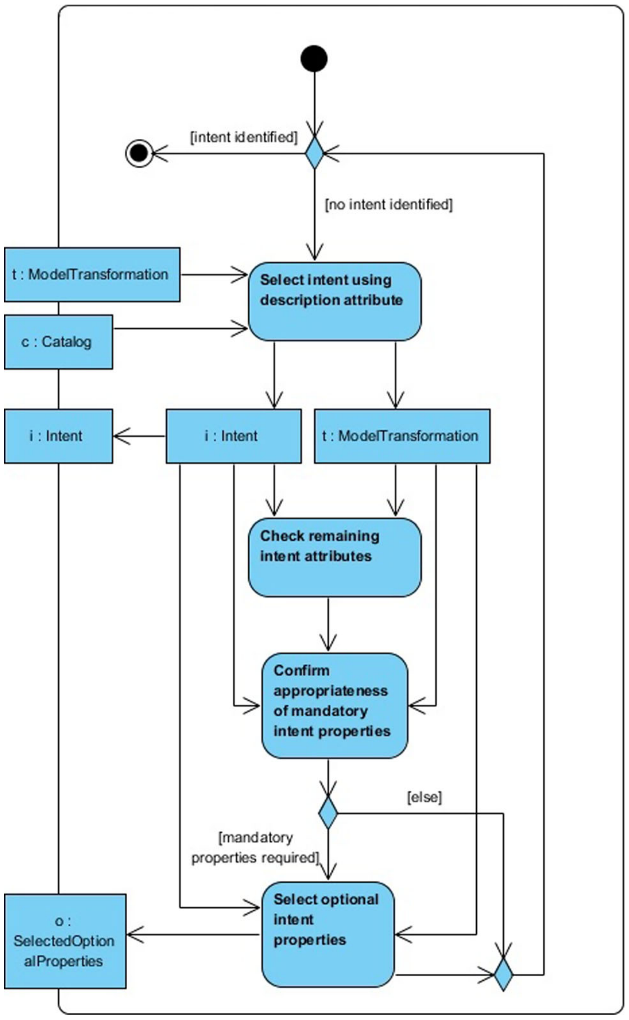

[intent identified]
[no intent identified]
t : ModelTransformation
c : Catalog
Select intent using description attribute
i : Intent
i : Intent
t : ModelTransformation
Check remaining intent attributes
Confirm appropriateness of mandatory intent properties
[else]
[mandatory properties required]
o : SelectedOptionalProperties
Select optional intent properties

Fig. 4 Identifying the intent of a model transformation

2.3 Usage scenarios

We think that our work can be of use for practitioners and researchers alike by supporting the following activities.

2.3.1 Intent identification

Given an existing transformation, our intent catalog can be used to determine the intent of that transformation together with any relevant optional intent properties as depicted in Fig. 4. Should the transformation not match any intent in the catalog sufficiently well, our framework could be used to describe the new intent and add it to the catalog. Knowing the transformation's intent may facilitate the documentation, maintenance, validation, or reuse of the transformation. If the transformation has not been implemented yet, intent identification may still be possible using, e.g., requirements documents or interviews with MDE engineers. In this case, knowing which intent the transformation is to have may facilitate implementation.

2.3.2 Model transformation validation

For validating a given transformation with respect to a specific intent, the mandatory intent properties and, to the extent appropriate, the optional intent properties need to be concretized into transformation properties pertaining to the given transformation. Validation succeeds if the transformation satisfies all transformation properties. This process is summarized in Fig. 5.

2.3.3 Model transformation research

Our work is relevant to researchers interested in the specification and analysis of model transformations, since it describes and formalizes properties that transformations may have to possess. Allowing for these, and perhaps other, properties to

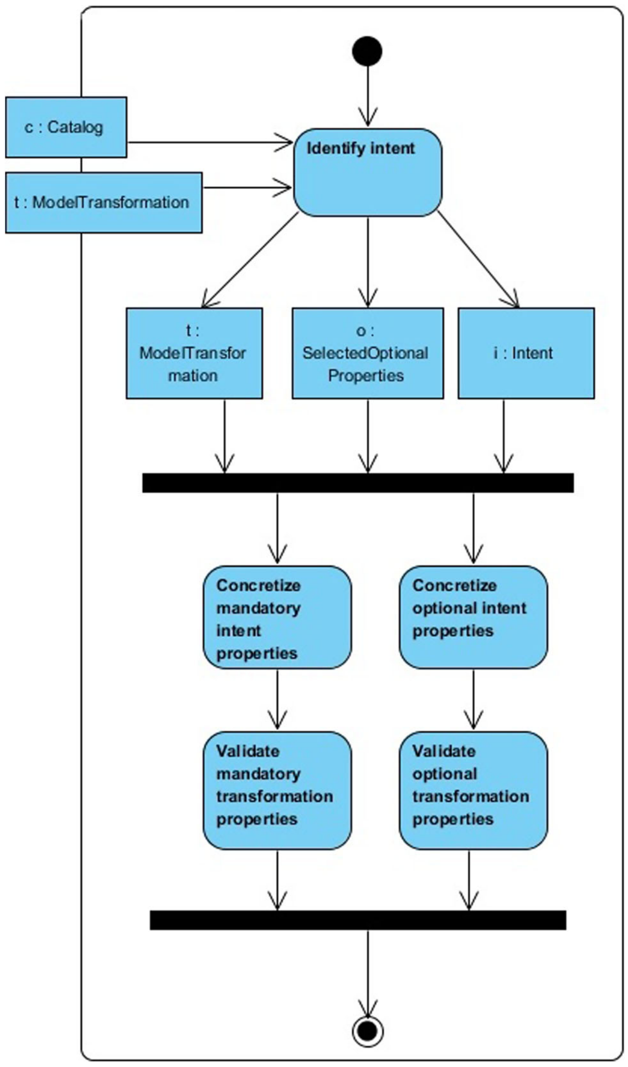

c : Catalog
t : ModelTransformation
Identify intent
t : ModelTransformation
o : SelectedOptional Properties
i : Intent
Concretize mandatory intent properties
Concretize optional intent properties
Validate mandatory transformation properties
Validate optional transformation properties

Fig. 5 Validating a model transformation with a specific intent

be expressed in a uniform, elegant specification language for model transformations would be of interest, as would be the development of effective analysis and validation techniques and tools for model transformations.

Some intents may occur so frequently and require so much development effort, that the development of an "intent-specific" (e.g., domain-specific) transformation language may be helpful. The new language may be a subset of an existing one obtained by removing certain constructs (e.g., constructs that introduce non-termination), or a completely new language employing paradigms and features that optimally support the efficient construction of transformations with a specific intent. Should these transformations be part of the development of safety-critical software, designing the transformation language in such a way that the proof of transformation properties is facilitated (e.g., a transformation language without possibly non-terminating constructs will only allow the construction of terminating transformations) could further increase productivity. Consequently, the work presented here may also stimulate more research into the design, implementation, and analysis of domain-specific model transformation languages.

2.4 Structure of the remainder of the paper

Section 3 presents a non-exhaustive catalog of 25 common transformation intents. The description of each intent is rather short, using only a small part of the framework in Sect. 2. Section 4 presents high-level formalizations of some key intent properties. The list of properties is also not meant to be exhaustive. Section 5 uses the full framework from Sect. 2 to provide detailed descriptions of the six intents: query, refinement, translational semantics, translation, analysis, and simulation. In Sect. 6, we describe the Power Window Case Study (PWCS) which shows how MDE techniques in general, and model transformations in particular, can be used for the development of software for a power window. The case study contains a transformation chain of over 30 transformations. After a detailed description of two transformations in the case study, their intents are identified and some of their intent properties are concretized into transformation properties for validation purposes (the validation itself is left for future work, though). At the end of the section, a list of the intents of all transformations in the case study is given together with their optional properties. Section 7 discusses related work. Finally, Sect. 8 summarizes the paper's contributions and presents opportunities for future work.

3 The intents catalog

Several classifications for model transformations exist in the literature. Such classifications are based on the transformation features [26], the transformation form [26], or syntactic aspects [85]. From a formal verification point of view, what really matters is the intent behind a transformation [5]: The intent conveys the transformation's actual meaning, which influences the properties of interest that need to be verified.

This section proposes an *Intent Catalog*: a description of recurring model transformation intents and illustrative examples from the literature. With respect to the metamodel in Fig. 2, this catalog informally provides the following information: name, description, and example.

Our intent catalog is not an exhaustive list of all model transformation intents, but it encompasses existing lists (e.g., [26,57,85,113,119] which are discussed in Sect. 7). An empirical evaluation of the catalog follows.

The catalog is divided in nine categories of model transformation intents as illustrated in Fig. 6. The second level of intents are concrete intents that describe a given model

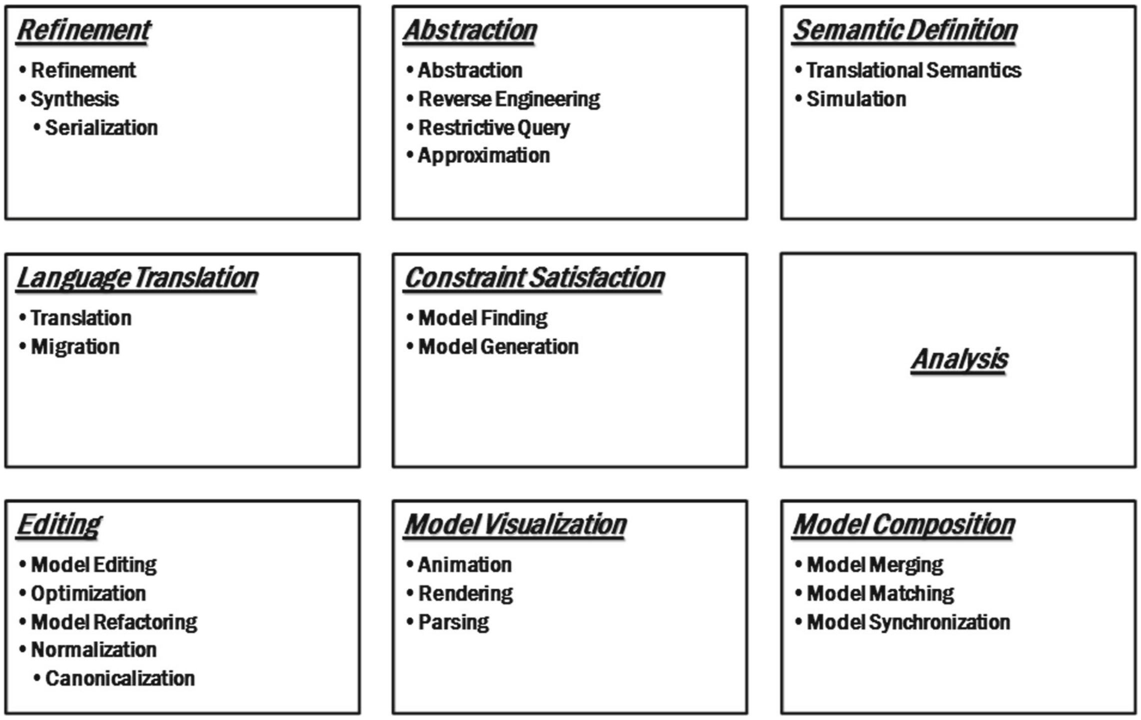


Fig. 6 Intent catalog

transformation. The third level of intents emphasizes typical special cases of concrete intents.

3.1 Refinement category

The refinement category groups intents that produce a more precise model by reducing design choices and ambiguities with respect to a target platform.

3.1.1 Refinement

A refinement transformation produces a lower-level specification (e.g., a platform-specific model) from a higher-level specification (e.g., a platform-independent model) [67], i.e., refinement adds precision to models. As defined in [46], a model m_1 refines another model m_2 if m_1 can answer all questions that m_2 can answer. Typically, m_1 contains at least the same information as m_2. For example, a non-deterministic finite state automaton (NFA) can be refined into a deterministic finite state automaton (DFA). Denil et al. [30] defined a set of refinement transformations that iteratively add platform knowledge to a deployment model.

3.1.2 Synthesis

A synthesis transformation is a refinement where the output of the transformation is an executable artifact expressed in a well-defined language format (typically textual). Synthesis is also referred to as *Model-to-code generation* [110] when the transformation produces source code in a target programming language. For example, Java code can be synthesized from a UML class diagram model. Note that the synthesis intent can be considered as a special case of the refinement intent where the output of the transformation is an executable artifact. Furthermore, a refinement transformation often precedes a synthesis transformation as demonstrated in [83,122].

Serialization A special case of synthesis where the goal of the transformation is to store the model on a medium, such as the serialization of Ecore models into XMI.

3.2 Abstraction category

The abstraction category is the inverse of the refinement category. It groups intents where some information of a model is aggregated or discarded to simplify the model and emphasize specific information.

3.2.1 Abstraction

Abstraction is the inverse of refinement: If m_1 refines m_2, then m_2 is an abstraction of m_1. Typically, m_2 will hide some information while revealing other information. For example, an NFA is an abstraction of a DFA. Also, Mannadiar and Vangheluwe [83] used a transformation to extract user-interface behavior from a Statecharts model into a

PhoneApps model. An view of a model that is not a submodel, but an aggregation of some of its information, is also a abstraction. For example, "retrieve all cycles in a Causal Block Diagram model" outputs a view of the causal block diagram model represented as a cyclic graph composed of strongly connected components.

3.2.2 Restrictive query

A query transformation requests some information about a model in the form of a proper submodel or a *view*. Restrictive query is a special case of abstraction where the result of a query is a submodel of the input model. EMF INC-Query [10] is a model transformation language that is used specifically for querying EMF models. For example, the query "get all the leaves of a tree" is a restrictive query. We consider any subsequent aggregation or restructuring of the resultant submodel or view as an abstraction.

3.2.3 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 [38]. Reverse engineering is considered as a special case of abstraction where the input model is code.

3.2.4 Approximation

We consider transformation m_1 is an approximation of m_2 when m_1 is equivalent to m_2 up to a certain error margin. Naturally, m_1 preserves more properties of m_2 as the error decreases. The error margin is typically based on a distance measure between models. For example, a fast Fourier transform is an approximation of a Fourier transform, which is computationally very expensive.

3.3 Semantic definition category

The semantic definition category groups transformation intents whose purpose is to define the semantics of a modeling language.

3.3.1 Translational semantics

A translational semantics transformation gives the meaning of a model in a source language in terms of the concepts of another target language. It is typically used to capture the semantics of new DSLs: As in [55], the semantic mapping transformation defines the mappings from the abstract syntax of the DSL into a semantic domain with well-known semantics. For example, Causal Block Diagram's semantics are expressed as ordinary differential equations.

3.3.2 Simulation

A simulation transformation defines the *operational semantics* of a modeling language that updates the modeled system's states. The output model of the transformation is then an "updated version" of the input model (i.e., the transformation is in-place). Simulation updates the abstract syntax of the model, which may trigger modifications in the concrete syntax. One example is in [72], where a model transformation was used to simulate a Petri Net model and produced a trace of the transitions firing.

3.4 Language translation category

The language translation category groups transformation intents that define a translation between two modeling languages.

3.4.1 Translation

A model translation transformation maps the concepts of a model in a source language to the concepts of another target language while translating the semantics of the former in terms of the other. A typical translation transformation is the class diagram to the relational database schema case study [12]. The resulting model can then be used to achieve several tasks that are difficult, if not impossible, to perform on the originals. For example, Syriani and Ergin [114] transformed a UML activity diagram into a Petri Net model in order to detect deadlocks and starvation, i.e., analysis is *delegated* to the Petri Net workspace.

3.4.2 Migration

A migration transformation is such that it transforms software models written in one language (or framework) into software models conforming to another language (or a modified version of it), while keeping the models at the same abstraction level [15]. Migration can be thought of the consequence of evolving a model language to a newer version. For example, transforming Enterprise Java Beans 2.0 (EJB2) class diagrams so that the resulting models conform to EJB3 can be achieved by a migration transformation as in [6]. The process of migrating each model individually so that they conform to the evolved metamodel can be automated through model transformations as presented in [22].

3.5 Constraint satisfaction category

The constraint satisfaction category groups transformation intents that output models given a set of constraint to satisfy.

3.5.1 Model generation

Model generation is a transformation that automatically produces possible (correct) instances of a metamodel, such as in [132]. The metamodel of a language can be defined using a grammar, e.g., expressed in the Extended Backus-Naur Form (EBNF), or a graph grammar [128] which, in a sense, encode the constraints that the instances need to satisfy. Such model transformations are very useful for testing model transformations since it facilitates the automatic generation of input test models to verify the correctness of a transformation [27].

3.5.2 Model finding

Adapted from [120], model finding is a transformation that searches for models that satisfy given constraints. In that case, several models are generated according to a set of rules and evaluated to check whether the generated models satisfy some constraints. If not, a backtracking mechanism reverses some of the applied rules to find another model. A typical use of this intent is in *design-space exploration* (e.g., [104]) which supports decision-making when several solutions exist.

3.6 Analysis

The analysis intent is a category on its own that encompasses all analysis techniques that are too long to enumerate here. An analysis transformation implements analysis algorithms of varying complexities, from detecting dead code or unapplicable rules to model-checking temporal formulae over appropriate structures described by models. For example, Lúcio and Vangheluwe [81] implemented a symbolic model-checker for the DslTrans transformation language using model transformations.

3.7 Editing category

The editing category groups transformation intents that manipulate a model directly.

3.7.1 Model editing

The simplest operations on a model are *adding* an element to the model, *removing* an element from the model, *updating* an element's properties, *navigating* through the elements, and *accessing* the properties of an element. These primitive operations are also known as the *CRUD* operations as first introduced by Kilov [66]. These simple operations are considered as a model transformation when the system is completely and explicitly modeled, such as in AToMPM [118].

3.7.2 Optimization

Optimization is a special kind of model edition that aims at improving operational qualities of models, e.g., scalability and efficiency. For example, replacing n-ary associations with binary associations in a UML class diagram can optimize the code generated from the class diagram [45].

3.7.3 Model refactoring

Model refactoring is a special kind of model edition where the model is restructured to improve certain internal quality characteristics without changing the model's observable behavior [40,48]. Zhang et al. [135] proposed a generic model transformation engine that can be used to specify refactorings for domain-specific models.

3.7.4 Normalization

Normalization is a special kind of model edition that aims at decreasing the syntactic complexity of models by translating complex language constructs of an input model into more primitive constructs. For example, Agrawal et al. [2] normalized a Statechart model into its flattened form, replacing OR and AND states by the appropriate states and transitions.

Canonicalization A special case of normalization where the representation of a model is normalized in a unique form. This is typically useful when verifying the equality of two models. For example, the work in [102] discusses how to compute normal forms of equation expressions using model transformation with Maude.

3.8 Model visualization category

The model visualization category groups transformation intents that deal with the relation between the abstract and concrete syntax of a modeling a language.

3.8.1 Animation

Animation is the visualization of a simulation. It projects the behavior of a model on a specific animation view. In contrast with a simulation transformation, an animation transformation operates on the concrete syntax (or the abstract syntax of the concrete syntax) of a model. For example, Ermel and Ehrig [36] used a model transformation to define the mapping from simulation steps to animation steps of a radio clock.

3.8.2 Rendering

A rendering transformation assigns one (or more) concrete representation(s) to each abstract syntax element or group of elements in an input model, as long as the metamodel of the concrete syntax is defined explicitly. For example, Guerra and de Lara [54] used event-driven grammars to relate the abstract and concrete syntaxes of visual languages.

3.8.3 Parsing

Parsing is the inverse of rendering: It maps the concrete syntax of a modeling language back to its abstract syntax. This is implemented by a model transformation involving the metamodel of the concrete syntax and the meta-metamodel of the language. For example, a model written in the Textual Concrete Syntax (TCS) [59] is transformed into a KM3 model of its abstract syntax.

3.9 Model composition category

The model composition category groups transformation intents that integrate models produced in isolation into a compound model, where each isolated model represents a concern that may overlap with any of the other isolated models.

3.9.1 Model merging

A particular instance of composition is *model merging*. In this case, the composition creates a new model such that every element from the union of both models is present exactly once in the merged model. Engel et al. [34] proposed a transformation language that allows one to compute the merged model from two models conforming to the same metamodel.

3.9.2 Model matching

A model matching transformation creates correspondence links between corresponding entities. This is also known as *model weaving*. Del Fabro and Valduriez [37] defined a generic metamodel to capture correspondences between models.

3.9.3 Model synchronization

Model synchronization integrates models that have evolved in isolation and that are subject to global consistency constraints by propagating changes to the integrated models. Such transformations are typically used when multiple views of a common repository model are accessed or modified as in [53].

3.10 Empirical evaluation of the intent catalog

We first started presenting a preliminary version at several workshops with various audiences (CAMPaM'11'12, AMT'12, AOM'13) in order to receive feedback from the community. Once the catalog reached a fixed point, we proceeded with a succession of iterative empirical studies conducted over several months. In the following, we report on the final study that led to the intent catalog presented in this paper.

3.10.1 Objectives

The goal of this study was to evaluate the correctness, unambiguity, and completeness of the intent catalog. There are two levels of correctness that we wanted to measure: (Q1) "Up to what degree do people agree with what we (the authors of the catalog) expected?" and (Q2) "Up to what degree do people agree with each other independently from the expected answers?" The unambiguity objective can be formulated as (Q3) "How difficult is it to distinguish between two or more intents to characterize one model transformation?" Finally, the completeness objective can be formulated as (Q4) "Is there any intent of an existing transformation that does not fall under an intent of the catalog?"

3.10.2 Methodology

Formally validating the intent catalog is intractable because of the informality in which it is defined. Therefore, we opted to empirically validate the catalog with respect to the four objectives defined previously. We prepared an online survey where we asked 26 questions: 25 randomized multiple choice and one free form. Each question of the first 25 described a model transformation example in one sentence in English and stated explicitly the input and output metamodels involved. For example, "Map a custom DSML for wrist watches to a Statechart model in order to define its behavior. Input: Watch DSML Output: Statechart." The participants had to drag and drop one intent from the list of intents provided to them in alphabetical order at each question in the intent box. If they were doubting between two intents and they could not decide, they were allowed to drop the least likely one in the alternate box.

The last question was an open question asking them to optionally answer the objective question Q4 or any other comment they would have.

For their training, each participant was given the intent catalog and the instructions. They were not allowed to ask any question regarding the catalog or the questionnaire, since we were evaluating the unambiguity of the catalog. However, they were allowed to seek resources from the Web if they were not familiar with concepts in a question. There was no time limit for the experiment. The participants took on average 42 min to perform the experiment.

3.10.3 Inclusion and exclusion criteria

A participant was eligible for the experiment if he had implemented at least one complete model transformation in the past. In total, we surveyed 38 participants with very different backgrounds and expertise. We did not distinguish between participant profiles because the catalog is intended to be used by anyone who wants to develop or analyze a model transformation. Among the participants, there were masters and doctoral students, researchers, and professors in computer science, electrical, and mechanical engineering. For the last iteration of this experiment, which we discuss next, we surveyed 14 participants.

3.10.4 Results and discussions

To quantitatively measure Q1, every question of each participant was assigned a score of 1 if the answer in the intent or alternate intent box matched our expected answer and 0 otherwise. Figure 7 shows how many participants agree with our answers on how many questions based on the scores. For example, 93 % of the participants found the same intent as we expected on 28 % of the questions. Therefore, we can conclude that participants agreed with our answers on average 73 % of the time. Furthermore, the scores recorded varied between 56 and 88 %, which reflects that all participants agreed with our expected answers more often than they did not. This is a very satisfactory result for Q1 to measure the correctness of the catalog given the heterogeneity of the participants and the subjectivity of the experiment, and that they could not ask for any clarification.

For Q2, we used the statistical measure, called Fleiss' kappa [39], to assess the inter-rater reliability of agreement. The values of κ range between 0 (no agreement) and 1 (perfect agreement) indicating how much multiple judges agree with their decisions. This measure is therefore unbiased with our expected answers since every answer given by a participant is taken into account equally. For the 14 participants, $\kappa = 0.57$ that indicates a "moderate agreement." However, as depicted in Fig. 8, we observe a quasi-linear correlation between κ and the score. This means that the more participants agreed with our expected answers, the more they were in agreement. This indicates that our expected answers were correct according to the participants. Scores of at most 68 % are in "moderate agreement." Some answers provided by these participants were dramatically different and incorrect, e.g., when expecting translation, model matching was given or when expecting synthesis, restrictive query was given. Therefore, if we partition the participants into one group with all scores of at least 68 % (10 participants) and another group with remaining (4 participants), than the former group is in "substantial agreement" ($\kappa = 0.64$), whereas the latter one is in "moderate agreement" ($\kappa = 0.51$). We may therefore consider the latter group as outliers.

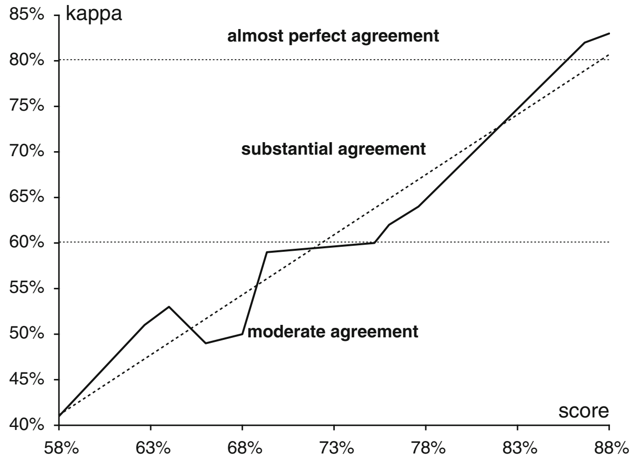


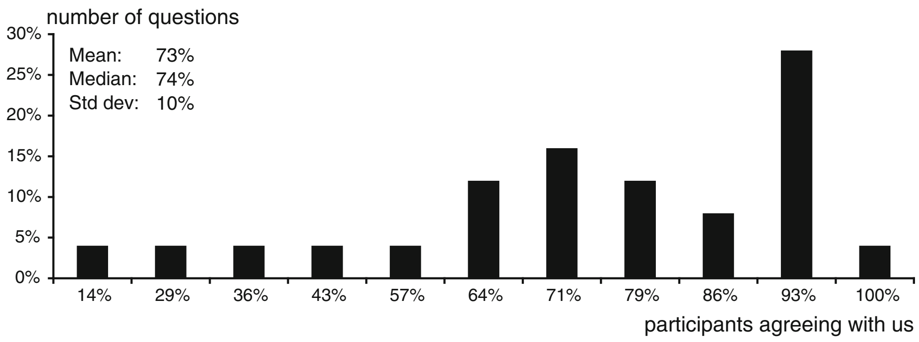
Fig. 8 The correlation between κ and the score

For Q3, the alternate intent box was used only 7.7 % of all the questions among all participants (350 answers). Therefore, on average, a participant could not decide between two intents only once. This low ratio reflects the low level of ambiguity of the catalog.

For Q4, none of the participants was able to suggest additional intents that they thought are missing in the catalog.

In order to improve the catalog, we extracted which questions were problematic. We considered a question to be problematic if at most 50 % of the top 10 participants agreed with the correct answer or if at least two of them agreed on an

Fig. 7 The ratio of participants agreeing with respect to the number of questions



incorrect answer. In this experiment, we were able to reduce to six problematic questions: those dealing with translation, synthesis, optimization, rendering, approximation, and parsing. The variations between the answers ranged from three to five different answers for each question. Nevertheless, there was only "slight agreement" ($\kappa = 0.19$) on the problematic questions and "almost perfect agreement" ($\kappa = 0.81$) on all remaining questions.

3.10.5 Threats to validity

The first threat to the validity of this study is in the participants themselves. The number of participants and their arbitrary selection may have had an influence on the results. Furthermore, the survey was anonymous: We did not distinguish between beginners, novices, and experts in model transformation. Through the various experiments we conducted, several participants were not familiar with some of the concepts involved in the questions. Distinguishing between these groups will give stronger insights into how to formulate the questions and possibly the catalog depending on the profile of the reader and suggests the appropriate background needed to use the catalog. Also, although instructed to do so, the participants may not have always totally relied on the description of each intent, but instead relied on their familiarity with the intent name which has different meanings in different domains.

A second threat is the possible ambiguity of the questions. Although we assessed the ambiguity of the catalog, the participants may have found the questions ambiguous, leading to different answers than those expected. One remedy is to provide additional information per question, such as complete input and output models or the complete transformation.

We see a few threats in the statistical measures used. For example, Fleiss' kappa gives equal weight to all answers. A weighted kappa could have been used by giving higher weights to intents that fall in the same category as the expected answer, for instance.

Finally, we believe that providing the participants with the full characteristics of each intent, as described in Sect. 5 for six of them, would have mitigated the ambiguities they faced. However, this would require them to spend several hours to get familiar with the content. We therefore interpret the results of this empirical evaluation as a motivation to further formalize the intent catalog as described in the following sections. As future work, a more extensive evaluation is planned once this formalization of the catalog is complete. Such an extensive evaluation might then clarify if ambiguities that occurred in this experimental evaluation could indeed be avoided. At the stage of using the intent catalog for the purpose of verifying model transformations, this would be very important, since preciseness then plays an indispensable role.

4 Formalization of intent properties

Figure 9 depicts the general ideal process of model transformation. An input model, conforming to a source metamodel, is transformed into an output model, itself conforming to a target metamodel, by executing a transformation specification that conforms to its transformation language. A transformation specification is defined in terms of the source and target metamodels, whereas its execution operates on the model level. Both source and target metamodels, as well as the transformation specification, are themselves models, conforming to their respective metamodels: For metamodels, this is the classical notion of meta-metamodel; for transformations, it actually refers to the transformation language, which allows a sound transformation specification. Of course, some transformations manipulate several input and/or output models.

This section provides a minimal mathematical framework sufficient for our purpose: It allows to formally define the mandatory and optional properties of our Description Framework (Sect. 2) that are detailed in Sect. 5, paving the way toward transformation validation, and later serves to illustrate the transformations extracted from the Pwcs. This framework is obviously not exhaustive: It is tailored for the set of intents covered by this paper. Describing the other intents not yet formally described by the Description Framework will necessitate different property classes whose formalisation can elaborate on the current mathematical framework's state.

Section 4.1 provides notations for the general notions (metamodels and models, model transformations, and model semantics) on which the properties used for describing the transformation intents of Sect. 4.2 are based. The readers not

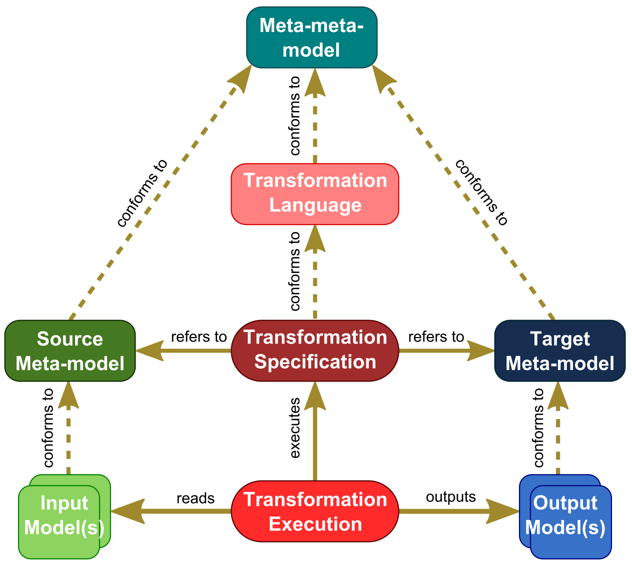


Fig. 9 Model transformation: the big picture (adapted from [115])

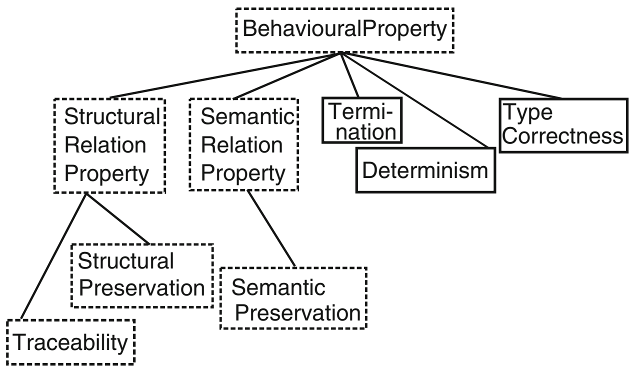


Fig. 10 Intent property hierarchy

particularly interested in the mathematical content can safely skip this section and only refer to Fig. 10 to retrieve the correspondence between mandatory/optional properties and their formal counterpart.

4.1 Metamodels & models, model transformation, and model semantics

Figure 9 depicts models, metamodels, and the conformance relationship between them. The following definition introduces notation for these notions.

Definition 1 *((Meta-)models—Conformance)* Let $\mathcal{M}$ and $\mathbb{M}$ be the sets of all metamodels and models respectively, as defined by a meta-metamodel. For a given model $\mathsf{M} \in \mathbb{M}$ and a given metamodel $\mathsf{MM} \in \mathcal{M}$, we write $\mathsf{M} \lhd \mathsf{MM}$ if M *conforms to* MM. We denote by $\mathcal{L}(\mathsf{MM})$ the set of all models $\mathsf{M} \in \mathbb{M}$ conforming to MM, i.e. all models $\mathsf{M} \in \mathbb{M}$ such that $\mathsf{M} \lhd \mathsf{MM}$.

Historically, one of the first definitions for model transformation was proposed by the OMG, in line with the model-driven architecture view. The OMG perceives transformations as "the process of converting one model to another model of the same system" [49]. This system-centric view was enlarged by Kleppe et al.: "a model transformation is the automatic generation of a target model from a source model, according to a transformation definition" [67]. This definition brings a change from the system-centric view and considers general input/output models, while insisting on the fact that transformations are mostly perceived as *directed* and *automatic* (i.e., without users' intervention) manipulation of models. Similarly, Tratt describes a transformation as "a program that mutates one model into another" [121], emphasizing the computational aspect of transformations. More recently, two contributions have widened the perspective with two important aspects: Mens et al. proposed to view transformations as "*the automatic generation of one or* multiple *target models from one or* multiple *source models, according to a transformation description*" [85], whereas Syriani re-introduced the crucial importance of the intent behind transformations: "the automatic manipulation of a model with a specific intention" [113].

We propose a broader definition that clearly embeds the dual nature of model transformation, distinguishing its specification from its execution, and places the transformation's intent at its core.

Definition 2 *(Informal Definition adapted from* [5]*)* A *transformation* is the **automatic** processing of **input models** to produce **output models**, that conforms to a **specification** and has a **specific intent**.

In this definition, note that the input and output models may be the same artifact in the case where a transformation is in-place. As with any computational artifact, a transformation operates at two levels: the *specification*, which is defined by the transformation designer, refers to source/target metamodels; and the *execution*, performed by the transformation engine, following a specific *semantics*.

Definition 3 *(Model Transformation Specification)* A transformation specification t is a triple

$$t = ((\mathsf{MM}_\mathsf{s}^\mathsf{k})_{\mathsf{k}\in[1..n]}, (\mathsf{MM}_\mathsf{t}^\mathsf{k})_{\mathsf{k}\in[1..m]}, \mathsf{spec})$$

where $(\mathsf{MM}_\mathsf{s}^\mathsf{k})_{\mathsf{k}\in[1..n]}$ and $(\mathsf{MM}_\mathsf{t}^\mathsf{k})_{\mathsf{k}\in[1..m]}$ are indexed sets of source and target metamodels, respectively, and $\mathsf{spec} \in \mathcal{L}$ is a well-formed transformation specification written in a transformation language $\mathcal{L}$.

Transformation specifications have a dual nature, as noticed by Bézivin et al. [11]. As a *model transformation*, it emphasizes a particular manipulation of source and target metamodels that spec describes precisely—this corresponds to, in Fig. 9, the horizontal links named *refers to* from *Transformation Specification*. As a *transformation model*, it emphasizes the linguistic nature of spec, i.e., the conformance relationship between spec and its language definition $\mathcal{L}$, and subsequently, its execution—this corresponds to, in Fig. 9, the vertical links from *Transformation Specification* to its *Transformation Language* and *Transformation Execution*. Moreover, the form of spec depends on $\mathcal{L}$'s underlying paradigm (either operational or declarative, or both, i.e., hybrid—see [26]) and manipulates directly the concepts defined by the source and target metamodels.

This paper generally considers intents involving one input model and one output model. Without loss of generality, the rest of our definitions for transformations is therefore focused toward this particular case: We consider $n = m = 1$ and simply write a specification $t = (\mathsf{MM}_\mathsf{s}, \mathsf{MM}_\mathsf{t}, \mathsf{spec})$.

Due to its computational nature, a transformation execution can be represented by a transition system, whose execution provides the semantics for the transformation specification.

Definition 4 (*Model Transformation Execution*) The *execution*, or *semantics*, of a model transformation specification t is given by a transition system $\mathsf{TS}_\mathsf{t} = (\mathsf{S}, \mathsf{I}, \longrightarrow)$ where S is a set of *execution states* that is a subset of $\mathbb{M}$; $\mathsf{I} \subseteq \mathsf{S}$ is a set of *initial* states, called *input models*; and $\longrightarrow \subseteq \mathsf{S} \times \mathsf{S}$ is the transition relation over S.

A transformation *execution* TS_t is linked to its *specification* $t = (\mathsf{MM}_\mathsf{s}, \mathsf{MM}_\mathsf{t}, \mathsf{spec})$ through the subscript notation (avoided when clear from context). The precise definitions of S and $\longrightarrow$ strongly depend on the specification language $\mathcal{L}$.

Note that we will only consider in this paper transformations with *legal* input models, i.e., transformations executing or starting, from a conforming input model: $\forall \mathsf{M} \in \mathsf{I}, \mathsf{M} \in \mathcal{L}(\mathsf{MM}_\mathsf{s})$.

From the perspective of formal language theory, what a model designer defines with a metamodel is a language's *abstract syntax*, i.e., the designer captures the relevant concepts and their relationships in a way that enables their internal representation for further computations. To allow their manipulation by modelers, a metamodel must be accompanied with one or several *concrete syntaxes* that define their concrete representation, be it graphical or textual (or both). As a last ingredient, the *semantics* is necessary to perform manipulations of models according to the meaning attached to the modeled concepts.

Definition 5 (*Model Semantics* [55]) Let $\mathsf{MM} \in \mathcal{M}$ be a metamodel. The semantics of MM, denoted $[\![\mathsf{MM}]\!]$, is a pair $[\![\mathsf{MM}]\!] = (\mathbb{D}_\mathsf{MM}, \mu_\mathsf{MM})$, where $\mathbb{D}_\mathsf{MM}$ is called the *semantic domain* and μ_MM the *semantic mapping* given as follows:

$$\begin{aligned} \mu_\mathsf{MM} : \mathcal{L}(\mathsf{MM}) &\longrightarrow \mathbb{D}_\mathsf{MM} \\ \mathsf{M} &\mapsto \mu_\mathsf{MM}(\mathsf{M}) \end{aligned}$$

The precise definition of $\mathbb{D}_\mathsf{MM}$ and μ_MM (noted without subscripts when clear from context) highly depends on the nature of the models in $\mathcal{L}(\mathsf{MM})$ and what the semantics will be used for. If it does not have any associated behavior, the semantic domain usually consists only of data structures. Otherwise, $\mathcal{L}(\mathsf{MM})$ has a behavior, and the semantic domain needs to appropriately capture it.

Note also that the semantics *style* depends on the machinery associated with $\mathbb{D}_\mathsf{MM}$: It can be denotational if $\mathbb{D}_\mathsf{MM}$ comes with a functional framework, operational if it is equipped with rewriting capabilities, axiomatic if it defines a Floyd-Hoare logic, or even translational if $\mathbb{D}_\mathsf{MM}$ actually represents a target computer language the semantics is translated into.

4.2 Intent properties

This section details the second important class of our intent domain metamodel in Fig. 2, namely IntentProperty. In our description framework, each transformation intent has corresponding mandatory (optional) properties that all transformations with this intent must (may) satisfy. Some intent properties can directly be instantiated for a given transformation, and other properties are still quite abstract and will need to be concretized for the given transformation before they can be checked. Section 6 demonstrates for two example transformations how it is possible to find out the appropriateness of mandatory properties for a given transformation and also how to concretize abstract intent properties to concrete transformation properties that can be validated.

In the following, we assume a transformation specification $t = (\mathsf{MM}_\mathsf{s}, \mathsf{MM}_\mathsf{t}, \mathsf{spec})$ with its associated execution $\mathsf{TS}_\mathsf{t} = (\mathsf{S}, \mathsf{I}, \longrightarrow)$. Each property is given an abbreviation (in square brackets following the name) that is used to refer to the property in the remainder of the paper.

We start with the following three well-known properties: *termination*, *determinism*, and *type correctness*. Type correctness is specific to model transformations, whereas termination and determinism apply to any computational system. The reader can refer to [5] for further details about the verification of such properties.

Definition 6 (*Termination* [T]) TS_t is *terminating* if there exists no infinite chain $\mathsf{M}_0 \longrightarrow \mathsf{M}_1 \longrightarrow \cdots \longrightarrow \mathsf{M}_n \longrightarrow \cdots$ starting from an input model M_0. We say that M_n is an *output model* for M_0 if there exists a finite chain $\mathsf{M}_0 \longrightarrow \mathsf{M}_1 \longrightarrow \cdots \longrightarrow \mathsf{M}_n$ such that no further transition from M_n exists.

A terminating transformation execution ensures the existence of an output model for any input model.

Definition 7 (*Relation* R_t) Given TS_t, the relation R_t over $\mathsf{I} \times \mathbb{M}$ consists of all pairs of models $(\mathsf{M}, \mathsf{M}')$ s.t. M is an input model and M' is an output model for M.

Note that for a terminating transformation the relation R_t is left-total.

Definition 8 (*Determinism* [D]) TS_t is *deterministic* (or *confluent*) if for each model M that can be reduced to M_1 and M_2 (i.e. $\mathsf{M}_1 \;{}^*\!\!\longleftarrow \mathsf{M} \longrightarrow^* \mathsf{M}_2$), there exists another model M' into which both M_1 and M_2 reduce, i.e. $\mathsf{M}_1 \longrightarrow^* \mathsf{M}' \;{}^*\!\!\longleftarrow \mathsf{M}_2$.

Executing a deterministic transformation means that the execution result does not depend on the order in which actions (leading to transitions) are performed. Note that if TS_t is terminating and deterministic, it is said to be *convergent*, or *functional*, since for each input model, there exists a unique output model. In other words, the relation R_t is right-unique and left-total.

Definition 9 (*Type Correctness* [**TC**]) TS_t is *type correct* if each output model for an input model conforms to MM_t.

$\forall (\mathsf{M}, \mathsf{M}') \in \mathsf{R}_\mathsf{t}, \mathsf{M}' \lhd \mathsf{MM}_\mathsf{t}$

We proceed with the description of more abstract properties that share the same form and that rely on the same artefacts. For a specific transformation obeying a given intent, these properties still need to be concretized for the given transformation afterward to effectively validate the transformation correctness. This is demonstrated in Sect. 6, for some example properties on two example transformations.

Definition 10 (*Structural* [*STR*]/**Semantic** [**SMR**] *Relation Property*) A *structural relation property* is a property of all input/output model pairs $(\mathsf{M}, \mathsf{M}')$ in $\mathsf{R_t}$.

A *semantic relation property* is a property of the semantics $(\mu_{\mathsf{MM_s}}(\mathsf{M}), \mu_{\mathsf{MM_t}}(\mathsf{M}'))$ of all input/output model pairs $(\mathsf{M}, \mathsf{M}')$ in $\mathsf{R_t}$.

For example, an interesting concrete structural relation property is the fact that an *injective homomorphism* needs to exist between each input and output model pair in *t* (this is, for instance, useful for the *query* intent, among others).

An example for a semantic relation property is *simulation*, a property that may hold on the semantics of each input and output model pair in $\mathsf{R_t}$ expressing that the execution of the output model cannot be observationally distinguished from the input model. This means that the output model can be transparently used *in lieu* of the input.

Definition 11 (*Traceability* [*TR*]) *Structural correspondences* between an input model and an output model $(\mathsf{M}, \mathsf{M}')$ in $\mathsf{R_t}$ consist of a relation $\rho_{\mathsf{M},\mathsf{M}'} \subseteq \mathsf{M} \times \mathsf{M}'$. We say that $\mathsf{R_t}$ demonstrates *traceability* if structural correspondences are created during transformation execution $\mathsf{TS_t}$ for each model pair $(\mathsf{M}, \mathsf{M}')$ in $\mathsf{R_t}$.

Note that in the above definition, we have a slight abuse of notation, where we assume $\rho_{\mathsf{M},\mathsf{M}'}$ to be a relation over the set of elements that make up M and M''s structure. Moreover, note that traceability is a special case of a structural relation property, since it is a property of all input/output model pairs in $\mathsf{R_t}$.

Definition 12 (*Structural Preservation* [*STP*]) Let $\mathcal{P}(\mathsf{MM_s})$ (resp. $\mathcal{P}(\mathsf{MM_t})$) be a property language operating on all models conforming to the source (resp. the target) metamodel. A *structural preservation property* stipulates that for each input and output model $(\mathsf{M}, \mathsf{M}')$ in $\mathsf{R_t}$, it holds that whenever a property $\pi \in \mathcal{P}(\mathsf{MM_s})$ holds on the input model M, then an equivalent property $\pi' \in \mathcal{P}(\mathsf{MM_t})$ holds on the output model M'.

$$\mathsf{M} \vdash_{\mathsf{s}} \pi \in \mathcal{P}(\mathsf{MM_s}) \implies \mathsf{M}' \vdash_{\mathsf{t}} \pi' \in \mathcal{P}(\mathsf{MM_t})$$

Note that the structural preservation property is also a special case of the structural relation property, since it is a special property on all input and output model pairs in $\mathsf{R_t}$.

Definition 13 (*Semantic Preservation* [*SMP*]) Let $\mathcal{P}(\mathbb{D}_{\mathsf{MM_s}})$ (resp. $\mathcal{P}(\mathbb{D}_{\mathsf{MM_t}})$) be a property language on the source (resp. target) metamodel's semantic domain. A *semantic preservation property* stipulates that for the semantics $(\mu_{\mathsf{MM_s}}(\mathsf{M}), \mu_{\mathsf{MM_t}}(\mathsf{M}'))$ of each input and output model $(\mathsf{M}, \mathsf{M}')$ in $\mathsf{R_t}$, it holds that whenever $\mu_{\mathsf{MM_s}}(\mathsf{M})$ satisfies a semantic property ϕ, then $\mu_{\mathsf{MM_t}}(\mathsf{M}')$ satisfies an equivalent property ϕ'.

$$\mu_{\mathsf{MM_s}}(\mathsf{M}) \models_{\mathsf{s}} \phi \in \mathcal{P}(\mathbb{D}_{\mathsf{MM_s}}) \implies \mu_{\mathsf{MM_t}}(\mathsf{M}') \models_{\mathsf{t}} \phi' \in \mathcal{P}(\mathbb{D}_{\mathsf{MM_t}})$$

Similar to the syntactic case, the satisfaction relation (represented here by $\models_{\mathsf{s}}$ and $\models_{\mathsf{t}}$, to differentiate from the syntactic case) can differ on each side. It is sometimes possible to predefine a property translator for syntactic or semantic properties if the property languages on input and output are the same, or at least comparable. Generally, however, when they differ too much, or the semantic gap between each metamodel is too deep, no general procedure exists for building such translators. This becomes the designer's job, with all the accompanying issues: Aside from the properties' correctness, the translation can add another source of errors for the validation process [127]. Finally, note that the semantic preservation property is also a special case of the semantic relation property, since it is a special property on the semantics of all input and output model pairs in $\mathsf{R_t}$.

In contrast with relation properties, behavioral properties qualify in a more general way the transformations instead of only considering properties on the input and output models.

Definition 14 (*Behavioural Property* [*BP*]) Let $\mathcal{P}(\mathsf{TS_t})$ be a property language over the transformation execution and $\mathsf{M} \in \mathsf{I}$ an input model. A *behavioural property* $\phi \in \mathcal{P}(\mathsf{TS_t})$ expresses the fact that starting from M, the transformation execution $\mathsf{TS_t}$ satisfies ϕ.

$$\mathsf{M}, \mathsf{TS_t} \models \phi \in \mathcal{P}(\mathsf{TS_t})$$

where $\mathsf{M}, \mathsf{TS_t}$ denotes the part of the transition system reachable from M.

A behavioral property can be seen as the most general property, since it is able to qualify the complete transition system $\mathsf{TS_t}$. All the other properties either qualify only particular parts of the transition system (e.g., input and output models) or express a special property of the transition system (e.g., termination). We can summarize the introduced properties in this section into the hierarchy of properties depicted in Fig. 10. This figure summarizes the kind of properties that we distinguish and the relationships between them. The dashed properties are intent properties that still need concretization for a given transformation.

5 Six intents: restrictive query, refinement, translational semantics, translation, analysis, and simulation

This section presents the six intents we have chosen to illustrate in detail: *Restrictive Query*, *Refinement*, *Translational Semantics*, *Translation*, *Analysis*, and *Simulation*. The choice of this particular set of intents was based on the transformation intents present in our Power Window Case Study, introduced in Sect. 6 of this paper. As can be seen in Table 6, *Restrictive Query*, *Translation*, and *Simulation* are the most abundant intents in the Power Window Case Study, and according to that criterion we chose them as part of the intent set, we analyze in depth in this section. The *Synthesis* intent is also very present in the Power Window Case study. However, as per our catalog, *Synthesis* is a form of refinement. As such, and in the interest of reuse and incrementality, we have chosen to first explore the *Refinement* intent. Finally, the *Analysis* intent is part of our original study on intent of model transformations [4]. As such, after some updates following the most recent version of our intent catalog, we have naturally included it in this study.

Each intent is described systematically using the following approach:

1. We informally present the intent to convey the general idea behind it;
2. We review contributions from the literature to demonstrate different intent usages and help explain how the ModelTransformationIntent instance is built;
3. We formalize the intent as an instance of the metamodel presented in Fig. 2. The goal of this formalization is to provide a *mapping* between the intent, informally presented in Sect. 3, and its properties, defined in Sect. 4.

5.1 Restrictive query

As with queries over databases, a query transformation applied to a model extracts a subset of information from that model. We refer to the extracted subset of information as a *view*. The Query/View/Transformations initial call for submissions [43] defines a query as "an expression that is evaluated over a model" and a view as "a model which is completely derived from another model." This definition is very general since any automated transformation could be viewed as a way of completely deriving one model from another. In this paper, we define a *restrictive query* transformation as one that produces a *restrictive view* of the model by omitting a portion of the model—that is, it extracts a submodel. For example, the query "show only classes/associations of a class diagram" produces a restrictive view that extracts the submodel of a class diagram containing all and only the classes and associations.

5.1.1 Restrictive query in the literature

Restrictive query transformations are often used as a pre-processing step to extract the portion of a model that is needed as input for another transformation. For example, in order to apply an analysis transformation to a state machine within a larger UML model, a restrictive query transformation will first be used to extract this state machine. Restrictive query transformations are also used to support the separation of concerns by extracting the submodels related to different concerns and then feeding these to their own transformations. For example, the UWE web application engineering method using MDE [68] initially uses restrictive query transformations to extract the subsets of the requirements model related to Web site function and Web site architecture. It then feeds these submodels into their own transformation chains that ultimately reintegrate these concerns downstream. We give additional examples of this technique below for the power window case study.

Model slicing represents a type of restrictive query transformation that has received recent attention by the modeling community (e.g., see [74]). Model slicing, like program slicing, is intended to support human comprehension of a complex model by extracting submodels that are restricted to the behavior and properties for a subset of model elements. Some of the slicing techniques produce *amorphous* [56] models, while other produce *structure-preserving* ones. The techniques that produce *structure-preserving* models can be considered as restrictive queries. For example, in [74], the authors describe slicing techniques for various UML diagrams with the goal of producing analyzable models from those diagrams.

Similar approaches have been proposed in the literature for metamodel slicing. For example, the authors of [8] use a model slicing technique to modularize and manage the complexity of the UML metamodel. The technique takes as input key elements of UML diagrams (e.g., Class Diagrams, Use Case Diagrams, etc) and, for those key elements, produces a sub-metamodel that describe such diagrams. The algorithm produces the sub-metamodel by navigating associations emanating from those key elements. Following a similar line of thought, Sen et al. [107] propose in a more generic approach that makes use of a Kermeta [89] model transformation to prune any given metamodel. The goal is to find a sub-metamodel for a given purpose, such as defining the allowed set of inputs for a model processing program or tool. The model transformation takes as inputs the large metamodel and a set of required classes and properties and returns a sub-metamodel including those classes and properties, and their mandatory dependencies. The authors also provide an additional algorithm to check that the pruned metamodel is a subtype of the source metamodel. This ensures that instances of the pruned metamodel are also instances of the source metamodel.

Since the application of a restrictive query on a model produces a model fragment that is not necessarily well-formed, an important consideration for a restrictive query transformation is how to ensure type correctness (i.e. well-formed results). The work of Kelsen et al. [61] provides an efficient algorithm to address this problem by decomposing a fragment into its atomic constituents and then re-merging them while preserving well-formedness. The net effect is that the fragment is expanded to the nearest well-formed submodel that contains it.

5.1.2 Restrictive query metamodel instance

The attributes of the restrictive query transformation intent are shown in Table 1. If we consider the mandatory properties, *termination* **[T]** is a reasonable property to expect from a query—since it is of no use if it never produces a result. We also expect the resulting view to be well-formed with respect to the target metamodel, and so it must be *type correct* **[TC]**. Most importantly, the resulting view must be a submodel of the input, or base model. This is the key property that identifies a transformation as a restrictive query and can be formalized as a *structural relation* **[STR]** enforcing an *injective homomorphism* mapping from the view to the base.

A property that is optional is for the restrictive query to be *deterministic* **[D]**—i.e., that the query should always produce the same result on the same input model. Often this is expected, but there are cases where it is not needed. For example, consider a restrictive query transformation that extracts a submodel of a UML model showing an example of how a class is used. In this case, any sequence diagram that uses the class is sufficient, and it is not necessary to always produce the same one. The optional property that the restrictive query be *semantics preserving* **[SMP]** means that the information in the view submodel should not change its meaning even though it is taken out of context of the whole model. This is often an important requirement when the view has a human consumer (e.g., model slicing) since otherwise the information in the view could be misleading.

Note that in practice, the target metamodel of a *restrictive query* may be also a subset of the input metamodel, or slightly different to accommodate the resulting models. This means for certain transformation implementations the *restrictive query* intent may in fact be exogenous. However, in this study, we privilege a conceptual view on intents and consider that in the general case, a *restrictive query* is an *endogenous* transformation.

Table 1 Restrictive query intent characterization

Attributes	
`name`	Restrictive Query
`description`	Extract the unique submodel (the view) from a model that satisfies some criterion (the query)
`useContext`	1. Want to extract the relevant part (view) of a model for a task
	2. Want to decompose a model to manage complexity
`example`	1. Extract the submodel that are immediate neighbors of a particular element
	2. Extract the submodel of structural elements from a UML model
	3. Model slicing
	4. Model decomposition
`is_exogeneous`	False
`is_endogeneous`	True
`preconditions`	1. Must be able to characterize the submodel of interest using a condition expressible in terms of the metamodel of the base model
Associations	
`mandatory`	1. **[T]** Terminating
	2. **[TC]** Type Correct
	3. **[STR]** The view must be a submodel of the base
`optional`	1. **[D]** Deterministic
	2. **[SMP]** Semantics preservation

5.2 Refinement

A transformation with the refinement intent is a transformation that produces a lower-level specification (e.g., a platform-specific model) from a higher-level specification (e.g., a platform-independent model) [67].

5.2.1 Refinement in the literature

Model transformations from the literature that fall under the refinement intent can be either *interactive* refinement transformations or *fully-automated* refinement transformations. In particular, we investigated the approaches described in the following paragraphs in order to come up with a characterization of the refinement intent as given in Table 2. Moreover, we also considered more general studies [46,96] describing the characteristics of refinement in MDE and its corresponding verification.

Interactive refinement transformations The refinement approaches presented by Padberg [95] as well as by Scholz [106] are rule-based. In the first approach, the rules need to adhere to specific properties in order to guarantee the

Table 2 Refinement intent characterization

Attributes	
`name`	Refinement
`description`	Add precision such that the output model contains at least the same amount of information as the input model. The information contained by a model is equivalent to the relevant questions that can be asked concerning the model [46]
`useContext`	Add more detail to a model
`example`	Going from a platform-independent model to a platform-specific model [67]
`is_exogeneous`	True
`is_endogeneous`	True
`preconditions`	1. A clear understanding of the amount of information described by the input model, and how to preserve it
	2. A clear understanding of the information that needs to be added, and how to add it
Associations	
`mandatory`	1. **[T]** Termination
	2. **[TC]** Type Correctness
	3. **[STR, SMR, STP, SMP]** Information Preservation
	4. **[STR, SMR]** Information Creation
`optional`	

preservation of safety properties, and in the second approach, specific refinement rules already exist. The user decides where to apply which rules. Van der Straeten et al. [111] present a formal approach to model refinement and its interplay with model refactoring. The user of the refinement needs to decide how to refine the models, and afterward, behavior preservation can be checked.

Fully-automated refinement transformations Baresi et al. [9] describe in their work exogenous refinements of business-oriented architectures, abstracting from technology aspects, into service-oriented ones. Mannadiar et al. [83] introduce two exogenous graph transformations, one of which is used to refine a domain-specific model (DSM) of the *PhoneApps* domain-specific language (DSL) for a conference registration mobile application. The authors present the *PhoneApps* DSL for capturing both the behavior and the visual structure of mobile device applications. Tri and Tho [122] discuss an approach for the automatic refinement of SEAM models. SEAM is a language and tool that supports visual modeling. SEAM has the same modeling capability as that of UML with the additional advantage that SEAM can easily maintain consistency between design components since it can capture the entire system in a single view. Due to that single-view representation, the final SEAM model can become too complicated. Thus, the paper describes an approach to automatically refine abstract SEAM models into detailed SEAM models such that the final SEAM model can eventually be used to generate code.

5.2.2 *Refinement metamodel instance*

Table 2 instantiates the intent metamodel of Fig. 2 for the refinement intent, summarizing our findings in the literature. Since transformations with the refinement intent are required to add detail to existing models, it is intuitive that having a clear understanding of the information to be preserved and the information to be added are preconditions for such transformations. These preconditions were mentioned implicitly in all the papers discussed in Sect. 5.2.1. For example, in the rule-based refinement approaches, these preconditions are needed to be able to design the refinement rules as well as to apply them. In the studies discussed in Sect. 5.2.1, it was usually mentioned that the mandatory *termination*, *type correctness*, *information preservation*, and *information creation* properties stated in Table 2 need to be fulfilled. Whereas *termination* and *type correctness* have a one-to-one correspondence with properties **[T]** and **[TC]** in Sect. 4, *information preservation* and *information creation* will generally need to be shown by a collection of several concrete properties, both at the structural and the semantic level. For example, the fact that there is a simulation between each input and output model's semantics might imply information preservation and can be expressed as *semantic relation* **[SMR]** property. Also, having a bijection between the syntactic elements of the input and the output models might imply information preservation and can be expressed as a *structural relation* **[STR]** property. It is also reasonable to think that *information preservation* might be expressed as a set of *structural preservation* **[STP]** properties where the information to be preserved is encoded in the syntactic property that is transported to the output model. The same reasoning holds at the semantic level for the usage of a set of *semantic preservation* **[SMP]** properties. Note that in Table 2, the notation **[STR, SMR, STP, SMP]** means that any non-empty combination of those four properties can be used to formally show *information preservation*.

Information creation implies the existence in the output model of syntactic and semantic elements that did not exist in the transformation's input model. It thus seems reasonable to think that **[STR, SMR]** can be helpful, if necessary, in showing *information creation*, depending on the notion *information creation* required by the considered transformation.

Some of the papers we surveyed have not explicitly verified all the properties in Table 2. Our work aims at identifying these gaps in order to allow for a more systematic engineering of model transformations with specific intents in the future.

For example, in [83,122], the mandatory properties were not verified and case studies were used to demonstrate that the refinement transformations fulfill their purpose. Both studies also informally discussed how a mapping is done between the input model and the refined model and, thus, how information is preserved. Usually, the information creation property does not need to be checked explicitly, since it trivially follows from applying a refinement in the corresponding approaches. For endogenous approaches like [95], it is moreover usually trivial to check type correctness.

5.3 Translation and translational semantics

A transformation with the *translation* intent is a transformation that translates the meaning of models conforming to a source metamodel into models conforming to a target metamodel. The resulting models can then be used to achieve tasks that are difficult, or impossible, to perform on the originals.

This section describes two intents of our catalog: *Translation* and *Translational Semantics*. Despite the fact that it made sense to distinguish them from an engineering point of view in the catalog, they are very similar from a verification point of view. As the name suggests, a *translational* semantics is no more than a translation whose purpose is to provide semantics to a metamodel (or more often, to a DSL) by mapping its concepts into the ones of the target metamodel, which becomes the so-called *semantic domain*: This is the exact definition of the *Translation* intent, which implies that the associated properties should be similar.

5.3.1 Translation in the literature

From the review of the contributions present in the literature, it appears that a translation is performed for two main reasons:

Bridging structures to enable metamodel exchanges at a structural level (e.g., for importing models from another metamodeling framework);

Delegating actions to the target metamodel by *simulating*, or formally *analyzing* input models using dedicated engines available for the output models. The delegation is valuable in the case where the cost of re-implementing a simulation/analysis engine for the source metamodel is too high.

Bridges The four-layered organization depicted in Fig. 9 is shared by several technical spaces: modelware, grammarware, ontoware, or dataware, to name just a few [91,131]. Often, one has to bridge artifacts from one to another: For example, query languages and transaction operations in dataware are already available, and taking advantage of SQL and its many capabilities and various implementations, one can simply reuse instead of reimplementing things for a novel technical space. The goal of a bridge is to translate the meaning of the meta-metamodel itself, i.e., offering a way to automatically convert any metamodel of one technical space into another. This differs from the usual understanding of a transformation shown in Fig. 9, where the transformation is specified on a metamodel and executes on a model, not the level above. However, as previously noted, a meta-metamodel can usually be treated just as a metamodel and manipulated as such. Furthermore, bidirectional bridges are usually required for enabling round-trips between technical spaces.

Two papers [91,131] published in 2005 explicitly use the terms *grammarware* and *modelware* to refer to exchanges between textual and visual representations of models. Most probably, closing the gap between language theory or compilation techniques (based on BNF grammars) and MDE (usually using MOF) are the most represented contributions [28,58,91]. Kern and his colleagues performed several bridges from various meta-metamodels into either MOF or its specific Eclipse implementation EMF: GOPRR, used in the commercial transformation engine MetaEdit+ [62]; Aris, the well-known enterprise modeling tool [64]; Visio, the Microsoft general-purpose modeling tool [65]. A comparative study is available in [63], where the authors also describe the bFlow Toolbox, their integrated tool for performing these bridges seamlessly. Brunelière et al. [16] and Bézivin et al. independently studied bidirectional bridges between Microsoft DSL Tools and Eclipse EMF, providing an efficient way to exchange models between one of the most popular DSL development tools.

Simulation delegation A *Translation* is often specified for providing simulation (or execution) capabilities for models. This type of delegation is a popular approach for defining the semantics of DSLs: This kind of translation is more precisely called *Translational Semantics*. Since they capture domain expertise as concepts and rules with a precise meaning, the *Translation* just transposes these semantics in terms of a target metamodel that offers the necessary execution machinery. Another popular use for *Translation* consists of taking advantage of a richer framework to perform tasks specific to simulation, such as calibrating the parameter values of models to enhance their non-functional properties (typically, performance). The main difference resides in whether the source metamodel's semantics is known a priori or not: For a translational semantics, the translation itself serves as a semantics definition.

Rivera et al. [100] use Maude for specifying the behavioral semantics of domain-specific modeling languages and for simulating the models by executing them using Maude rewriting rules. Kühne et al. [71] define a transformation from Finite State Automata into Petri Nets, implementing the automata's semantics: By running the Petri Net translated model over an input sequence, it can check whether it belongs to the language accepted by the input automaton model.

MoTif results from combining the T-Core framework with the discrete event simulation language DEVS [116]. This allows model transformations to be expressed in a modular and compositional way together with the explicit introduction of a time dimension. In [112], Syriani et al. demonstrated how adding the notion of time allows for the simulation-based design of reactive systems such as computer games. This allows the modeling of player behavior and the incorporation of data about human players' behavior and reaction times. The models of both player and game were used to evaluate, through simulation, the playability of a game design.

Troya et al. [123,124] employ simulations based on model transformations for reasoning about aspects of quality of service (QoS), such as performance and reliability. In their work, Troya et al. add not only general runtime information to the models, as is for example, done in [35] or in fUML, but they also add specific elements called observers to track information the designer is interested in. The authors used e-Motions [100] for implementing and executing the behavior of the models to simulate. Internally, e-Motions is compiled to Maude.

Analysis delegation A *Translation* can take advantage of the analysis capabilities of the target metamodel.

De Lara and Taentzer transform in [76] models for process interaction expressed in a discrete event formalism tailored for the manufacturing domain into timed transition Petri Nets. This transformation is expected to terminate, to be deterministic, to type correct, and to preserve process interaction's behavior. Termination, type correctness, and behavior preservation are proved informally, but determinism is proved using the classical critical pairs technique already implemented in AGG, the tool used to specify the transformation.

Varró et al. [126] prove the termination of graph transformations with negative application conditions by translating them into Petri Nets and showing that the resulting Petri Net is not partially repetitive, i.e., no (initial) marking has a firing sequence in which a transition occurs infinitely many times. Augur2, a graph transformation tool proposed by König and Kozioura [69], approximates transformations with Petri Nets for analyzing property preservation. Here, the property is specified as a graph pattern and then translated into an equivalent marking, which is checked for reachability. A counterexample is produced in case the marking is not reachable.

Naranayan and Karsai [93] proved reachability within StateCharts using a two-layered translation. First, a StateChart model is translated into an Extended Hybrid Automaton model, building traceability links between both instances. Then, the Extended Hybrid Automaton is translated in PROMELA, the entry language of the SPIN model-checker, where reachability can be checked. If a counterexample is produced, it can be traced back to the StateChart model following the traceability links previously established. Notice, however, that this technique is not general: It checks a particular property (reachability in the paper) on a particular StateChart model and works only because the StateChart and the Hybrid Automaton models are proved to be bisimilar.

Cabot et al. [19] automatically extract OCL invariants from bidirectional transformations expressed declaratively in QVT [50], using a higher-order transformation. These invariants allow one to answer various questions about the transformation, such as whether a valid input or output model exists for the transformation, or whether an output model exists for any possible valid input. However, the actual invariant satisfaction problem is delegated to specialized tools able to work on UML models decorated with OCL invariant constraints.

5.3.2 Translation metamodel instance

Table 3 shows the ModelTransformationIntent's instance for the *Translation* intent.

A *Translation* is by nature terminating **[T]** and deterministic **[D]**; otherwise, the expected output models could never exist, or could be ambiguous regarding the original input. Because the output is expected to somehow "represent" the input, the transformation should be type correct **[TC]**.

The remaining mandatory properties depend on both the *Translation*'s nature and the existence of a precise semantics for the source metamodel.

Bridge If it is possible to attach a formal semantics to both meta-metamodels, then it becomes possible to formally compare conforming metamodels of both sides; otherwise, it should be possible to define structural preservation **[STP]** between both sides.

Simulation If the *Simulation* defines the source metamodel's semantics, then structural preservation **[STP]** is the only possible property. Otherwise, in case the source metamodel has a predefined semantics, structural **[STP]**, but also semantics preservation **[SMP]**, are possible. It can also be interesting to prove a simulation relation **[SMR]** between the input and the output, thus ensuring for reactive systems that all actions of the input can actually be performed by the output (but obviously, also more actions, typically time-related).

Table 3 Translation intent characterization

Attributes	
`name`	Translation
`description`	Translate the meaning of conforming input models into models conforming to a target metamodel to achieve a subsequent task
`useContext`	Equip a DSL with an executable semantics, or perform a task difficult, or impossible to realize over the original models
`example`	1. Provide a reference semantics for a DSL. (cf. PWCS) 2. Exchange models between Microsoft Visio and Eclipse EMF [65] 3. Prove reachability in StateCharts using PROMELA [93]
`is_exogeneous`	True
`is_endogeneous`	True
`preconditions`	
Associations	
`mandatory`	1. **[T]** Termination 2. **[D]** Determinism 3. **[TC]** Type Correctness 4. **[STP]** Semantic equivalence (*Bridge*) 5. **[SMR, SMP]** Observational equivalence / Similarity (*Simulation*, when source semantics available) 6. **[STR, STP]** Structural Preservation (*Simulation*, without source semantics available) 7. **[STP, SMP, SMR]** Soundness (*Analysis*)
`optional`	1. **[TR]** Backward Traceability to relate results back to the input 2. Readability of the transformation's output for debugging purposes

Analysis Since an *Analysis* generally focuses on particular aspects of the inputs, the transformation should be "sound", i.e. it should verify some form of preservation of the property under analysis **[STP,SMP]**. Depending on the abstraction level difference of both sides, it is also possible to verify a simulation relation **[SMR]** between models in each side.

Some optional properties are also sometimes desirable. As already mentioned, a *Bridge* could sometimes be bidirectional. Traceability **[TR]** is also desirable for *Analysis* and *Simulation* to be able to relate results back to the input, for example, playing a counterexample obtained from an analysis in terms of the input to help identify errors.

5.4 Analysis

A transformation with the *Analysis* intent is a transformation that implements an analysis algorithm of any complexity. Examples include the following: the computation of a call graph for operations of a MOF model, detecting dead code or inapplicable rules, and the model-checking of a temporal formula over a given structure.

5.4.1 Analysis in the literature

From the literature review, we noted two types of scenarios in which *Analysis* occurs. A transformation is a:

Pure Analysis Transformation if it expresses an analysis algorithm on its own, i.e. computes the necessary information for performing the analysis.

Built-In Analysis Transformation if it is executing with a transformation engine that is already equipped with analysis features.

Pure analysis transformations This *Analysis* scenario is, in fact, very rare. One reason is that specifying an analysis algorithm is typically complicated, and so it is often easier instead to delegate the analysis to a dedicated tool after having translated the models. Furthermore, a key issue when analyzing models is scalability, and this often requires the use of dedicated data structures to enable performance gains (e.g., consider the use of binary decision diagrams for scalable model-checking).

Narayanan and Karsai [94] have implemented a graph rewriting system in GREAT to transform UML activity diagrams to Communicating Sequential Process (CSP) models. The graph rewriting system was then checked for preserving structural correspondences between input and output models (property preservation). Unfortunately, no data related to the performance and scalability are given. Recently, Lúcio and Vangheluwe [81] explored the possibility of checking structural correspondence properties on DSLTrans transformations. The approach scales up to 21 rules for a transformation with acceptable computation times.

Built-In Analysis Transformations This *Analysis* scenario corresponds to the fact that a transformation is expressed in a transformation framework that is natively equipped with formal analysis capabilities. When possible, this represents a good choice, since the analyses are tailored for the transformation engine, ensuring good performance.

Rivera et al. [101] encode graph transformations into Maude [25]. Graph transformations are specified visually by using AToM^3 [77] as a front end and encompass negative application conditions, well-formedness rules, and both single and double pushout approaches. Since Maude provides reachability analysis, LTL model-checking, and theorem-proving capabilities, all these analysis become available for graph transformations, and the results are easily traced back due to their high-level encoding of (meta-)models. Gargantini, Riccobene, and Scandurra [44] use Abstract State Machines (ASM) [14] to encode a DSLs' semantics. Metamodels and models are expressed with EMF, whereas the transformation expressing their operational semantics is expressed with the ASM language. Using the built-in bidirectional translation into νSMV, it becomes possible to perform LTL model-checking in this framework. Groove [99] allows the (bounded) model-checking of CTL formulæ over graph-based transformations with negative conditions [60]. The tool can also handle reachability analysis by expressing adequate invariants in CTL.

5.4.2 Analysis metamodel instance

Table 4 shows the `ModelTransformationIntent` instance for the *Analysis* intent. This intent is closely related to two other intents: *Translation* and *Simulation*: A *Translation* often delegates an analysis to the target metamodel, whereas a *Simulation* can directly benefit from the potentially available analysis capabilities of the simulation engine. When such capabilities exist, the task of the transformation designer consists of just specifying the transformation adequately (i.e. in the engine's own language): the analysis becoming the transformation's engine responsibility, not the designer's. For example, Riveira et al. [101] (cited as example in Table 4) use Maude as such a target, providing model-checking and theorem-proving analysis for all transformations specified within their framework.

Table 4 Analysis intent characterization

Attributes	
`name`	Analysis
`description`	Perform an analysis on the input models
`useContext`	1. Develop an analysis algorithm using transformations 2. Benefit from the built-in analysis capabilities of a transformation engine
`example`	Reuse Maude's model-checking capabilities for model-checking graph transformations [101]
`is_exogeneous`	True
`is_endogeneous`	False
`preconditions`	1. Access to analysis algorithms
Associations	
`mandatory`	1. **[TC]** Type correctness
`optional`	

Pure analysis transformations are obviously type correct when delivering a result **[TC]**. Beyond this, it is difficult to say more since it highly depends on the analysis being performed. They are not necessarily required to be terminating, or deterministic, since many types of static analysis are undecidable. For example, consider a model-checking procedure: It does not generally terminate for infinite systems, and if it does, the only requirement is to answer with one counterexample among all possible ones. In general, proving an analysis transformation's correctness is roughly equivalent to proving the correctness of an implementation with respect to an analysis algorithm. For example in [81], the authors would be asked to prove that their transformations actually correctly realize model-checking.

The *Analysis* intent clearly needs further research. The fact that we cannot better characterize such an intent also comes from the fact that it is often, based on our observations, neither an atomic intent nor has a single-target (consider again model-checking: The analysis verdict is, if negative, accompanied with a counterexample).

5.5 Simulation

In the modeling community, simulation is a transformation that encodes some operational semantics of a language. Therefore, it simply updates the state of a model in response to events (e.g., time, trigger, causal dependency). We can define a simulation such that its trace of execution is a label-transition system (LTS) where a node is a legal snapshot of the state of the model and a transition is the application of a rule.

Note that the term "model simulation" is understood differently in the modeling community and the simulation community. In the modeling community, model simulation normally refers to the development of an operational semantics for a modeling language, while in the simulation community, simulation [109] refers to the process of designing a model of a real system and conducting experiments with this model for a certain purpose. Thus, the first interpretation can be seen as the enabler of the latter.

5.5.1 Simulation in the literature

There is a large body of work discussing how to implement the operational semantics for modeling languages. Generally, there are two approaches for defining the behavior of models: (i) by incorporating the runtime concepts into the metamodel and adding transformation rules for evolving the initial state of a model, and (ii) embedding the modeling language into some existing simulation formalism (as already discussed in Sect. 5.3.1). Thus, we refer the interested reader for the second approach to Sect. 5.3.1 and deal in this subsection only with the first one.

Concerning the first approach, one way for defining an operational semantics is to introduce executability concerns by defining graph transformation rules operating on metamodel instances as proposed by Engels et al. [35]. Another possibility is to follow an object-oriented approach by specifying the behavior of operations defined for the metaclasses of a modeling language (within the metamodels representing the abstract syntax of the languages) using a dedicated action language. Many action languages have been proposed, including the use of existing general-purpose programming languages: Kermeta [89], Smalltalk [33], xCore [23], EOL [103], the approach proposed by Scheidgen and Fischer [105], and fUML [84]. Prominent examples used in these papers are the definition of the operational semantics of Petri Nets or state charts.

Most of this work only addresses the definition of the operational semantics of languages that model discrete systems without time, i.e., the time elapsed between two state changes is not considered. However, there are also some approaches dedicated to modeling specific real-time systems that require an explicit notion of time. For instance, de Lara et al. [75] use the so-called flow graph grammars for scheduling graph transformation rules and scheduling grammars for introducing an explicit notion of time for modeling a mail system.

An interesting problem and solution are presented in [13,36] where the goal is to have consistency between animation rules operating on the concrete syntax of a model and the simulation rules operating on the abstract syntax of the model. Although we consider this consistency property between animation and simulation as very important, in this paper, we focus only on properties inherent to the simulation intent.

5.5.2 Simulation metamodel instance

Table 5 shows the `ModelTransformationIntent` instance for the *Simulation* intent.

As already mentioned, the purpose of simulation in MDE is to give operational semantics to a modeling language by updating the state of a model. Of course, this applies only to behavioral models. The transformation is considered to be either exogenous if an already existing simulation formalism is selected for this purpose or endogenous if the behavioral semantics is directly attached to the language's metamodel.

In general, a simulation is a terminating transformation. When a terminating condition is met, the simulation must stop. This condition can be based on the state of a model, on the gained information, or on time. This latter point means that the transformations are expected to terminate at some point in time, although it may happen that the simulation has to be stopped even though there are still rules that can be applied. Concerning the second point, sometimes the successful execution of the simulation is meant to be non-terminating unless an information saturation point is met. This can be a failure or an exception case arises that may lead to rejecting the hypothesis to be tested, or the opposite, the information gained allows to accept the hypothesis. To sum up, *controlled termination* **[T]** has to be supported.

Whether a simulation transformation is deterministic depends on the system being modeled and cannot be decided on a general basis. If the system is deterministic, the simulation should be deterministic, too. If the system is non-deterministic, and several transformation rule matches are found at some point, one of them is non-deterministically selected and applied.

Each simulation step should result in a valid model with respect to its metamodel as such *type correctness* **[TC]** needs to hold. However, ensuring this may require a sequence of several transformation rules corresponding to a single logical simulation step—similar to the concept of transaction.

A means for proving that a simulation is correct is to show that a set of desired *behavioral properties* **[BP]** hold.

Table 5 Simulation intent

Attributes	
`name`	Simulation
`description`	To give an operational semantics to a modeling language by updating the state of the model
`useContext`	Need to compute the trace of a model's execution, its final state or both
`example`	Compute worst-case execution time, throughput, error rates of a production model
`is_exogeneous`	False
`is_endogeneous`	True
`preconditions`	1. Access to intended semantics
	2. Metamodel contains runtime information as is currently provided by the dynamic metamodeling approach [35]. As an example for runtime information consider the token concept in Petri Nets
	3. Modeling language has behavior
	4. Real-time systems require a notion of time
Associations	
`mandatory`	1. **[T]** Controlled Termination
	2. **[TC]** Type correctness
`optional`	1. **[BP]** Preservation of properties of interest
	2. Log of simulation is accessible
	3. Readability of the transformation's output
	4. If animation is provided, it has to correspond to the simulation

Examples of such properties could be invariants or reachability constraints over the set of reachable states of the simulated system. Due to the fact that simulations may be also useful without proving such properties, they are considered optional.

Logging of transformation execution events is considered to be an useful but optional property. In particular, some transformation engines are able to produce complete logs, e.g., the order of the rules applied, the different execution states, the binding of the rules, and timing information. Some approaches also provide the means to automatically produce views on the logging information to support better understandability of the simulation results, e.g., to show the number of events per event type. This is also connected to an optional property, the readability of the transformation's output. Here, not only the output model has to be in a human-readable form, but also the logging information since it may be considered to form a critical aspect of the simulation result.

Because an animation of a simulation is optional, we consequently consider the consistency property between animation and simulation as an optional property.

6 Identifying transformation intents within the power window case study (PWCS)

This section introduces the case study for this paper, developed in the context of an industrial project aimed at building control software for an automobile's power window [29,92]. A power window is basically an electrically powered window. Such devices exist in the majority of the automobiles produced today. Besides lifting and descending the window, a power window also includes an increasing set of additional functionalities, aimed at improving the comfort and security of the vehicle's passengers. To manage this complexity while reducing costs, automotive manufacturers use software to handle the control of such physical devices.

The case study consists of a chain of model transformations aiming at generating control software for a power window as C code, starting from high-level requirements. The whole transformation chain contains 37 transformations and involves 28 different metamodels.

The Power Window Case Study (PWCS) serves as an experimental platform for the research presented in this paper. We use it for two complementary purposes: (i) Since the PWCS was developed independently of our research, it presents an unbiased collection for partially validating the Intent Catalog of Sect. 3 by allowing us to identify occurrences of intents in a real-world transformation chain; (ii) the PWCS can be used to illustrate the practical usefulness of our Intent/Property mapping in Sect. 5 and of our abstract framework for formalizing properties provided in Sect. 4.2. In particular, we extract from the transformation chain two witness transformations in Sect. 6 for two exemplary intents, namely, *Translation* and *Simulation*. By using our mapping, we illustrate how to build concrete transformation properties that help in showing the correctness of the two witness transformations.

Section 6.1 presents the FTG+PM formalism in which the PWCS transformation chain is expressed. Section 6.2 describes the transformation chain itself, with an emphasis on the steps the witness transformations are extracted from. In Sects. 6.3 and 6.4, we describe for two given PWCS transformations how we identified them with the *translation* and *simulation* intent, respectively, according to the process described in Fig. 4. In Sect. 6.5, we provide an overview of the intents identified within the Power Window Case Study.

6.1 FTG+PM: formalism transformation graph and process model

Figure 11 depicts a simplified version of the FTG+PM (Formalism Transformation Graph + Process Model) for the PWCS. On the left, the FTG describes which domain-specific formalisms, represented as labeled rectangles, are consumed and produced by which transformations, represented as labeled small circles. On the right, the PM describes two flows in a way similar to UML activity diagram [51]. The transformation control flow, corresponding to thick arrows, describes how transformations, depicted as round-edged labeled rectangles corresponding to *actions*, are chained to produce the expected final result. These actions are "typed" as executions of the transformations declared in the FTG with the same label. The data control flow, corresponding to thin arrows, describes how models, depicted as square-edged rectangles, are consumed and produced by transformations executions. These *data objects* have to be valid instances of the formalisms with the same label surrounding the transformations whose execution uses them. Similar to activity diagrams, the control flow can join or fork sets of actions (depicted as horizontal bars with decision nodes as diamonds). The FTG and the PM distinguish between automatic transformations (using yellow circles and rectangles, respectively) from manual or assisted ones (colored in gray).

6.2 Description

The FTG+PM for the PWCS contains several phases: *Requirements Engineering*, *Design*, *Verification*, *Simulation*, *Calibration*, *Deployment*, and finally *Code Generation*. We focus on the four first phases that lead to a viable, trusted system that can then be calibrated and deployed: The construction of transformation properties used in Sect. 6 is extracted from these phases. A detailed description of the PWCS can be found in the corresponding literature [29,79,92].

Note that the PM of Fig. 11 contains collapsed blocks (e.g., *Model Requirements*, *Safety Analysis*, or *Hybrid Simula-*

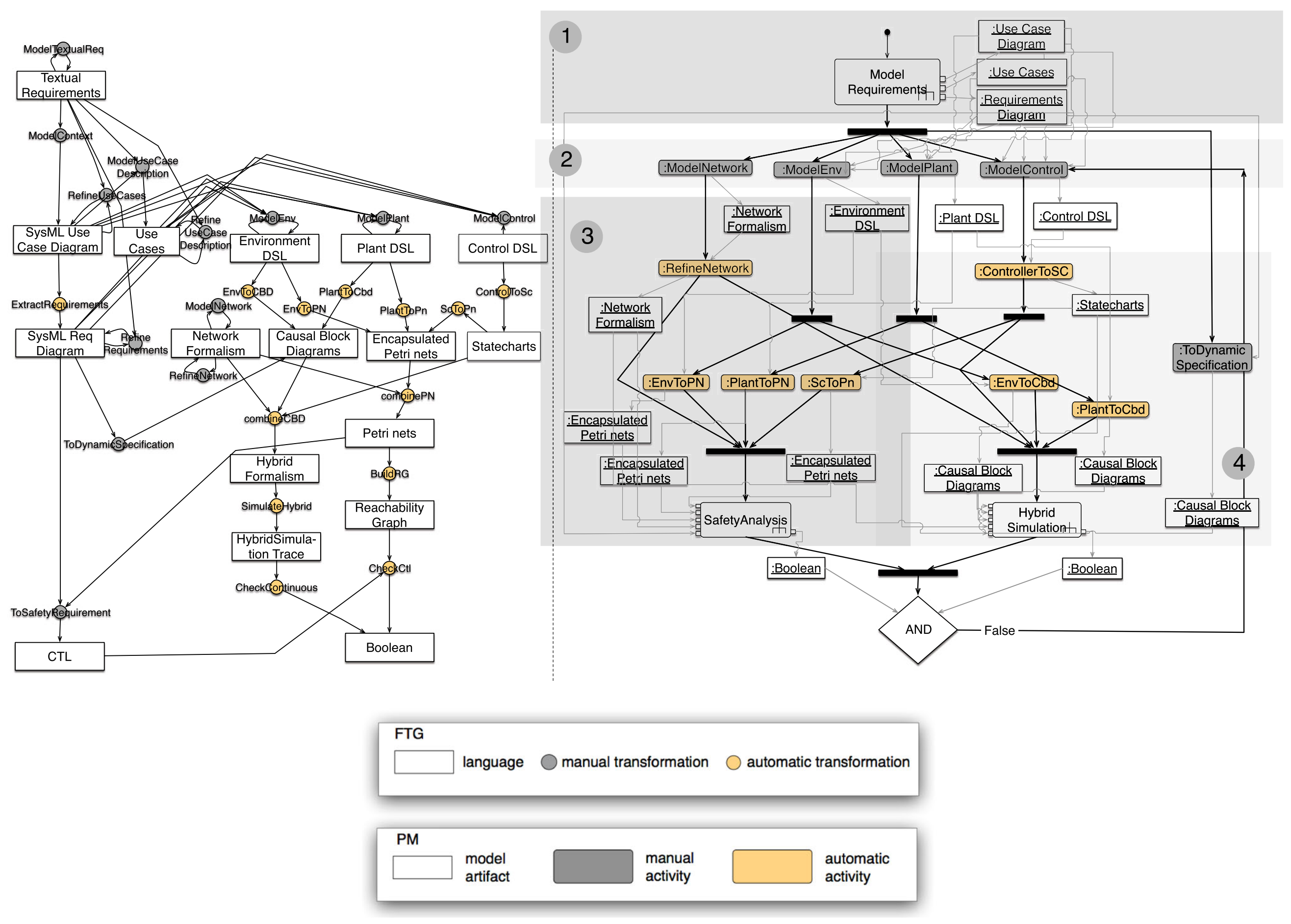


Fig. 11 Partial FTG (on the *left*) and PM (on the *right*) for the Power Window software development

tion) that hide the details of the corresponding tasks. Whenever relevant for the explanation, we will explicitly detail the blocks' content.

① *Requirements Engineering* Before design activities can start, engineers have to extract requirements from legal and technical documents in order to produce requirement and use case diagrams that document what is expected from the system. These transformations are usually done manually, although some parts could be automated (e.g., for populating those diagrams).

② *Design* Using these requirement artefacts, software engineers start the design activity following design practices inspired from control theory [31]: The *controller* is the piece of software controlling the window's functionalities; the *process* (also called *plant*) is the physical power glass window with all its mechanical and electrical components, i.e., the mechanical lift, the electrical engine, and the sensors detecting the window's position or collision events; and the *environment* is constituted of the human actors and the other vehicle subsystems, e.g., the central locking system, the ignition system, etc (the way the PWCS is built closely follows the work by Mostermann and Vangheluwe [88]). Each aspect of the system is captured by a dedicated DSL (Domain-Specific Languages explicitly named *Environment*, *Plant* and *Control* in Fig. 11), later bound together using an extra *Network* DSL for expressing how they interact.

After this phase, the entire system is modeled and can be deployed. However, regulations in the automotive sector have strong security concerns that need to be addressed at early stages of the system design. Since the Power Window is a critical system, two validation tasks, namely verification and simulation, are conducted in parallel in the PWCS to ensure that the code generated from the models is trustable.

③ *Verification* Formal Verification is applied by translating all domain-specific models from the previous stage into corresponding Petri Nets [97]. All the resulting Nets are then composed accordingly to the *Network* model in order to obtain a fully functional Petri Net, on

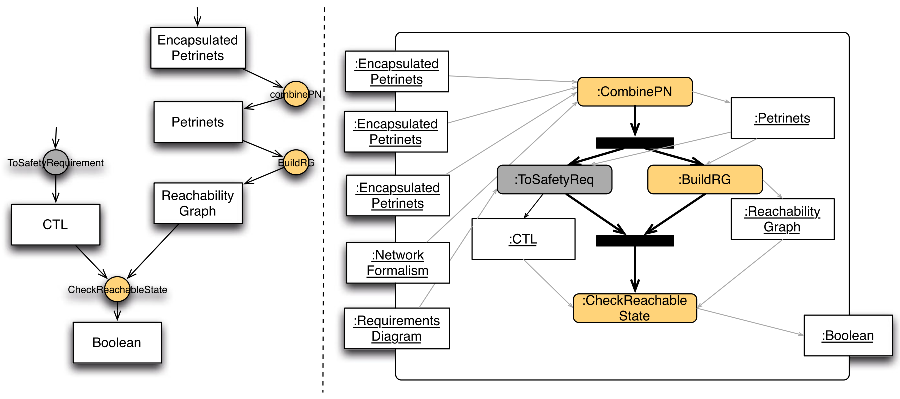


Fig. 12 Safety Analysis FTG+PM Slice, with FTG on the left and PM on the right

which reachability analysis of undesired states, specified according to the requirements, is then checked.
Figure 12 describes the details of the collapsed block corresponding for safety analysis. On the right side, the :CombinePN composes the five models resulting from the previous *Design* activity into a combined Petri Net that describes the behavior of the whole system. This combined Petri Net is the source of two activities performed in parallel: The :toSafetyReq, which requires human intervention, produces a set of CTL formulas encoding the requirements based on a safety requirement model; and the :BuildRG automatically builds the reachability graph corresponding to the combined Petri Net model. These activities are then joined together, since they are prerequisites before the ReachableState action is executed, for model-checking the combined Petri Net behavior against the safety requirements, and produces a verdict (given as a boolean value).

④ *Simulation* On the other hand, a simulation of the whole system is conducted to evaluate the responsivity when interacting with the passengers. The continuous behavior of the window is modeled using a hybrid formalism: The models for the environment and the plant resulting from the *Design* phase are translated in Causal Block Diagrams (CBDs)[2], whereas the controller model is transformed into a StateChart. The process of verifying the continuous behavior is similar to the *Verification* phase, although as a requirement language CBDs are also used.

When the *Verification* and *Simulation* tasks are both completed, engineers can think about how to efficiently deploy the system on the platforms they target. The *Calibration* phase aims at extracting a performance model that gives measurements about the execution times corresponding to the different use cases. This performance model is then used during the *Deployment* phase for selecting a deployment solution with real-time behavior where spatial and temporal requirements are respected. Finally, when a feasible solution is found, the code specific to the target platforms can be synthesized: This includes the code of the application itself, but also the code corresponding to the middleware and to the runtime environment. The complete FTG+PM for the Power Window Case study can be found in [78].

6.3 Translation

As a first example transformation for which we want to identify the intent, we chose the *EnvToPN* transformation located in Area ③ in Fig. 12.

Select intent using description attribute As aforementioned, the *EnvToPN* transformation takes a model expressed in the *Environment DSL* language and produces as result a model in the *Encapsulated Petri Nets* language. The purpose of this translation is to profit from the fact that the *Encapsulated Petri Net* has a well-known and studied semantics which can be used as a semantic domain for the *analysis* of models of the *Environment DSL* language. The *Environment DSL* language has no explicitly formalized semantics, and the role of the translation is to provide an artifact that can explicitly produce such semantics in the form of a Petri Net-like formalism. Consequently, the obvious intent of the transformation is to provide *Translational Semantics* to *Environment DSL* models in terms of the Petri Net formalism. This fits nicely to the description of the *Translation* intent in Table 3.

Check remaining intent attributes The *useContext* mentioned in Table 3 and the fact that the transformation needs to be exogeneous fit both as well. In what concerns the *example* attribute, example 3 is the same kind of translation having *analysis* as its purpose.

Check appropriateness of mandatory properties Let us now switch to checking if the mandatory intent properties are appropriate for the *EnvToPN* transformation. As can be observed in Table 3, the *Translation* intent has as mandatory properties *termination*, *determinism*, *type correctness*,

[2] Causal Block Diagrams are a general-purpose formalism used for modeling causal, continuous-time systems, mainly used in tools like Simulink.

and *soundness*. As for the first three properties, it is obvious that they are appropriate.

Because the semantics of models written in the *Environment DSL* language is not defined, it is not meaningful to discuss the preservation of semantic properties for the *EnvToPN* transformation. It is, however, meaningful to preserve syntactic properties of an *Environment DSL* model that reflect its correct translation into an *Encapsulated Petri Net* model. Consequently, we can conclude that the mandatory properties are appropriate.

Select optional properties The optional properties for the translation intent are *backward traceability* and *readability*. The implementation traceability was not required given the simple nature of the properties being verified in the PWCS. Special care was, however, devoted to the readability of the transformation's output such that, given the very visual nature of Petri Nets, the models resulting from the *EnvToPN* transformations could be understood by humans. This proved useful both for debugging and especially for demoing purposes, as the PWCS has been presented at several venues as an example of transformation chaining for the construction of complex systems using MDE principles.

Outlook to validating properties After having identified the intent for the *EnvToPN* transformation, we want to validate as described in Fig. 5 if its mandatory/selected optional properties are indeed fulfilled. We give a brief idea of this process and first have a more detailed look into the *EnvToPN* transformation.

Figure 14 depicts the result of executing the *EnvToPN* transformation on the model in Fig. 13. The model in Fig. 14 represents the parallel issuing of two sequences of statements. The box annotated with *"Driver"* sends out four sequential commands to the set of buttons on the driver's door, and the box annotated with *"Pass"* send three sequential commands to the set of buttons on the passenger's door. Note that each command box has a number in it, which represents the amount of time during which the command is in effect. The translational semantics of the model in Fig. 13 is produced by transformation as the *Encapsulated Petri Net* model in Fig. 14. The resulting model is a Petri Net where the commands issued by the driver are merged with the commands issued by the passenger along the same Petri Net transition timeline. Petri Net transitions pass messages to outside of the component via ports, represented as black squares on the border of the component. Due to timing constraints, the driver and the passenger commands are sometimes issued simultaneously. In the model in Fig. 14, this translates into the fact that some of the transitions on the Petri Net in Fig. 14 are connected to more than one *port* in the component.

In Fig. 15, we express a structural preservation **[STP]** transformation property that we wish to hold for the *EnvToPN*

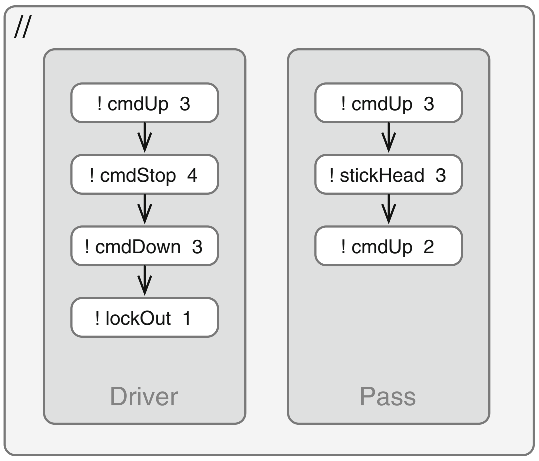


Fig. 13 Example model for the *Environment DSL* language

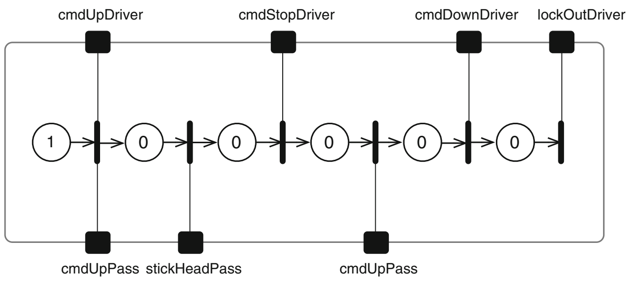


Fig. 14 Example model for the *Encapsulated Petri Nets* language

transformation. Several authors [3,17,18,21,47,52,82,125] have studied structural preservation properties. They allow expressing in a similar fashion how the structure of the transformation's input model influences the structure of the transformation's output model. In order to express such properties for all executions of a transformation, those languages typically use a mix of the transformation's source and target metamodel elements, additional constraint languages (e.g., OCL [17,18,21,47,52,125]) and often metaclasses allowing describing traceability connections between the source and target metamodel elements [3,17,82]. For our example purposes, we have chosen the property language defined in [82], which we have based ourselves upon to express the transformation property in Fig. 15.

The **[STP]** transformation property in Fig. 15 states that whenever the input model includes two sequences of parallel output commands, each of those sequences containing a *first* and a *last* command, the resulting output model will merge the two *first* commands as a single transition, and the final transition is coming from the *last* command of either the first or the second sequence (but not both, as denoted by the XOR operator). Note that in Fig. 15, the thick dashed arrows between *Precondition* or *Postcondition* elements state those elements are indirectly linked; thin dashed arrows between *Precondition* and *Postcondition* elements represent traceabil-

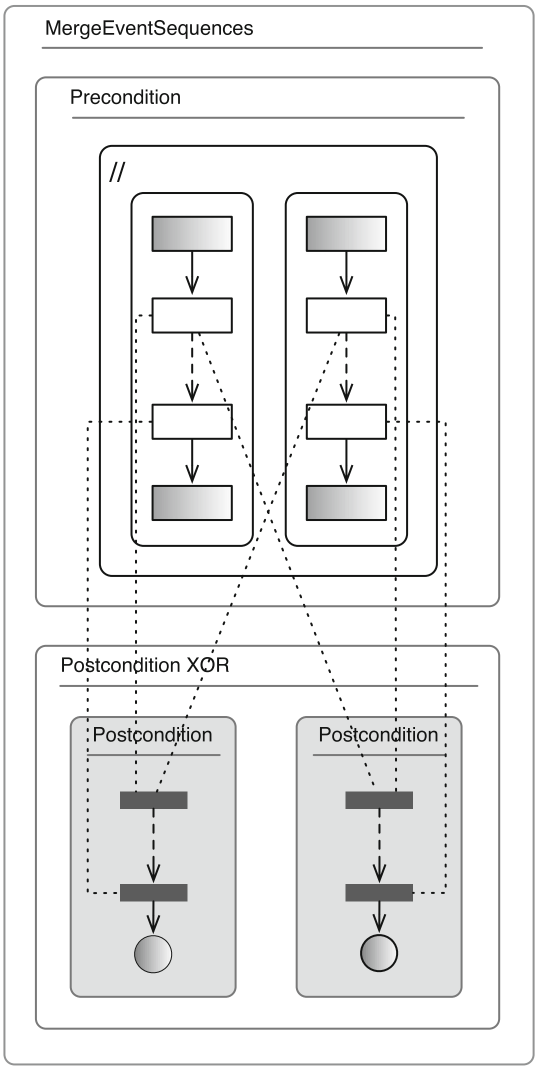


Fig. 15 Syntactic property preservation example for the EnvToPN Power Window transformation

ity links; and blend colored elements represent negative condition, i.e., elements that cannot not occur in input/output models.

In Sect. 4, we have defined **[STP]** properties as follows:

$$\mathsf{M_i} \vdash_\mathsf{s} \pi \in \mathcal{L}(\mathsf{MM_s}) \implies \mathsf{M_o} \vdash_\mathsf{t} \pi' \in \mathcal{L}(\mathsf{MM_t})$$

For the example property in Fig. 15, π and π' correspond respectively to the *Precondition* and *Postcondition* part of the property. Also, $\mathsf{M_i}$ and $\mathsf{M_o}$ are instances of the *Environment DSL* and *Encapsulated Petri Net* languages respectively, and $\mathsf{M_o}$ is the result of applying the *EnvToPN* transformation to $\mathsf{M_i}$. If $\mathsf{M_i}$ instantiates the property's *Precondition* pattern π, then $\mathsf{M_o}$ necessarily instantiates the property's *Postcondition* pattern π'. Note also that π and π' are related by the property's traceability links connecting the property's *Precondition* and *Postcondition* elements.

6.4 Simulation

As a second example transformation, we have selected a Petri Net simulation called *BuildRG* and located in Area ③ of Fig. 12. We describe the intent identification of this transformation with less detail. In particular, we select the intent using the description attribute and then describe merely why one of the mandatory properties is appropriate.

Select intent using description attribute The transformation *BuildRG* specifies the semantics of Place/Transition Petri Nets operationally, i.e., in an inplace fashion. This fits obviously to the description attribute of the *Simulation* intent in Table 5.

Check appropriateness of mandatory property ***[BP]***:

Let us call $t = (\mathsf{MM_s}, \mathsf{MM_t}, \mathsf{spec})$ the corresponding transformation specification. Since a simulation is in-place, $\mathsf{MM_s}$ and $\mathsf{MM_t}$ both represent a metamodel for Place / Transition Petri Nets. The specification follows a graph-based approach, using MoTif [117] as a model transformation language $\mathcal{L}$. The attached transformation execution $\mathsf{TS_t} = (\mathsf{S}, \longrightarrow)$ corresponds to the semantics of MoTif execution engine.

As shown in Fig. 16, it is possible in MoTif to specify the transformation rules (on the left, adapted from [72] for the purpose of the PWCS), but also their scheduling (on the right). Four rules compose the specification: `FindTr`, `ConsumeTks`, `NonFiringTr`, and `ProduceTks`. The rules are organized in two nested loops: the outermost, called Simulation, runs in an infinite loop. The first rule `FindTr` (which is a query consisting of solely a LHS) selects one transition. The transition found is assigned to a pivot variable *transition* to be referred by subsequent rules. Then,

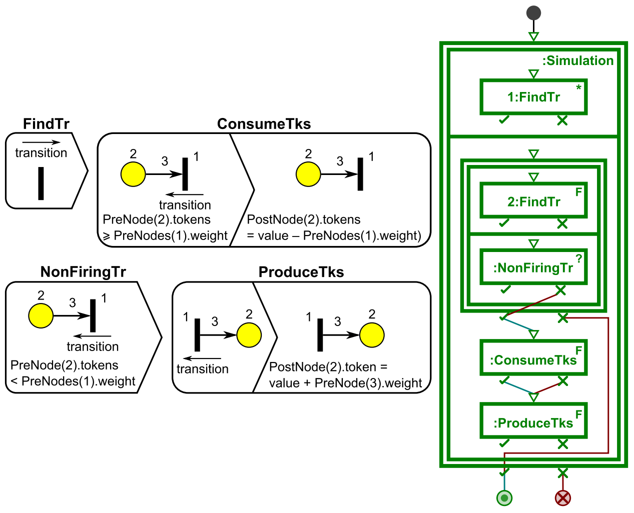


Fig. 16 Simulation of petri nets: transformation rules (*left*) and schedule (*right*)

the transformation ensures that only firing transitions will be processed. To find enabled transitions, the transformation iterates through all transitions until one has been found that does *not* satisfy the pattern of a *non*-firing transition. This is done by iterating over every transition in the model, and if the `NonFiringTr` rule cannot succeed, it is assigned to the pivot in order to fire the transition. This interruption in the inner loop is represented by connection from the fail port of the rule `NonFiringTr` to the success port of the enclosing rule block. When a firing transition is found, it is assigned the *transition* pivot, replacing the former transition. Then, tokens are transferred along this transition as depicted by rules `ConsumeTks` and `ProduceTks`. These two rules are applied for all adjacent arcs and places (denoted by an 'F'). After that, the first `FindTr` rule is applied again recursively, by re-matching the new model looking for a transition given the new marking. This control flow goes on until no more transitions are fireable. This transformation succeeds if the input model contains a transition and fails if not.

Outlook to validating mandatory property ***[BP]***: After having identified the intent for the *BuildRG* transformation, we want to validate as described in Fig. 5 if its mandatory/selected optional properties are indeed fulfilled. We give a brief idea of this process for the identified mandatory property **[BP]** as described above.

The **[BP]** mandatory intent property can be concretized in the following way. As defined in Definition 14, a **[BP]** depends on an input model. In our case, $\mathsf{M_i}$ is the Petri Net model illustrated in Fig. 9 of [80] that models the behavior of the power window control software. In particular, each place in that Petri Net must contain at most one token during its execution. Petri Nets of this kind are also called *1-safe*. Therefore, an indicator of the correctness of the *BuildRG* transformation that at each step of the simulation each place has at most one token, assuming of course the Petri Net model being simulated is indeed 1-safe. The following **[BP]** states that given that input Petri Net, the execution of the transformation from Fig. 16 will always satisfy that property ϕ expressed in LTL. Here, M denotes the marking of a place p in $\mathsf{M_i}$.

$$\forall p \in \mathsf{M_i}\,.\; \mathsf{M_i}, \mathsf{TS_t} \models \neg G\,(|M\,(p)| > 1)$$

Simulation transformations often include a loop where the same steps are re-executed on the resulting model. Furthermore, some steps may require choices to be done. Thus, a simulation execution consists of one branch in $\mathsf{TS_t}$. In our example, every loop of the simulation starts by looking for a non-firing transition. However, when found, only one such transition is taken into consideration. Therefore, to verify that a transformation does not satisfy ϕ, it suffices to check whether ϕ is not satisfied for one step in one simulation execution. Some approaches allow one to specify such invariants on the model transformation steps directly, such as in [130].

6.5 Overview

As a partial validation of our description framework, we applied the scenario described in Fig. 4: We identified the intents of transformations of the PWCS. In Table 6, the PWCS transformations are classified according to the intent they obey. As one can easily notice, some of our intents are not represented at all. This is not surprising however: The purpose of the PWCS transformation chain is to generate trustable C code for hardware execution; consequently, intents related to transformation visualization, synchronization, or syntactic manipulation, among others, have no corresponding transformation in the chain. On the contrary, some of the intents collect many transformations: Most of the transformations belong to either *Query*, *Refinement*, *Synthesis*, *Translation*, or *Simulation*.

In addition to the coarse-grained alignment, Tables 7 and 8 show the detailed results for the *Simulation* and *Transla-*

Table 6 Intents of transformations present in the PWCS

Intent	Transformations
Restrictive Query	`CheckReachableState, CheckContinuous, ExtractPerformance, CheckBinPacking, SearchArchitecture, SearchECU, SearchDetailed, CheckSchedulability, CheckDEVSTrace`
Refinement	`ArchitectureDeployment, ECUDeployment, DetailedDeployment`
Abstraction	`ExtractTimingBehaviour`
Synthesis	`SCToAUTOSAR, SwToC, ToInstrumented, GenerateCalibration, ArToMw, ArToRte`
Translation	`EnvToPN, PlantToPN, ScToPN, ControllerToSc, EnvToCBD, PlantToCBD, ToBinPackingAnalysis, ToSchedulabilityAnalysis, ToDeploymentSimulation`
Simulation	`BuildRG, SimulateHybrid, ExecuteCalibration, SimulateDEVS, CalculateSchedulability`
Composition	`CombinePN, CombineCBD, CombineCalibration, CombineC`

Table 7 Model transformation examples from the PWCS falling under the *Simulation* intent

Transformation	Description	Precond.	Mandatory	Optional
`BuildRG`	The BuildRG transformation simulates the execution of a Place/Transition Petri Net in order to build that net's reachability graph. Safety requirements for the power window can then be checked on the produced reachability graph	(1), (2), (3)	(1), (2), (3), (4)	(1), (2)
`SimulateHybrid`	This transformation simulates the interactions between the physical window and the designed window controller. While the physical window has continuous behavior, i.e., the window is moving up/down in a continuous manner, the user can push buttons to control the window that correspond to discrete signals. Casual Block Diagrams (CBD) representing the window behavior are co-simulated with Statecharts that represent the user events	(1), (2), (3), (4)	(1), (2), (3), (4)	(1), (2), (3)
`CalculateBinPacking`	The bin packing transformation is a simple transformation that simulates and evaluates the usage of a hardware component by calculating the sum of each execution time of a function mapped to the hardware component divided by the period of the functions. The transformation is implemented as an equation and produces measurements	(1), (2), (3), (4)	(1), (2), (3), (4)	(1), (2)
`ExecuteCalibration`	By running a simulation on a host computer, the input to execute an instrumented software application on the target platform for collecting measurements to obtain calibration parameters	(1), (2), (3), (4), (5)	(1), (2), (3), (4)	(1), (2)
`SimulateDEVS`	AUTOSAR models are translated into DEVS for producing traces by simulating the DEVS representations. The output of the DEVS simulations are traces that are further analyzed by a boolean formula	(1), (2), (3), (4), (5)	(1), (2), (3), (4)	(1), (2)

tion transformations identified in the PWCS. For each transformation, we describe what the transformation does and report on the satisfaction of the preconditions, and mandatory/optional properties. This gives an interesting snapshot on the applicability of our method in real-world transformation chains. Although both tables collect transformations with the same intent, the preconditions and even the mandatory properties are not the same for all transformations: In Table 7, these differences are due to the fact that preconditions (4) and (5) in the Simulation intent (Table 5) are considered optional; in Table 8, the differences in the mandatory properties come from the fact that some properties in the Translation intent (Table 3) are dependent on the translation type (bridge, simulation, or analysis).

6.6 Lessons learned on intent choice for the PWCS transformations

The decision on which intents are attributed to each transformation of the PWCS, as described in Table 6, was made by making use of the process described in Fig. 4. The step of *selecting an intent using the description attribute* in Fig. 4 was achieved by interviewing the transformation engineer. This was the starting point to the process as it is the transformation engineer who can best describe the role, or *intent*, of a given transformation in a model transformation chain. We observed that the intuitive *intent* from the engineer's viewpoint of the transformation is very important and provides the most accurate entry point into the identification and verification process.

We have also observed that the educational background of the transformation engineer influences the intuitive choice of an intent, as the keywords chosen by us for intents could be connotated with activities different than the ones we have associated with the keywords in our catalog. In fact, we observed the intent *description* attribute (as described in Fig. 2) is often quickly looked over in favor of a predefined notion of the keyword used to name the intent—as mentioned in Sect. 3.10 about the empirical evaluation of the catalog. As such, it is of the utmost importance the catalog reflects the common and intuitive under-

Table 8 Model transformation examples from the PWCS falling under the *Translation* intent

Transformation	Description	Precond.	Mandatory	Optional
`EnvToPN`	Build a Petri Net representation of a specialized model of the passenger's interactions with the powerwindow	None	(1), (2), (3), (7)	(2)
`SCToPN`	Build a Petri Net representation of a statechart model representing the powerwindow control software to allow checking power window security requirements	None	(1), (2), (3), (4)	(2)
`PlantToPN`	Build a Petri Net representation of a specialized model of the powerwindow physical configuration to allow checking power window security requirements	None	(1), (2), (3), (7)	(2)
`ControllerToSC`	Produce a statechart for providing semantics to a specialized model of the power window control flow	None	(1), (2), (3), (7)	(2)
`PlantToCBD`	Generate a causal block diagram (as python code) that can be used both for simulation of the combined system and for calibration of the combined system	None	(1), (2), (3), (6)	(2)
`EnvToCBD`	Generate a causal block diagram (as python code) that can be used both for simulation of the combined system and for calibration of the combined system	None	(1), (2), (3), (6)	(2)
`ToBinPackingAnalysis`	Build an equational algebraic representation of the dynamic behavior of the involved hardware components from an AUTOSAR [7] specification to allow checking processor load distribution	None	(1), (2), (3), (6)	(2)
`ToSchedulabilityAnalysis`	Build an equational algebraic representation of the dynamic behavior of the involved hardware and software components from an AUTOSAR specification to allow checking software response times	None	(1), (2), (3), (6)	(2)
`ToDeploymentSimulation`	Build a DEVS representation of the deployment solution to allow checking latency times, deadlocks and lost messages	None	(1), (2), (3), (6)	(2)

standing of the intent vocabulary as possible. Additionally, it became clear that the more detailed information is available about the transformation and the context in which it is used, the more straightforward intent choice becomes. As such, the *use_context* and the *preconditions* attributes play a fundamental role in accommodating subjective aspects of intent choice by the transformation engineer.

The formal notion of *intent*, as presented in Sect. 5, forces transformations to fit within certain formal ranges defined by the *mandatory* and *optional* properties, as well as by the *is_endogenous* and *is_exogenous* attributes. However, given a model transformation chain, a transformation's intent might depend on the granularity at which a transformation is observed. For example, the *EnvToPN* transformation in Fig. 11 might be seen as locally having the *translational semantics* intent, given the *Environment DLS* language does not have explicitely defined semantics. However, in the context of the *verification* area of Fig. 11, it might also be seen as having the *analysis* intent given that in that context, the goal of the transformation is to delegate the analysis of the model to a Petri net checker. This last remark points to the fact that groups of transformations may also have intents that fall within the range of our catalog. This is for example the case of the *verification* or *simulation* blocks in Fig. 11 that could be seen as generally having the *analysis* and *simulation* intents, respectively. This demonstrates the fact that studying intent composition together with corresponding verification needs is an interesting topic of future work.

We have also realized that instantiating the mandatory and optional formal properties (described in Sect. 4) into concrete ones is, for the time being, not a straightforward task. In order to perform this instantiation, we require property languages built for the used model transformation language, and for which a automated property checking tool is available. However, given the fact that the verification of model transformations is a recent domain currently under active investigation, there is at the moment of the writing of this paper a lack of standard tools that can verify multi-type properties of model transformations, as required by our approach.

In this study, we have used in Sect. 6.3 a property language adapted for proving the properties of transformations with the *translation* intent, defined in [82]. In Sect. 6.4, we have CTL to express a property for a transformation having the *simulation* intent. We are nonetheless convinced that more mature property languages and verification tools associated with specific model transformation languages and toolsets are required to make it such that property instantiation within our framework becomes feasible for the required range of properties identified for each intent. In this sense, the research presented in this document can be seen as a roadmap for the development and unification of current verification techniques for model transformations.

7 Related work

Our study investigated model transformation intents and identified relevant properties for each intent that should be verified. Thus, we discuss three lines of studies related to our work: (i) intents in software engineering, (ii) classifications of model transformations, and (iii) classifications of model transformation verification approaches.

7.1 Intents in software engineering

The notion of *intents* in software engineering is not new. In 1994, Yu and Mylopoulos [134] realized that research in this area was, at the time, more focused on design and implementation of software—the *what* and the *how*—rather than on the requirements necessary to understand the software to improve the underlying development processes—the *why*. MDE is following a similar path: Research has been more devoted to the different modeling and transformation activities rather than exploring the intents behind such activities.

Two studies [70,90] investigated the rationale (i.e., purpose or *intent*) behind modeling artifacts. Kühne [70] identified two modeling intents based on the relationship between the modeled artifacts and their representative models: *token* models “project and translate” artifacts from the reality, and *type* models that additionally perform an “abstraction” step from the artifacts to represent universal aspects. Recently, Muller et al. [90] explored the relationship between artifacts and their symbolic representations, using intention as a core constituent to the modeling activity. The intents discussed in the two former studies are among the intents presented in this paper, besides other additional intents that we investigate using our Intent/Property mapping (Sect. 2.1).

In the field of requirements models, requirements patterns have been proposed to facilitate requirements analysis [133]. Similar to transformation intents, requirements patterns are high-level descriptions of the properties that the implementation should possess. A key difference is that our notion of intents focuses on model transformations used in MDE, whereas requirements patterns have a much wider scope and are not tailored to the intricacies of a specific domain.

Finally, Amrani et al. [4] presented a preliminary version of this work focusing on the *Analysis* intent. In this study, we present three major additions to our work in [4]: (1) we provide an intent catalog that summarizes many of the intents discussed in the literature, (2) we provide an updated definition of the *Analysis* intent, and (3) we investigate four other intents in depth (i.e., query, refinement, translation, and simulation).

7.2 Classifications of model transformations

Several studies [26,57,85,119,129] proposed different classifications of model transformations based on different transformation aspects. Mens and Van Gorp [85] provided a multidimensional taxonomy of transformations based on aspects related to the manipulated models (e.g., the abstraction level of the transformation’s input and output models) and the used transformation execution strategies (e.g., in-place and out-place transformations). The classification dimensions are illustrated on transformations that can be grouped according to our intents. In our study, we investigate well-known *uses* of transformations, propose fourteen additional intents to seven intents presented in [85], and discuss several intent properties. In [57], design patterns for model transformations expressed in QVT Relations are presented, but the intents behind the transformations are not discussed.

Tisi et al. [119] examined higher-order transformations, i.e., transformations manipulating transformations. They classified them based on whether their input and/or output models are transformations or not, resulting in four combinations: *synthesis* produces a transformation from a non-transformation; *analysis* takes an input transformation and produces a non-transformation output; *(de-)composition* uses multiple transformations both in input and output; and *modification* takes an input transformation and the produces a modified version of the input as an output. 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.

Czarnecki and Helsen [26] classified the features of transformation languages by establishing a feature model. To do so, they introduced five intended applications of transformations which are also covered in our transformation intent catalog.

A taxonomy of program transformations is presented by Visser [129]. Instead of proposing a taxonomy of multiple dimensions as in [85], Visser employed 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*. Other nodes in the taxonomy are covered in our intent catalog. Moreover, we present several intents that are specific to model transformations.

To sum up, our 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) and properties of interest for the given intent are proposed. To the best of our knowledge, the latter aspect has not been previously investigated.

7.3 Classifications of model transformation verification approaches

Several studies proposed classifications of formal verification approaches of model transformations [5,20,41,98]. In [5], we presented a tri-dimensional space for classifying transformation verification approaches where the three dimensions were transformation language, verification property, and verification technique. Furthermore, these three dimensions have been also reused in [20] to derive the state-of-the-art in model transformation verification. Lukman and Whittle [98] have classified model transformation verification approaches with respect to the general approach used (e.g., testing, theorem proving, and model-checking) and investigated the approaches with respect to the three dimensions in [5] (i.e., transformation language, verification property, and verification technique).

Gabmeyer et al. [41] presented a feature model for the classification of verification approaches of software models that can be leveraged for the classifications of model transformation verification approaches by considering transformations as models. The three dimensions used in [5] correspond on a general level to the main features of the feature model presented by Gabmeyer et al. in [41].

This study adds another dimension to the formerly mentioned classifications of model transformation verification approaches: the model transformation intent. Besides being another dimension in the classification, we believe that the dimension of transformation intent is the key in identifying all the other dimensions.

8 Conclusion

The long-term goal of our work is to facilitate the use of model transformations in industry in general, but also to pave the way to the efficient development of verified transformations for safety-critical applications. This paper summarizes the results of our work on intents and their properties for model transformations to capture transformation goals and requirements and simplify the process of transformation specification, development, reuse, maintenance, validation, and verification.

The paper builds on our previous work in [4,5], but extends it significantly and makes the following contributions:

- The description framework for transformation intents first proposed in [4] is extended and described in detail; in the framework, intents are described using, among other attributes, the properties that are relevant for them (Sect. 2).
- The preliminary intent catalog from [4] is extended to 25 intents and presented in more detail; for each intent, at least one sample transformation from the literature is given. In addition, the intents are now structured using a hierarchical classification scheme. The catalog is the result of a thorough literature review and was built to encompass the most frequently occurring intents (Sect. 3).
- The intent catalog has been empirically evaluated with respect to correctness, ambiguity, and completeness based on a survey of 38 transformation creators. The results indicate that there is substantial agreement that our intent descriptions are correct and unambiguous. Furthermore, no participant found the set of intents to be incomplete. While more evaluation is needed, the results suggest that the current catalog is good beginning and has the potential to bring value to the community.
- A list of relevant properties is identified in the form of intent properties, which are general descriptions of properties on varying levels of abstraction. High-level formalizations of these properties are presented (Sect. 4).
- The framework is evaluated and illustrated extensively by using it to describe six common intents in detail (Sect. 5) and by applying it to the Power Window Case Study (Sect. 6); the intents of the more than 30 transformations in the case study are identified, and for two transformations, concrete examples of mandatory properties are provided by concretizing the intent properties into transformation properties.

8.1 Future work

The work presented in this paper has many limitations that could be addressed in future work.

The detailed description of 16 of the intents identified in Sect. 3 using our framework still needs to be carried out. Also, the description of other transformations and their transformation properties in the case study could be attempted. These uses of the framework may reveal the need for additional intents or intent properties leading to an extension of the framework. In particular, making intent properties “real-time-sensitive" would be interesting, not just for automotive, but for safety-critical, real-time software in general.

On the more long-term, yet practical side, the true utility of our notion of intent for industrial model-driven software development remains to be determined. Input from industrial users of MDE might be helpful here.

On the more theoretical side, Sect. 2.3.3 has already discussed the relevance of our work to research on the, possibly "intent-specific," specification, implementation, and analysis of model transformations. In this context, it would be interesting to explore the potential connections with recent work on the formal specification, testing, and formal verification of model transformations [17,52,81,82,86,87,125]. For example, to what extent can existing techniques be used or extended to help developers verify the properties associated with a transformation's intent?

Finally, our case study illustrates that model transformations in MDE are typically composed to achieve some higher-level goal in the sense of goal-oriented requirements engineering (GORE) [73]. In GORE, the software to be developed and its environment are thought of as consisting of goal-seeking, cooperating agents. Goal modeling is used to identify and justify requirements which are goals whose achievement is the responsibility of a particular software agent. Since model transformations are typically the responsibility of particular agents, requirements in GORE appear to correspond to our model transformation intents. For example, it is conceivable that the PM in Fig. 11 is the result of the refinement of high-level goal "construct verified control software for a power window" into a suitable composition of subgoals, softgoals (often also called "non-functional requirements"), and intents, some of which are then operationalized into model transformations. A more concrete example is the realization of the safety analysis in the case study as shown in Fig. 12 in which the DSL models are first translated into Petri Nets using an abstraction and then combined for the construction of the reachability graph and the check of the CTL formula. These examples suggest that parts of the work on GORE might be applicable to MDE and that requirements engineering for MDE in general, and GORE for MDE in particular, might be fruitful topics for further research.

References

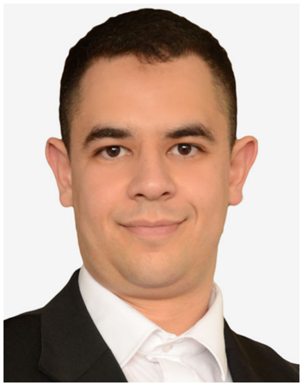
1. Adrion, W.R., Branstad, M.A., Cherniavsky, J.C.: Validation, verification, and testing of computer software. ACM Comput. Surv. **14**(2), 159–192 (1982)
2. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Softw. Syst. Model. **5**(3), 261–288 (2006)
3. Akehurst, D., Kent, S.: A relational approach to defining transformations in a metamodel. In: Proceedings of the 5th International Conference on the Unified Modeling Language (UML), pp. 243–258. Springer, Berlin (2002)
4. Amrani, M., Dingel, J., Lambers, L., Lúcio, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Towards a model transformation intent catalog. In: Proceedings of the First Workshop on Analysis of Model Transformations (AMT), pp. 3–8. ACM, New York (2012)
5. Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y., Cordy, J.R.: A tridimensional approach for studying the formal verification of model transformations. In: Proceedings of the First Workshop on Verification and Validation of Model Transformations (VOLT), pp. 921–928. IEEE Computer Society (2012)
6. Asztalos, M., Syriani, E., Wimmer, M., Kessentini, M.: Simplifying model transformation chains by rule composition. In: Models in Software Engineering—Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers, LNCS, vol. 6627, pp. 293–307 (2011)
7. AUTOSAR: http://www.autosar.org (2010)
8. Bae, J.H., Lee, K., Chae, H.S.: Modularization of the UML metamodel using model slicing. In: Proceedings of the Fifth International Conference on Information Technology (ITNG), pp. 1253–1254. IEEE (2008)
9. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Style-based modeling and refinement of service-oriented architectures. Softw. Syst. Model. **5**, 187–207 (2006)
10. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT), LNCS, vol. 6707, pp. 167–182. Springer, Berlin (2011)
11. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (Models) (2006)
12. Bézivin, J., Rumpe, B., Tratt, L.: Model transformation in practice workshop announcement. http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf (2005)
13. Biermann, E., Ehrig, K., Ermel, C., Hurrelmann, J.: Generation of simulation views for domain specific modeling languages based on the eclipse modeling framework. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 625–629. IEEE Computer Society (2009)
14. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Berlin (2003)
15. Mc Brien, P., Poulovassi, A.: Automatic migration and wrapping of database applications—a schema transformation approach. In: Akoka, J., Bouzeghoub, M., Comyn Wattiau, I., M'etais, E. (eds.) Proceedings of the International Conference on Conceptual Modeling (ER), LNCS, vol. 1782, pp. 99–114. Springer, Berlin (1999)
16. Bruneliére, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J.: Towards model driven tool interoperability: bridging eclipse and microsoft modeling tools. In: Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA), (LNCS), vol. 6138, pp. 32–47 (2010)
17. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM), (LNCS), vol. 7635, pp. 198–213. Springer, Berlin (2012)
18. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (Models), pp. 432–448. Springer, Berlin (2012)
19. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. J. Syst. Softw. **83**, 283–302 (2010)

20. Calegari, D., Szasz, N.: Verification of model transformations: a survey of the state-of-the-art. Electron. Notes Theor. Comput. Sci. **292**, 5–25 (2013)
21. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. ECEASST 24 (2009)
22. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: Proceedings of the International IEEE Enterprise Distributed Object Computing Conference (EDOC), pp. 222–231. IEEE Computer Society (2008)
23. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development. Ceteva, Sheffield (2004)
24. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
25. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti Oliet, N., Meseguer, J., Talcott, C.: All About Maude. A High-Performance Logical Framework, LNCS, vol. 4350. Springer, Berlin (2007)
26. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
27. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-based testing in practice. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 285–294. ACM Press, New York (1999)
28. Deltombe, G., Le Goaer, O., Barbier, F.: Bridging KDM and ASTM for model-driven software modernization. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 517–524 (2012)
29. Denil, J.: Design, verification and deployment of software-intensive systems: a multi-paradigm modelling approach. Ph.D. thesis, Universiteit Antwerpen (2013). http://msdl.cs.mcgill.ca/people/joachim/academic
30. Denil, J., Cicchetti, A., Biehl, M., De Meulenaere, P., Eramo, R., Demeyer, S., Vangheluwe, H.: Automatic deployment space exploration using refinement transformations. In: Amaral, V., Hardebolle, C., Vangueluwe, H., Lengyel, L., Bunus, P. (eds.) Recent Advances in Multi-paradigm Modelling, vol. 50, Electronic Communications of the EASST, Berlin (2012)
31. Dorf, R.C.: Modern Control Systems, 12th edn. Addison-Wesley Longman Publishing Co., Inc, Boston (2011)
32. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. CAD Integr. Circuits Syst. **27**(7), 1165–1178 (2008)
33. Ducasse, S., Gîrba, T.: Using smalltalk as a reflective executable meta-language. In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (Models), pp. 604–618 (2006)
34. Engel, K.D., Paige, R., Kolovos, D.: Using a model merging language for reconciling model versions. In: Rensink, A., Warmer, J. (eds.) Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications (ECMFA), LNCS, vol. 4066, pp. 143–157. Springer, Berlin (2006)
35. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Proceedings of the 3rd International Conference on the Unified Modeling Language (UML), pp. 323–337 (2000)
36. Ermel, C., Ehrig, H.: Behavior-preserving simulation-to-animation model and rule transformations. Electron. Notes Theor. Comput. Sci. **213**(1), 55–74 (2008)
37. Del Fabro, M.D., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Softw. Syst. Model. **8**(3), 305–324 (2009)
38. Fischer, T., Niere, J., Turunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modelling language and java. In: Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations (TAGT), LNCS, vol. 1764, pp. 296–309. Springer, Berlin (2000)
39. Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychol. Bull. **76**(5), 378–382 (1971)
40. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
41. Gabmeyer, S., Brosch, P., Seidl, M.: A classification of model checking-based verification approaches for software models. In: Proceedings of the 2nd International Workshop on the Verification of Model Transformation (VOLT) (2013)
42. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
43. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 query/views /transformations submissions and recommendations towards the final standard. In: Proceedings of the MetaModelling for MDA Workshop, pp. 178–197 (2003)
44. Gargantini, A., Riccobene, E., Scandurra, P.: Combining formal methods and MDE techniques for model-driven system design and analysis. JAS **3**(1–2), 1–18 (2010)
45. Gessenharter, D.: Mapping the UML2 semantics of associations to a java code generation model. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), LNCS, vol. 5301, pp. 813–827. Springer, Berlin (2008)
46. Giese, H., Levendovszky, T., Vangheluwe, H.: Summary of the workshop on multi-paradigm modeling: concepts and tools. In: Models in Software Engineering: Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers, LNCS, vol. 4364. Springer, Berlin (2007)
47. Gogolla, M., Vallecillo, A.: Tractable model transformation testing. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA), LNCS, vol. 6698, pp. 221–235. Springer, Berlin (2011)
48. Griswold, W.G.: Program restructuring as an aid to software maintenance. Ph.D. thesis, University of Washington (1991)
49. Object Management Group: MDA Guide (version 1.0.1) (2003)
50. Object Management Group: Mof Qvt: Query/View/Transformation (2008)
51. Object Management Group: Unified Modeling Language (UML) 2.4.1 Superstructure (2011)
52. Guerra, Esther, de Lara, Juan, Wimmer, Manuel, Kappel, Gerti, Kusel, Angelika, Retschitzegger, Werner, Schönböck, Johannes, Schwinger, Wieland: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. **20**(1), 5–46 (2013)
53. Guerra, E., de Lara, J.: Model view management with triple graph transformation systems. In: Proceedings of the International Conference on Graph Transformation (ICGT), LNCS, vol. 4178, pp. 351–366. Springer, Berlin (2006)
54. Guerra, E., de Lara, J.: Event-driven grammars: relating abstract and concrete levels of visual languages. J. Softw. Syst. Model. **6**(6), 317–347 (2007)
55. Harel, D., Rumpe, B.: Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Tech. Rep, Weizmann Institute of Sience (2000)
56. Harman, M., Binkley, D., Danicic, S.: Amorphous program slicing. In: Software Focus, pp. 70–79. IEEE Computer Society Press (1997)
57. Iacob, M.E., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Proceedings of EDOCW'08, pp. 1–10 (2008)
58. Izquierdo, J.L.C., Molina, J.G.: Extracting models from source code in software modernization. Software and Systems Modeling (SoSyM), pp. 1–22 (2012)

59. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering. GPCE '06, pp. 249–254. ACM, Portland (2006)
60. Kastenberg, H., Rensink, A.: Model checking dynamic states in groove. In: Model Checking Software (Spin), LNCS, vol. 3925, pp. 299–305. Springer, Berlin (2006)
61. Kelsen, P., Ma, Q., Glodt, C.: Models within models: taming model complexity using the sub-model lattice. In: Proceedings of the International Conference on the Foundational Approaches to Software Engineering (FASE), pp. 171–185. Springer, Berlin (2011)
62. Kern, H.: The Interchange of (meta)models between MetaEdit+ and eclipse EMF using M3-level-based bridges. In: Proceedings of the International Workshop on Domain-Specific Modeling (DSM) (2009)
63. Kern, H., Hummel, A., Kühne, S.: Towards a comparative analysis of meta-metamodels. In: Proceedings of the International Workshop on Domain-Specific Modeling (DSM) (2011)
64. Kern, H., Kühne, S.: Model interchange between ARIS and eclipse EMF. In: Proceedings of the International Workshop on Domain-Specific Modeling (DSM) (2007)
65. Kern, H., Kühne, S.: Integration of microsoft visio and eclipse modeling framework using M3-level-based bridges. In: Proceedings of the ECMDA Workshop on Model-Driven Tool & Process Integration (2009)
66. Kilov, H.: From semantic to object-oriented data modeling. In: Proceedings of the First International Conference on System Integration, pp. 385–393 (1990)
67. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
68. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-based web engineering. Web Engineering: Modelling and Implementing Web Applications, pp. 157–191 (2008)
69. König, B., Kozioura, V.: Augur 2-A new version of a tool for the analysis of graph transformation systems. Electron. Notes Theor. Comput. Sci. **211**, 201–210 (2008)
70. Kühne, T.: Matters of (meta-) modeling. Softw. Syst. Model. **5**, 369–385 (2006)
71. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Systematic transformation development. EcEasst 21 (2009)
72. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: Models in Software Engineering—Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers, Lncs, vol. 6002, pp. 240–255. Springer, Berlin (2010)
73. Axel van Lamsweerde: Goal-oriented requirements engineering: a guided tour. In: Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE) (2001)
74. Lano, K., Kolahdouz Rahimi, S.: Slicing techniques for UML models. JOT **10**, 1–49 (2011)
75. de Lara, J., Guerra, E., Boronat, A., Heckel, R., Torrini, P.: Graph transformation for domain-specific discrete event time simulation. In: Proceedings of the 5th International Conference on Graph Transformations (ICGT), Lncs, vol. 6372, pp. 266–281. Springer, Berlin (2010)
76. de Lara, J., Taentzer, G.: Automated model transformation and its validation using AToM3 and AGG. Diagrammatic Representation and Inference (Diagrams), pp. 182–198 (2004)
77. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. J. Vis. Lang. Comput. **15**(3–4), 309–330 (2004)
78. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Additional material for the paper "Model Transformation Intents and Their Properties". http://msdl.cs.mcgill.ca/people/levi/transformation_intents/material (2014)
79. Lúcio, L., Denil, J., Mustafiz, S., Vangheluwe, H.: An overview of model transformations for a simple automotive power window. Tech. Rep. SOCS-TR-2012.1, McGill University (2012)
80. Lúcio, L., Mustafiz, S., Denil, J., Vangheluwe, H., Jukss, M.: FTG+PM: an integrated framework for investigating model transformation chains. In: System Design Languages (Sdl) Forum: Model-Driven Dependability Engineering, LNCS, vol. 7916, pp. 182–202. Springer, Berlin (2013)
81. Lúcio, L., Vangheluwe, H.: Model transformations to verify model transformations. In: Proceedings of the 2nd Workshop on Verification of Model Transformations (VOLT) (2013)
82. Lúcio, L., Vangheluwe, H.: Symbolic execution for the verification of model transformations. Tech. Rep. SOCS-TR-2013.2, McGill University (2013). http://msdl.cs.mcgill.ca/people/levi/files/MTSymbExec.pdf
83. Mannadiar, R., Vangheluwe, H.: Modular synthesis of mobile device applications from domain-specific models. In: Model-Based Methodologies for Pervasive and Embedded Software Workshop (2010)
84. Mayerhofer, T., Langer, P., Wimmer, M.: Towards xMOF: executable DSMLs based on fUML. In: Proceedings of the 12th Workshop on Domain-Specific Modeling (DSM'12) (2012)
85. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
86. Selim, G.M.K., Cordy, J.R., Dingel, J.: Analysis of model transformations. Tech. Rep. 2012–592, Queen's University (2012)
87. Selim, G.M.K., Cordy, J.R., Dingel, J.: Model transformation testing: the state of the art. In: Proceedings of the 1st International Workshop on the Analysis of Model Transformations (AMT) (2012)
88. Mosterman, P.J., Vangheluwe, H.: Computer automated multi-paradigm modeling: an introduction. Simulation **80**(9), 433–450 (2004)
89. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS, vol. 3713, pp. 264–278. Springer, Berlin (2005)
90. Muller, P.A., Fondement, F., Baudry, B., Combemale, B.: Modeling modeling modeling. Softw. Syst. Model. **11**, 1–13 (2010)
91. Muller, P.A., Hassenforder, M.: HUTN as a bridge between ModelWare and grammarWare—an experience report. In: Proceedings of the Workshop in Software Model Engineering Wisme (2005)
92. Mustafiz, S., Denil, J., Lúcio, L., Vangheluwe, H.: The FTG+PM framework for multi-paradigm modelling: an automotive case study. In: Proceedings of the Multi-Paradigm Modelling Workshop (MPM). ACM (2012)
93. Narayanan, A., Karsai, G.: Towards verifying model transformations. Electron. Notes Theor. Comput. Sci. **211**, 191–200 (2008)
94. Narayanan, A., Karsai, G.: Verifying model transformations by structural correspondence. Electronic Communications of the European Association of Software Science and Technology (EASST) 10 (2008)
95. Padberg, J.: Categorical approach to horizontal structuring and refinement of high-level replacement systems. Appl. Categ. Struct. **7**, 371–403 (1999)
96. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Refinement via consistency checking in MDA. Electron. Notes Theor. Comput. Sci. **137**(2), 151–161 (2005)
97. Peterson, J.: Petri nets. ACM Comput. Surv. **9**(3), 223–252 (1977)
98. Rahim, L.A., Whittle, J.: A survey of approaches for verifying model transformations. Software and Systems Modeling (SoSym), pp. 1–26 (2013)

99. Rensink, A.: The groove simulator: a tool for state space generation. In: Proceedings of the Second International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE), LNCS, vol. 3062, pp. 479–485 (2003)
100. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: Workshop Proceedings of WRLA'10 @ ETAPS'10, pp. 174–190 (2010)
101. Rivera, J., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with maude. In: Proceedings of the International Conference on Software Language Engineering (SLE), pp. 54–73. Springer, Berlin (2009)
102. Romero, J.R., Rivera, J.E., Duran, F., Vallecillo, A.: Formal and tool support for model driven engineering with maude. J. Object Technol. **6**(6), 187–207 (2007)
103. Kolovos, D.S., Paige, R.F., Polack, F.: The epsilon object language (EOL). In: Proceedings of ECMDA-FA'06, pp. 128–142 (2006)
104. Schätz, B., Holzl, F., Lundkvist, T.: Design-space exploration through constraint-based model-transformation. In: Proceedings of the international conference and workshops on engineering of computer based systems, ECBS'10, pp. 173–182 (2010)
105. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Proceedings of the Third European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA'07), LNCS, vol. 4530, pp. 157–171. Springer, Berlin (2007)
106. Scholz, P.: A refinement calculus for statecharts. In: E. Astesiano (ed.) Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), LNCS, vol. 1382, pp. 285–301. Springer, Berlin (1998)
107. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model pruning. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (Models), pp. 32–46. Springer, Berlin (2009)
108. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
109. Shannon, R., Johannes, J.D.: Systems simulation: the art and science. IEEE Trans. Syst. Man Cybern. **6**(10), 723–724 (1976)
110. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development—Technology, Engineering, Management. Wiley, London (2006)
111. Van der Straeten, R., Jonckers, V., Mens, T.: A formal approach to model refactoring and model refinement. Softw. Syst. Model. **6**, 139–162 (2007)
112. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. In: Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT), LNCS, vol. 5063, pp. 91–106. Springer, Berlin (2008)
113. Syriani, E.: A multi-paradigm foundation for model transformation language engineering. Ph.D. thesis, McGill University (2011)
114. Syriani, E., Ergin, H.: Operational semantics of UML activity diagram: an application in project management. In: Proceedings of RE 2012 Workshops. IEEE (2012)
115. Syriani, E., Gray, J., Vangheluwe, H.: Modeling a model transformation language. In: Domain Engineering: Product Lines, Conceptual Models, and Languages. Springer, Berlin (2012)
116. Syriani, E., Vangheluwe, H.: DEVS as a semantic domain for programmed graph transformation, chap. 1, pp. 3–28. CRC Press, Boca Raton (2010)
117. Syriani, E., Vangheluwe, H.: A modular timed model transformation language. Softw. Syst. Model. **11**, 1–28 (2011)
118. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: AToMPM: a web-based modeling environment. In: MODELS'13 Demonstrations. CEUR (2013)
119. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Proceedings of the 5th European Conference on Model Driven Architecture—Foundations and Applications ECMDA-FA, LNCS, vol. 5562, pp. 18–33. Springer, Berlin (2009)
120. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 4424, pp. 632–647. Springer, Berlin (2007)
121. Tratt, L.: Model transformation and tool integration. Softw. Syst. Model. **4**, 112–122 (2005)
122. Tri, D., Tho, Q.: Systematic diagram refinement for code generation in SEAM. In: Proceedings of the Fourth International Conference on Knowledge and Systems Engineering (KSE), pp. 203–210. IEEE (2012)
123. Troya, J., Rivera, J.E., Vallecillo, A.: On the specification of non-functional properties of systems by observation. In: Models in Software Engineering, Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers, LNCS, vol. 6002, pp. 296–309. Springer, Berlin (2010)
124. Troya, Javier, Vallecillo, Antonio, Durán, Francisco, Zschaler, Steffen: Model-driven performance analysis of rule-based domain specific visual models. Inf. Softw. Technol. **55**(1), 88–110 (2013)
125. Vallecillo, A., Gogolla, M.: Typing model transformations using tracts. In: Proceedings of the 5th International Conference on the Theory and Practice of Model Transformations (ICMT), LNCS, vol. 7307, pp. 56–71. Springer, Berlin (2012)
126. Varró, D., Varró Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination analysis of model transformations by petri nets. In: Proceedings of the International Conference on Graph Transformations (ICGT), pp. 260–274. Springer, Berlin (2006)
127. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E.B. (eds.) Proceedings of the UML'03 Workshop CSDUML 2003: Critical Systems Development in UML, TUM-I0323, pp. 63–78. Technische Universität München (2003)
128. Viehstaedt, G., Minas, M.: DiaGen: a generator for diagram editors based on a hypergraph model. In: Proceedings of the International Workshop on Next Generation Information Technologies and Systems, pp. 155–162 (1995)
129. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symb. Comput. **40**, 831–873 (2005)
130. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Right or wrong?—Verification of model transformations using colored petri nets. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM) (2009)
131. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Proceedings of MoDELS Satellite Events, pp. 159–168 (2005)
132. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. Electron. Notes Theor. Comput. Sci. **211**, 159–170 (2008)
133. Withall, S.: Software Requirement Patterns. Microsoft Press, Redmond (2007)
134. Yu, E.S., Mylopoulos, J.: Understanding "Why" in software process modelling, analysis, and design. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 159–168 (1994)
135. Zhang, J., Lin, Y., Gray, J.: Generic and domain-specific model refactoring using a model transformation engine. In: Volume II of Research and Practice in Software Engineering, pp. 199–218. Springer, Berlin (2005)

Levi Lúcio is a Research Associate with the Modelling, Simulation and Design Laboratory of McGill University. He received his PhD from the University of Geneva, Switzerland, in 2008. His research is about bridging software engineering and formal techniques. Some of his concrete areas of interest are model-driven development, model transformation languages, the verification of model transformations, correctness-by-construction, models of concurrency (in particular Algebraic Petri Nets), model evolution, model-based testing, and tool construction. Together with several PhD students, he is currently developing a suite of techniques and tools for the verification of model transformations for the automotive industry.

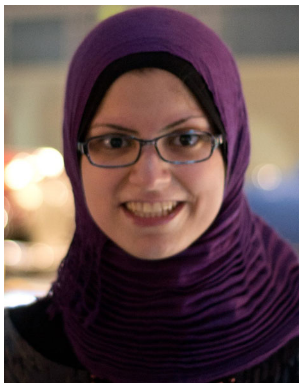


Moussa Amrani is currently a postdoc at the University of Namur (Belgium), working on the analysis of model transformations that include real-time features. He recently received his PhD from University of Luxembourg for a dissertation entitled "Formal Verification of Model Transformation—An Application to Kermeta" that addressed two challenges: providing a methodological framework for helping model transformation designers to ensure the correctness of their transformation; and proposing formal verification techniques for Kermeta, a popular object-oriented model transformation framework used in industrial projects. Before his PhD, Moussa Amrani received a BSc, a MSc, and a Magistere in Computer Science from University Joseph Fourier (Grenoble, France) where his work was dedicated to the formal verification of object-oriented programming languages like Java, using popular techniques such as model-checking and abstract interpretation. His main areas of expertise are model-driven engineering, formal verification techniques, and formal semantics.

Juergen Dingel received an MSc from Berlin University of Technology in Germany and a PhD in Computer Science from Carnegie Mellon University (2000). He is an Associate Professor in the School of Computing at Queen's University where he leads the Modeling and Analysis in Software Engineering group. His research interests include model-driven engineering, formal methods, and software engineering.

Leen Lambers is a postdoctoral researcher working on the DFG-project CorMoranT (Correct Model Transformations) in the group of Prof. Holger Giese at the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam since January 2010. She received her PhD for her dissertation "Certifying Rule-Based Models using Graph Transformation" at the Technical University of Berlin in December 2009, where she has been a scientific assistant in the group of Prof. Hartmut Ehrig from October 2003 until December 2009. Her main research focus is formal modeling and analysis in software engineering, in particular, using graph transformation. She has spent several research periods in the group of Prof. Mauro Pezzé at the University of Milano Bicocca and in the group of Prof. Fernando Orejas at the Technical University of Catalonia. She served as PC co-chair for the International Workshop on Graph Transformation and Visual Modeling Techniques and for the International Workshop on Verification of Model Transformation. She served as PC member for the International Conference on Model Transformation, the International Conference on Graph Transformation, and the International Conference on Software Maintenance and Evolution.

Rick Salay is a NECSIS research associate in the Department of Computer Science of the University of Toronto, working in the Software Engineering Group with Prof. Marsha Chechik. He received a B.A.Sc. and M.A.Sc. in Systems Design Engineering from University of Waterloo (1991) and a PhD in Computer Science from the University of Toronto (2010) with Prof. John Mylopoulos. His research focus is on developing formal theories about non-formal concepts such as modeler intent and modeler uncertainty in order to provide a foundation for tool support that will help software engineering practioners. Prior to his PhD, he had a 15 year career in advanced software product development holding various senior software design roles, most recently as chief architect at InSystems Technologies Inc. (now Oracle).



Gehan M. K. Selim received an MSc from Cairo University (Faculty of Computers and Information) in Egypt and is currently a PhD candidate in the School of Computing of Queen's University in Canada. Her research interests include model transformations, testing of model transformations, and formal verification of model transformations.

Eugene Syriani At the time of the paper, Eugene Syriani was an Assistant Professor in Computer Science at the University of Alabama. He currently holds the same position at l'UniversitŽ de MontrŽal. He received his PhD in Computer Science in 2011 and holds a BSc in Mathematics and Computer Science from 2006, both at McGill University. He also pursued postdoctoral research on model transformation in the Canada-wide NECSIS project on model-driven engineering for automotive systems. He mainly teaches software engineering courses at the undergraduate and doctoral level. His main research interests are in model-based design, in particular model transformation design and verification, model-driven methodology, simulation-based design, and application of MDE in non-computer science domains. He serves on the program committee and organizes several international conferences/workshops and is a reviewer for many international journals in modeling and in simulation. Eugene has worked in service-oriented software companies as a software engineer in Canada for a decade. He is also a member of the ACM and IEEE societies.

Manuel Wimmer is postdoctoral researcher at the Business Informatics Group of the Vienna University of Technology. Previously, he has been working as visiting researcher at the Software Engineering Group of the University of Málaga. He is/was involved in several national and international projects dealing with the foundations of model engineering techniques such as modeling languages and model transformations as well as with the application of these techniques for domains, such as tool interoperability, model versioning, cloud computing, and Web engineering.