Towards Safe Model Transformation for Constraint-driven Modeling

Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, and Alexander Egyed
Johannes Kepler University (JKU)
Linz, Austria
Email: {firstname.lastname}@jku.at

***Abstract*—Model transformation is a key technology in model-driven engineering. Unfortunately, transformations are typically written manually and thus they are likely to contain errors and to produce incorrect or undesired output. Safe transformation is the guarantee that for every possible source model, the transformed target model is valid with respect to user-definable well-formedness criteria. This paper introduces safe transformation for constraint-driven modeling (CDM), an approach that employs model transformation to generate target model constraints instead of generating a target model directly. Safe transformation for CDM ensures that transformations only produce correct and non-contradictory constraints. We demonstrate the feasibility of safe transformation and present a formal framework for applying it to CDM in arbitrary domains.**

I. Introduction

With the increasing popularity of *Model-Driven Engineering (MDE)* and modeling languages such as the *Unified Modeling Language (UML)*, there is also an increasing need for validating the correctness of design models as they are used for safety-critical systems or other systems that are subject to strong regulations [1]. This verification consists of checking whether a model is syntactically and semantically correct. The desired conditions are usually expressed in form of model constraints, using constraint languages such as the *Object Constraint Language (OCL)*.To date, there exist efficient approaches for checking constraints (e.g., [2]) on design models. However, these approaches are not readily applicable to *model transformations* [3] where both the source model and the target model must adhere to defined syntactic and semantic constraints. Of course, it is possible to verify a target model once it has been generated. However, what if we want to ensure that all generated target models are correct? The dilemma is that there are likely infinite possible source models and thus it is unscalable to generate all possible target models and verify them.

Safe model transformation ensures that transformations always produce results that meet defined criteria. These criteria are typically defined by designers based on either a specific metamodel or domain knowledge. For example, an important criterion may be that every transformation result is syntactically and semantically correct with respect to the target metamodel – i.e., the transformation will not generate an ill-formed target model. This implies that the applied criteria are likely to differ for different metamodels and domains. Safe model transformation is thus a warranty the transformation designer would like to provide to the transformation user (software engineer) – that the transformation result will always be correct with respect to the defined criteria, *regardless of the given input model*.

This work demonstrates safe transformation on *Constraint-driven Modeling (CDM)* [4]. CDM avoids premature design decisions by constraining target models instead of generating them directly. CDM uses model transformations to automate the generation of model constraints and to perform required updates. Hence, designers only have to write and maintain transformation rules instead of managing specific model constraints. However, transformation rules in CDM—and most other MDE-approaches—are written manually. This means that there is always a chance that these rules contain errors. CDM thus benefits from safe transformation as much as any other transformation-based approach would. Since CDM transforms source models into constraints, an incorrect transformation may result in contradictory constraints which make the problem of finding a valid model unsatisfiable. As an example, imagine two constraints forcing a model element to have two different values at the same time. Safe transformation for CDM ensures that defined transformation rules for a given source metamodel will never lead to contradictory constraints. This is a satisfiability test for which we apply the formal modeling and reasoning engine FORMULA [5]. Safe transformation generally ensures that all possible target models meet *structural* and *static semantical conformance criteria* [6]. For CDM, for instance, it ensures that no unsatisfiable constraints can be generated.

In this paper we present a formal approach for safe model transformation. The approach is generic and validates the correctness of arbitrary transformations on arbitrary metamodels and models by using a formal reasoning engine for automatic solution space exploration and transformation effect analysis. The approach uses only validity conditions and transformation definitions to automatically find input models suitable for validating transformation correctness. The feasibility and applicability are demonstrated in the context of constraint-driven modeling using the FORMULA [7] reasoning engine. For enabling reasoning about constraints as the result of model transformations, we used a formal (meta)modeling framework and formalized a basic consistency checking technique. In addition to a theoretical foundation, we validated the correctness of the implementation through systematic testing and by

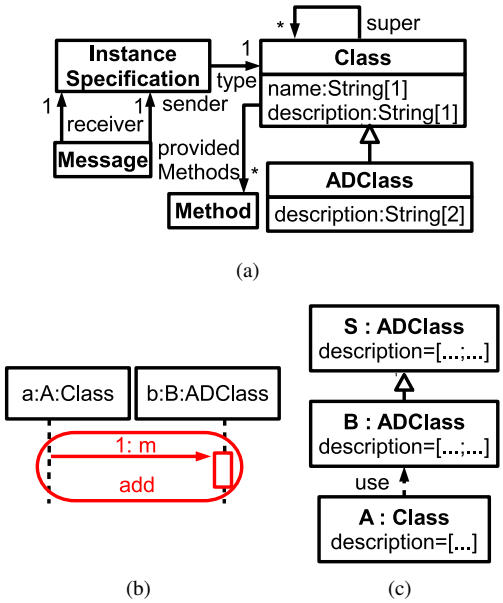

Fig. 1. Metamodel (a) and model (sequence (b) and class diagram (c)).

applying it to typical UML-like models.

II. Running Example

As a running example, we use an excerpt of a metamodel not unlike the UML metamodel as depicted in Fig. 1(a). The metamodel consists of the following elements: `InstanceSpecification`, `Message`, `Class`, `Method`, and `ADClass`. The element `Class` is used for modeling types. Such classes must provide a `description`, may have an arbitrary number of `Method` assigned, and may be part of a hierarchy if they have supertypes assigned (`super`). The element `ADClass` is a specialization of `Class` and provides an additional `description`, hence the prefix "AD" and an array of size 2 for the field `description`. The element `InstanceSpecification` is used for modeling specific instances of modeled classes and `Message` elements are used for modeling method calls between instance specifications.

We use this metamodel to build the models shown in Fig. 1(b) and Fig. 1(c). In Fig. 1(b) a sequence diagram is shown that defines instance specifications `a` and `b` of the types `A` and `B`, respectively. Fig. 1(c) shows a class diagram in which the types `A`, `B`, and `S` are defined. `A` is a `Class`, `B` and `S` are instance of `ADClass`. `S` is also defined as superclass of `B`. The message named `m`, sent from `a` to `b` and highlighted by the rounded frame in Fig. 1(b), is added to the model. According to UML semantics, this means that the class `B` must provide a method named `m`. At first sight, a simple model transformation that automatically adds a method with the appropriate name to the defining type of the message receiver would solve the problem. Note, however, that adding a method to any of `B`'s superclasses and interfaces (e.g., `S`) would also be a valid option. Clearly, deciding where to add the required method – if a new method is required at all – is non-trivial and requires knowledge of the modeled system. Thus automating such a decision through a model transformation can be dangerous as it might produce undesired models. In the next section we will discuss how the constraint-driven modeling approach addresses this issue and also illustrate why safe transformation is important when transformations are used.

III. Constraint-driven Modeling

As the illustration shows, using model transformation in situations where more than one valid solution may exist is not straightforward and deciding which solution is correct involves domain knowledge that often cannot be generalized. Using constraints avoids ambiguous transformations that may produce invalid results. However, managing them manually is non-practical in large models. The CDM approach tackles this problem and uses transformations that generate constraints on a model automatically instead of directly generating or changing model elements. When using a traditional transformation, the target model is generated or adapted automatically through the transformation execution on the source model. In contrast, the constraint-driven modeling approach executes the transformation T_c on the source model SM to generate the constraints C, as shown in Eq. (1). Compared to traditional transformations, the source model SM remains unchanged. The core idea is that defining invariants and desired conditions of valid models through constraints requires less knowledge and is therefore easier than defining a fixed set of rules that always produce a specific result that is always correct, (e.g., it is easier to define that two model elements must not have the same name than defining a rule that assigns meaningful names to all elements automatically). Moreover, constraints for similar elements (e.g., different instances of the same metamodel element) typically require similar constraints in which only specific parts differ, hence transformation rules can be seen as a structural template for a specific kind of constraint that can be instantiated and filled with data for specific model elements. Note that the target model TM_c (the subscript c indicates that it is constrained) is not manipulated directly in our approach, neither through a transformation nor through constraints.

$$\text{SM} \xrightarrow{T_c} C \rightsquigarrow \text{TM}_c \tag{1}$$

Let us now illustrate how the approach can be applied to our running example. As we have shown, there is more than one possibility for changing the class diagram in Fig. 1(c) to accommodate the change in the input models and thus defining a single transformation that always generates the right solution is not possible. However, we can use the knowledge provided in the model to define some requirements that must be fulfilled[1]:

- `B` must provide a method `m` (based on the message from `a` to `b`).
- `A` must provide a `description` (based on its type `Class`).

[1]We omit other invariants (e.g., that instances of `Class` and `ADClass` must provide a `name`) for space reasons.

```
1  rule RuleM
     from  s : Message
3    to    t : Constraint (
               element <- s.receiver.type
5              inv <- "self.providedMethods->
                   exists(m|m.name='" + s.name + "')")
7  rule RuleC
     from  s : Class
9    to    t : Constraint (
               element <- s
11             inv <- "self.description->size()=1")
   rule RuleADC
13   from  s : ADClass
     to    t : Constraint (
15             element <- s
               inv <- "self.description->size()=2")
```

Listing 1. ATL-like transformation rules to generate constraints.

- `S` and `B` must provide two `description` fields (based on their type `ADClass`).

These requirements can be easily stated as constraints, for example written in OCL. To generate them automatically, we use *ATL* [8] to define the three transformation rules shown in Listing 1. The first rule $RuleM$ (lines 1–6) is defined for messages in the model and creates constraints that require the receiver element's defining class to provide a corresponding method. Note that this does not require the class to directly provide the method; it is sufficient that the method is defined somewhere in the class hierarchy (e.g., in a superclass). Rule $RuleC$ (lines 7–11) is executed for `Class` instances and generates constraints that check the cardinality of the attribute `description`. Finally, the third rule $RuleADC$ (lines 12–16) checks that instances of `ADClass` provide two descriptions. Note that for simplicity we treat the two diagrams shown in Fig. 1(b) and Fig. 1(c) as one model (SM). When executing the transformation rules on the model in Fig. 1 we expect the following constraints to be generated:

C1 `element B: self.providedMethods->exists(m|m.name='m')`
C2 `element A: self.description->size()=1`
C3 `element B: self.description->size()=2`
C4 `element S: self.description->size()=2`

We expect one constraint to be generated for the message `m` and three constraints to be generated for checking the description attributes of `Class` and `ADClass` instances. The constraint $C1$ will show the inconsistency in our running example that `b` provides no method `m`.

Although the constraints in such a small example could be written manually, maintaining a large number of constraints during the modeling process where the metamodels and models change frequently is error prone. Consequently, constraints should be managed automatically. We have shown that CDM is practical in [4].

However, even with automated constraint management, designers manually write transformation rules or templates for desired constraints that can be executed or instantiated, respectively. Thus there is no guarantee that these transformation rules or templates produce constraints that are syntactically and semantically correct. Coming back to our running example, the transformation rules in Listing 1 actually not only produce the discussed constraints $C1$–$C4$, but – because instances of `ADClass` are also instances of `Class` – they will also create the following constraints $C5$–$C6$:

C5 `element B: self.description->size()=1`
C6 `element S: self.description->size()=1`

Obviously, the constraint combinations $C3 \wedge C5$ and $C4 \wedge C6$ are overconstraining the elements `B` and `S`, respectively. Both elements `B` and `S` are forced to have two different numbers of descriptions at the same time. Hence, the transformation leads to contradictory constraints.

The reason for this, somehow unexpected, result is that the transformation rules for cardinality checking constraints are defined for instances of the types `Class` and `ADClass` (lines 7 and 12 in Listing 1). Because `ADClass` is a subclass of `Class`, as defined in the metamodel depicted in Fig. 1(a), $RuleC$ is also executed for all instances of `ADClass`. Indeed, the transformation rules $RuleC$ and $RuleADC$ need additional guarding statements that ensure the execution of only one of the rules per element. In fact, automating constraint generation may even increase the risk of contradictions between constraints and overconstraining of models as designers specify transformation rules that are executed when certain requirements are fulfilled. This means that – in contrast to manual addition of constraints to individual model elements – the aggregation of constraints restricting a model element is no longer directly visible for a designer. Note that this situation is not unique to constraint transformations. Such issues may arise in other domains as well. Using transformations to generate target models implies that the generated model is not known before rule execution. Automating model generation typically involves large numbers of transformation rules that are written for complex metamodels. Visualization of dependencies between these rules is usually not available and thus the risk of overlooking possible side-effects increases with the number of rules. Checking the correctness of transformation rules by looking at the results for existing sample models is not a valid solution to this problem as source models may change and thus cause errors in target models. This can also occur later during model refactoring, updates, or extensions when target models have to be re-generated. Next, we present how safe transformation helps designers avoid errors in target models.

IV. Safe Model Transformation

As we have shown in the previous section, writing transformation rules – regardless of the specific domain – can be a highly complex task as there may be source models that cause the execution of combinations of rules that are not supposed to be executed together or also the execution of rules for elements where the rule should not be executed at all. **Safe model transformation guarantees that a transformation will always produce valid results.** Validity rules are defined by designers and typically stem from both the metamodel of the transformation's target model (syntax rules) and domain knowledge (semantic rules). Intuitively, safe transformation can be easily proven for a single source model by executing the transformation and validating whether the target model

is valid with a SAT-solver. However, our definition of safe transformation is that a transformation must not, under any circumstances, produce invalid results, meaning that there must not be any possible source model that causes invalid results. Based on the conventional notation for model transformation, we can formally define a transformation as the function shown in Eq. (2). A transformation is a function that uses a source model $sm \prec$ SMM and a set of transformation rules T as input and returns a target model $tm \prec$ TMM[2]. The signatures of the validity functions for both input and target model are shown in Eq. 3 and Eq. 4, respectively. We can then define a transformation to be safe if the condition in Eq. (5) holds for T. Proving that this condition holds requires the execution of the transformation rules specified in T for all possible instances $sm \prec$ SMM, which is indeed not a feasible option because common metamodels often allow an infinite number of different instantiations.

$$trans : (sm \prec \text{SMM}, T) \rightarrow tm \prec \text{TMM} \quad (2)$$

$$valid_S : sm \prec \text{SMM} \rightarrow \{true, false\} \quad (3)$$

$$valid_T : tm \prec \text{TMM} \rightarrow \{true, false\} \quad (4)$$

$$\begin{aligned} Safe(\text{SMM}, T) \Leftrightarrow \forall sm \prec \text{SMM} : valid_S(sm) \\ \Rightarrow valid_T(trans(sm, T)) \end{aligned} \quad (5)$$

$$\begin{aligned} Unsafe(\text{SMM}, T) \Leftrightarrow \exists sm \prec \text{SMM} : valid_S(sm) \wedge \\ \neg valid_T(trans(sm, T)) \end{aligned} \quad (6)$$

However, we take Eq. (5) as the condition that must hold for a transformation to be safe and use it to define the condition that holds for any unsafe transformation, as shown in Eq. (6). Proving that a transformation is unsafe thus requires only a single instance $sm \prec$ SMM to lead to a result $tm \prec$ TMM that is not valid.

The FORMULA reasoning engine supports automatic proofs through solution space exploration and transformations, thus it is an ideal choice for implementing safe transformation. However, any reasoning engine for automatic proofs can be used in principle, even though this could require the additional formalization of transformations. The solver is capable of expanding models on its own for finding proofs and partial models can be used for reasoning. For details about the reasoning mechanism please refer to [7] and [5].

Note also that safe transformation does not rely on static analysis and thus allows arbitrary input/output (meta)models, transformation languages, and conditions. The approach therefore is generic and not limited to specific domains or languages. Moreover, the complete integration of input model construction, the transformation, and the validity conditions means that both the validity conditions and the transformation itself are available to the solver for reasoning, potentionally enabling higher efficiency and precision.

[2]We use the notation $a \prec b$ to indicate that a is an instance of b.

V. Applying Safe Model Transformation to Constraint-driven Modeling

As discussed in Section III, constraints can be contradicting. Because we need to formally specify such a contradiction in order to apply safe transformation, let us first give formal definitions of constraints and the models they are applied to.

A. Formalizing Model Constraints and Consistency Checking

Constraints restrict model elements, therefore we need a formal definition of metamodels and models along with constraints definitions. Eq. (7) defines a metamodel MM_{cdm} that consists of a set of types $Types$ and the set of expressions available for individual constraints, called $CExpr$. An instance of the metamodel is then a model $M \prec MM_{cdm}$, as defined in Eq. (8), consisting of model elements N, connections E that link model elements, and constraints C.

$$\text{MM}_{cdm} := \langle Types, CExpr \rangle \quad (7)$$

$$\begin{aligned} M \prec MM_{cdm} := \langle N, E, C \rangle | \\ N = \{x | x \prec y \in Types\} \wedge \\ E = \{\langle a, b, c \rangle | a \in N \wedge b \in N \wedge c \subseteq N\} \wedge \\ C = \{\langle context, exp \rangle | context \in N \wedge exp \prec CExpr)\} \end{aligned} \quad (8)$$

$$value : (N, N) \rightarrow \mathcal{P}(N), (x, y) \mapsto z : \exists \langle x, y, z \rangle \in E \quad (9)$$

$$validate(c \in C) \mapsto \begin{cases} inconsistent & \text{if } c \text{ is violated} \\ consistent & \text{otherwise} \end{cases} \quad (10)$$

$$\begin{aligned} allowed(\langle a, b, c \rangle \prec MM_{cdm}) \mapsto \{\langle n, e, v, r \rangle | \\ n \in a \wedge e \in a \wedge v \subseteq a \wedge r \subseteq c \wedge \\ (\exists \langle h, i, j \rangle \in E : n = h \wedge e = i) \wedge \\ (y = value(n, e) \wedge y \notin v \Rightarrow \exists r_x \in r : \\ validate(r_x) = inconsistent)\} \end{aligned} \quad (11)$$

$$\begin{aligned} contradiction(m \prec MM_{cdm}) \Leftrightarrow \\ \exists \langle n, e, v, x \rangle \in allowed(m) : v = \emptyset \end{aligned} \quad (12)$$

Note that a connection between model elements in Eq. 8 consists of three parts: i) the source model element (a), ii) a connection identifier (b) which is also a model element, and iii) the source of the connection (c). Such a connection can be interpreted as a value c being assigned to the property b of element a. Thus, we can define a helper function $value$ to retrieve the value assigned to a given model element property, as shown in Eq. 9.

We define the generic constraint validation function as shown in Eq. (10). We omit a detailed discussion of constraint validation because of space restrictions and the fact that validation functions heavily depend on the used consistency checking technique. Additionally, we define the function $allowed$ that returns the set of allowed values for model element properties, as shown in Eq. (11). This set includes tuples of a model element (n), a property of n (e), a set of values (v) that may be assigned to the model element's property without causing a constraint to become inconsistent,

and a set of constraints (r) that restricted the allowed values. The function $contradiction$ takes a model as input and checks it for contradictions, as shown in Eq. (12). A contradiction occurs if there is at least one element for which no value can be assigned without causing an inconsistency. We omit more detailed definitions as they would strongly depend on the used modeling and constraint languages.

Now that the metamodel and the corresponding functions for constraint validation and the detection of contradictions are defined, we can use this information to check the transformations from Section III.

B. Ensuring Safe Constraint Transformation

We converted the ATL-like transformations shown in Listing 1 to transformations T_{cdm} that can be executed by the FORMULA solver. A constraint transformation CT can be written as shown in Eq. (13). When executing the transformation for a source model $sm \prec \text{MM}_{cdm}$, the result is the constrained model $tm_c \prec \text{MM}_{cdm}$ which includes all model elements defined in sm and a set of constraints generated for these element. The last part needed for safe transformation are the source and target validity functions $valid_{Scdm}$ and $valid_{Tcdm}$, respectively. For the former, the conditions are domain-specific and thus independent of CDM. Therefore, we omit a formal definition in Eq. 14. For the latter, results to be valid if they are free of contradicting constraints, as shown in Eq. (15). By substituting the constraint transformation function (13), the metamodel definition (7), and the validity conditions (14) and (15) in Eq. (6) we get the unsafe condition as shown in Eq. (16).

$$CT : sm \prec \text{MM}_{cdm} \xrightarrow{T_{cdm}} tm \prec \text{MM}_{cdm} \tag{13}$$

$$valid_{Scdm}(x \prec \text{MM}_{cdm}) \Leftrightarrow x \text{ is syntactically correct and valid w.r.t. domain semantics} \tag{14}$$

$$valid_{Tcdm}(x \prec \text{MM}_{cdm}) \Leftrightarrow \neg contradiction(x) \tag{15}$$

$$Unsafe(\text{MM}_{cdm}) \Leftrightarrow \exists sm \prec \text{MM}_{cmd} : valid_{Scdm}(sm) \wedge \neg valid_{Tcdm}(CT(sm)) \tag{16}$$

Note that we use the transformation function CT directly on the right-hand side of the equation and thus we eliminate the corresponding argument on the left-hand side. We now ask the FORMULA solver to prove this condition and search for examples of source models sm that lead to invalid target models: `solve CT sm Unsafe`. The solver will return an input that leads to a contradiction, which tells us that the transformation is unsafe. In particular, this input will include an instance of `ADClass` and also include facts that help designers find the constraints and the transformations that caused the contradiction).

Assume that, after taking a closer look at the transformation rules $RuleC$ and $RuleADC$ based on the solver output, we identified the error and added guarding statements (e.g., `s.getClass().getName()="Class"`) so that $RuleC$ is no longer executed for instances of `ADClass`. Asking the solver again for an example sm of a contradiction-causing model will fail and the solver will tell us that it could not find such an example, meaning that the transformation is safe.

VI. Validation

In addition to the presented theories, we demonstrate the applicability of the generic safe transformation approach by implementing it for constraint-driven modeling. We also discuss threats to validity in this section.

A. Correctness

In the paper we presented the theoretical foundations of safe model transformation. The presented conditions are correct and they capture the ideas and principles of safe transformation. The correctness was further assessed by implementing those conditions and functions in a program that produced correct results as we discuss next.

B. Implementation

To ensure the feasibility of safe transformation and the validity of the theoretical foundations presented in the paper, we have implemented a safe transformation framework for CDM using the FORMULA language. This implementation is a straight forward translation of the presented formal definitions to type declarations and functions, thus we omit a detailed discussion. It allows the definition of metamodels and models (based on an existing metamodeling framework [5]), and it supports the definition of constraints. Moreover, it provides functions for constraint evaluation and for reasoning about constraints. For this feasibility study we used a subset of the constraint expressions available in OCL. The correct behavior of the implementation was ensured through testing with systematically varied inputs for expressions.

As mentioned in Section V, we used the implementation with the model discussed in Section II and the transformation rules from Listing 1. For the running example and the incorrect transformation rules, the FORMULA solver returned different solutions (i.e., sample models) that all included an instance of `ADClass` and contained contradicting constraints. Moreover, the resulting facts derived by FORMULA included as many constraint contradictions as there were `ADClass` instances. After fixing the transformation rules, the problem was not satisfiable anymore – as expected. This indicates that the translation to FORMULA is correct and the approach produces the desired result in practice.

C. Applicability

The presented framework is suitable for ensuring safe transformation for CDM in arbitrary modeling domains. To enable safe transformation in general (i.e., for transformations different to those used in CDM), the following three steps are necessary: i) define source and target metamodels using an existing metamodeling framework, ii) describe transformation rules in FORMULA, and iii) describe desired validity conditions. We believe that the metamodel definition can be automated for metamodels based on common frameworks (e.g., MOF). The transformation description in FORMULA

is straightforward and uses concepts similar to common transformation languages such as ATL – this step may also be automated. Overall, we deem the effort needed for using safe transformation acceptable given the importance of transformation correctness and the effort required for locating and fixing errors caused by unsafe transformations in later development stages. Overall, we believe that the steps required for using the framework are manageable.

D. Threats to Validity

We have used the FORMULA solver for finding input models for which a transformation is unsafe with respect to defined criteria. Indeed, FORMULA is capable of searching solutions efficiently. Nevertheless, it can only guarantee that there is no solution for a defined maximum size of solution models [5]. In [9], Sen et al. use Alloy for constructing test models that meet defined well-formedness criteria from partial models. We use this principle more extensively as our input data for constructing input models for transformations is often empty. Although we expect target model errors to typically involve only a relatively small number of elements, the bounded search space is a practical threat to validity. Even if current solvers might not find input models that unveil errors in transformations, manually written test models are far from finding all errors [9], thus our approach gives designers additional confidence in the correctness of their work.

VII. Related Work

Model transformation has become a very active field of research. We now discuss other approaches closest to our topic. Sen et al. [9], [10] presented techniques for finding input models for testing model transformations. They use Alloy to expand partial models automatically to complete models that conform to well-formedness rules. The fact that they use different languages and strategies for finding sample models shows that our generic approach can be implemented with a range of languages. Note, however, that safe transformation integrates the construction of test input models and the execution of transformations. Guerra et al. [11] propose the use of contract for validating transformation correctness. This is basically the same concept safe transformation uses. However, they use QVT to enforce defined pre- and post-conditions of transformations and rely on existing input models while safe transformation uses such contracts to actively search for input models that lead to a violation of the target model constraints. Cabot et al. [12] proposed an approach for deriving OCL constraints from declarative transformation specifications. Their approach generates invariants that must hold between source and target models. Our approach can be used to ensure that, for example, only non-contradicting invariants are generated. Jackson et al. [5] previously used FORMULA for describing a general purpose (meta)modeling framework.The reasoning they did about transformations based on their framework can be seen as an application of the safe transformation approach to identify undesired behavior of transformations.

VIII. Conclusions and Future Work

In this paper we presented the safe transformation approach that allows the use of formal modeling languages for proving assumptions about arbitrary model transformations for either arbitrary source models or also source models with specific characteristics. In particular, we demonstrated its applicability for the constraint-driven modeling approach by ensuring constraint satisfiability and checking the semantical correctness of transformation rules and their results. The safe transformation approach was validated by implementing it using the FORMULA language. For future work we plan to integrate the framework in existing tools that use CDM and also implement the approach for other domains.

References

[1] Anda, B., Hansen, K., Gullesen, I., Thorsen, H.K.: Experiences from introducing uml-based development in a large safety-critical project. Empirical Software Engineering **11**(4) (2006) 555–581

[2] Egyed, A.: Instant consistency checking for the UML. In: ICSE. (2006) 381–390

[3] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)

[4] Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Constraint-driven modeling through transformation. Software and Systems Modeling (2013) DOI: 10.1007/s10270-013-0363-3.

[5] Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: MoDELS. (2011) 653–667

[6] Amrani, M., Lucio, L., Selim, G.M.K., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A tridimensional approach for studying the formal verification of model transformations. In: ICST. (2012) 921–928

[7] Jackson, E.K., Sztipanovits, J.: Towards a formal foundation for domain specific modeling languages. In: EMSOFT. (2006) 53–62

[8] OBEO, INRIA: ATLAS transformation language (ATL). http://www.eclipse.org/atl/ (2014)

[9] Sen, S., Mottu, J.M., Tisi, M., Cabot, J.: Using models of partial knowledge to test model transformations. In: ICMT. (2012) 24–39

[10] Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: ICMT. (2009) 148–164

[11] Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. **20**(1) (2013) 5–46

[12] Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software **83**(2) (2010) 283–302