VNU Journal of Science: Comp. Science & Com. Eng., Vol. 39, No. 1 (2023) 1–22

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

A Contract-Based Specification Method for Model Transformations

Thi-Hanh Nguyen, Duc-Hanh Dang*

VNU University of Engineering and Technology,
144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Received 31 December 2022
Revised 01 March 2023; Accepted 31 March 2023

Abstract: Model transformations play an essential role in model-driven engineering. However, model transformations are often complex to develop, maintain, and ensure quality. Platform-independent specification languages for transformations are required to fully and accurately express requirements of transformation systems and to offer support for realization and verification tasks. Several specification languages have been proposed, but it still lacks a strong one based on a solid formal foundation for both high expressiveness and usability. This paper introduces a language called TC4MT to precisely specify requirements of transformations. The language is designed based on a combination of a contract-based approach and the graph theory foundation of triple graph grammar. Specifically, we consider graph patterns as core elements of our language and provide a concrete syntax in the form of UML class diagrams together with OCL conditions to visually and intuitively represent such pattern-based specifications. We develop a support tool and evaluate our proposed method by comparing it with current methods in literature.

Keywords: Model Transformation, Contract-Based Specification, Domain Specific Languagues, UML/OCL, Triple Graph Grammars.

1. Introduction

In the context of model-driven engineering (MDE), models are considered as primary artifacts, and model transformations are crucial elements to automatically manipulate them with such tasks as querying, synthesizing, and transforming models. Many languages have been proposed to implement model transformations, but it still lacks effective methods, languages, and tools to support specification and verification of transformations. In practice, an effective transformation specification language often needs to be independent of implementation platforms and should

*Corresponding author.
E-mail address: hanhdd@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.657

be defined at a higher level of abstraction than implementation languages. Moreover, a powerful transformation specification language needs to be based on a solid formal foundation, capable of fully expressing various features of model transformations as what with complex software applications. Such a specification language is also required to be easy to use for different stakeholders.

Several transformation specification languages have been introduced in [1–6]. For the requirement of expressiveness, current languages in literature for transformations are often defined as domain-specific languages (DSLs) to capture specific features of transformations. The authors in [1, 2, 5, 7] proposed a technique to specify transformations as behavioral contracts, i.e., a transformation specification would include preconditions, postconditions, and invariants as restrictions on input models, output models, and the relationship between them, respectively. An advantage of the specification method is that it facilitates the verification of correctness and robustness of model transformations. Several other works [3, 4, 8] employ graph rewriting rules to specify transformations and such a graph-based formal environment could allow them to offer support for the implementation and verification of time-dependent properties of transformations. The works in [8–10] focus on Triple Graph Grammars (TGG) [11] as an effective approach for specifying and implementing bidirectional model transformations.

For the requirement of usability, transformation languages need to have an intuitive syntax with a symbology familiar to modelers. Several works proposed to represent constraints of transformation in a textual syntax such as OCL (Object Constraint Language) invariants [5, 7], Alloy [12] and TL [2]. These methods could accurately express complex requirements but lack visualization. Several other domain-specific languages are proposed based on visual UML (Unified Modeling Language) models to specify the model transformations such as DSLTrans [3], DelTa [4], Pamomo [1], and MTP [6]. Up to now, to design a platform-independent transformation specification language, using easy-to-use visual notations to precisely express complex requirements of model transformations in MDE is still a challenging issue.

In this paper, a high-level visual declarative language called as TC4MT[1] is proposed to specify transformation requirements. The language TC4MT allows specifying different aspects of model transformation, including preconditions, postconditions, invariants, and transformation rules as protocol contracts. The language is designed based on graph patterns as core elements. Pattern-based specifications in the language are represented using the notation of the UML class diagram together with OCL conditions in a similar way to the MOF language [13], the standard to define modeling languages in MDE [14]. The TC4MT has a solid formal theory foundation and could support bidirectional model transformations since transformation rules in a TC4MT language are defined as TGG rules.

The rest of this paper is organized as follows. Section 2 surveys related works. Section 3 presents a running example together with a research question how to precisely specify requirements of a transformation using contracts. Section 4 introduces a contract-based specification language. Section 5 then explains the support of the proposed specification method in quality assurance of model transformations. The support tool and the effectiveness of the proposed specification method are presented in Section 6 and Section 7. This paper is closed with a conclusion and future works in Section 8.

2. Related work

We review current approaches in literature related to implementation-independent transformation specification on the main issues: i) syntax and semantics of a specification language;

[1]TC4MT means Test Cases for Model Transformations

ii) specification of transformation features; and iii) purposes of the specification method.

First, source and target models of a model transformation need to conform to metamodels or other syntax definitions. There may be several input (source) models used by a transformation, and possibly several output (target) models. A model transformation is termed update-in-place if the resulting model includes both input and output model. The high-level model transformation specification languages provide key concepts to express features of model transformations. Model transformations can be specified by many different means, e.g., formal languages [2, 12], visual specification languages with formal semantics [1, 3, 4], or examples expressing real transformation situations [15–17]. Formal languages [2, 12] use a textual concrete syntax based on the constraint and computation expressions to express complex requirements. Transformation requirements can be expressed using a specific textual syntax as mentioned in [2, 5, 7, 12] or a visual graph as mentioned in [1, 3, 4, 8]. A transformation specification should be readable for different stakeholders including implementation developers, testers, and modeling experts; Therefore, several works tend to employ general-purpose languages like UML and OCL. The authors in [5, 7] propose using OCL constraints expressions to express properties of model transformations. The works in [6] employ UML models in order to express transformation requirements at different abstraction levels, concerning the phases of a transformation development life cycle. However, these approaches as regarded in [5–7] have not yet shown formal semantics of transformation specifications.

Second, in terms of functional coverage, transformation specification languages should be able to express different aspects of transformation requirements. Current specification languages and transformation implementations often allow specifying contracts about the data types of input and output models using meta-modeling techniques. Metamodels are often represented in the form of UML class diagrams enriched by OCL constraints as the well-formed rules [13]. The contract-based specification approaches [1, 3, 5, 12] often focus on expressing behavioral contracts by preconditions, postconditions, and invariants. At the specification level, the works in [3, 4, 8] use graph transformation rules to define model transformation rules as functional units of the transformation. A transformation specification with contracts should also allow us to capture valid sequences of state transitions. Such contracts are referred to as *protocol contracts*. Besides, in rule-based bidirectional model transformations, trace elements are used to define mappings between source and target elements. Triple graph grammars (TGG) [11] uses a corresponding graph to connect source graph and target graphs. An interesting feature of TGGs is that it could support bidirectional transformations. The authors in [1, 4, 8] introduce languages based on TGGs to specify transformations.

Finally, current transformation specification methods in literature base on the contract-based approach to support quality assurance purposes. This approach could provide a formal specification of requirements for transformation systems, that is used as a basis for the next phases of the development process. The authors in [18] propose a domain-specific language based on the language ETL [19] to embed design patterns into transformation specifications in Pamomo [1]. The authors in [2] propose a similar domain-specific language to represent transformation rules. In general, current transformation specification languages often focus on static structural contracts instead of on a full specification covering both static and dynamic aspects.

3. Overview of Contract-based Specification of Model Transformations

This section aims to provide a background of contract-based specification of model transfor-

mations. To illustrate the different transformation contracts, a transformation example named CD2RDBM between UML class diagram (CD) models and relational database (RDBM) models is considered.

3.1. Motivating Example

Figure 1 shows the simplified metamodels of UML class models and relational database models. A model transformation specification expresses the requirements by referring to the structural elements of the source and target metamodels. A model transformation often do not work on entire all instances of the input/output metamodel. Therefore, some constraints are added to exclude model instances of a metamodel that will not be handled by the transformation. For example, this work is interested in transformation situations of the CD2RDBM transformation with the following constraints on input models.

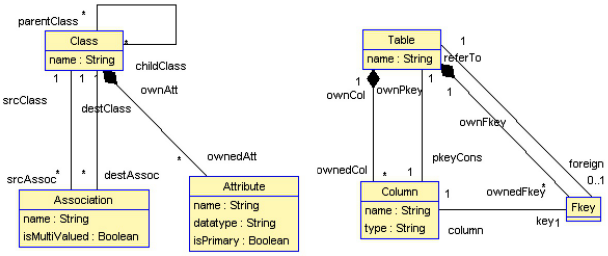

Figure 1. The source and target metamodels of the CD2RDBM transformation.

pre1: A class does not inherit itself;

pre2: The name of a class is unique;

pre3: The name of attributes in a class is unique;

pre4: The subclass does not redefine attributes of its super class;

pre5: The name of an association does not coincide with the name of a class.

A certain transformation might need to guarantee that produced output models fulfill certain conditions beyond metamodel constraints. The followings are constraints on the generated output models of the CD2RDBM transformation:

post1: A table name is unique;

post2: Two columns of a table must have distinct names;

post3: A table cannot have more than one primary key column.

Besides constraints on input and output data, a model transformation must ensure valid mapping relationships between input and output models. The followings are constraints on mapping relationships for the CD2RDBM transformation.

r1 - *Class2Table*: it maps classes in a CD model to corresponding tables;

r2 - *Attr2Column*: it maps non-primary attributes to columns without the primary key role;

r3 - *PrimaryAttr2Column*: it maps primary attributes to columns that play the primary key column;

r4 - *MultiToMultiAss2Table*: it maps multi-valued aggregation and association to a new associative table that relates the two original tables;

r5 - *SingleToMultiAss2Fkey*: it maps aggregation and association relationships characterized by a single-valued end and a multi-valued end (0..*, 1..*) to a foreign key that relates two original tables;

r6 - *SubClass2Table*: it maps a subclass to the table w.r.t. the super class; and an optional rule for model refactoring after forward transformation executions;

r7 - *InheritanceFlattening*: it flattens the inheritance hierarchy by copying the super class's attributes down to all of the subclasses and removing the super class from the model.

While the above requirements describe valid mapping relationships, the following invalid mapping relationships state forbidden situations of the CD2RDBM transformation.

inv1: If the class model has two classes in an inheritance relation, then the RDBM model should not only contain two distinction tables mapping to these classes;

inv2: If the class model has two classes in an inheritance relation, then the RDBM model should not only contain a corresponding table mapping to the subclass without a corresponding table mapping to the super class;

inv3: If the class model has a class, then the RDBM should not have two distinction tables that map to this class.

In MDE, a model transformation should be developed as a typical software: i) At the specification phase, transformation requirements can be expressed at different abstraction levels by different means such as natural language, modeling language, and formal language; ii) The transformation is then realized by an implementation language; and iii) The implementation of the transformation needs to be checked whether it satisfies the transformation requirements, so that the quality of the transformation could be ensured. Testing tends to be a promising technique for validating and verifying model transformations. To effectively apply this technique, suitable transformation specification techniques are required to support the analysis and design of test conditions.

3.2. Model Transformations Contracts

Design by contracts [20] was introduced as a means to increase quality in terms of the correctness and robustness of the constructed software. It allows formalizing requirements as contracts which may be used to test the software. Another advantage of contracts is that they tend to describe what a piece of the software is being done instead of how it is done. In MDE, design by contracts also is an approach for the quality assurance of model transformations as regarded in [1, 5, 7]. Transformation contracts could offer means to analyze at the specification level properties of the transformation and to verify them using black-box testing techniques. Similar to contracts introduced in [21], there are three typical types of transformation contracts regarding the functional aspect of software: type contracts, behavioral contracts, and protocol contracts.

Type contracts are defined as restrictions on the types of manipulated data. With model transformations, they ensure that source/target models conform to their metamodels.

Behavioral contracts include preconditions, postconditions, and invariants. The precondition part, e.g., the requirements *pre1-pre5* in the example transformation, puts restrictions on the required input models such that the transformation is applicable. The postcondition part, e.g., the requirements *post1-post3*, is used to express whether or not an output model should contain certain configurations of elements. The invariant part includes conditions that need to be satisfied by any pair of source-target models of a correct transformation. Positive (negative) invariants are used to specify valid (invalid) relations between source and target models, e.g., the requirements *r1-r7* correspond to positive invariants, and *inv1-inv3* correspond to negative ones.

Protocol contracts specify the global behavior of source/target models in terms of synchronizations between method calls w.r.t. state transitions of the transformation system. The aim of such a contract is to describe the dependencies between services provided by a component, such as sequence, parallelism, or shuffle. They are considered as protocol contracts and often represented by *transformation rules*.

4. TC4MT: A Transformation Specification Language

This section introduces the TC4MT language to specify model transformations. The main objective of the TC4MT language is to represent requirements of a transformation explicitly as a contract-based specification: It is a declarative, formal, visual language designed to express trans-

formation contracts. The TC4MT could support three contract levels, as explained in Section 3.2: type contracts, behavioral contracts, and protocol contracts. Designing such a language to represent contracts would bring out two advantages: i) the definition of contracts is implementation-independent, i.e., not tied to a particular target transformation language, which is especially favorable in MDE since no dedicated standard transformation language has been brought forward so far [14, 22]; and ii) the designer could specify explicitly desired properties of a transformation before implementation, which may be used for guiding the implementation.

The TC4MT language represents transformation requirements based on TGG rules and graph patterns. Such a TGG-based semantics would provide a formal basis to verify quality properties of a model transformation. For example, the validation of source and target models is performed by checking the fulfillment of contracts. Besides, transformation invariants expressing the relationship between source and target models are the basis for checking the semantic preservation property between source and target models. Analysis of the dependency relationship between operation rules in an execution scenario allows defining test scenarios to verify the behavior of the switch such as termination and convergence.

4.1. Abstract Syntax

Figure 2 shows the metamodel for the abstract syntax of the TC4MT language. It consists of four basic parts: (A) Meta-concepts to express contracts (preconditions, postconditions, invariants, and transformation rules); (B) Meta-concepts to specify transformation rules; (C) Meta-concepts to capture model structure; and (D) Meta-concepts to express test suites.

4.1.1. Meta-concepts to Express Contracts

The followings are meta-concepts for contract specification, as shown in Figure 2(A).

Metamodel. This meta-concept is used to express both source and target languages. These languages together with TC4MT are designed based on the MOF [13] standard for the metamodeling approach.

TrafoSpecification. The meta-concept is used to represent a transformation as a composition of `Preconditions` and `Posconditions` of the transformation and `Invariants`. Each condition of the transformation is expressed on patterns w.r.t. structure elements and their relationship, i.e., part of the metamodel. The meta-concept `Pattern` is further explained in Section 4.1.2.

Precondition. A precondition is expressed by a pattern as a restriction on input (source) models. A transformation specification might consist of several preconditions that can be positive patterns to express required conditions or negative patterns to express forbidden conditions on input models. A valid model transformation refuses to take invalid source models as the input.

Postcondition. A postcondition is expressed by a pattern as a restriction on output (target) domain models. A transformation specification might consist of a set of postconditions. Postconditions can be positive patterns to express required conditions or negative patterns to express forbidden conditions on generated output models. Faults will arise if any generated output model does not satisfy postconditions.

Invariants. The meta-concept expresses conditions that a transformation needs to fulfill at any moment in time. Within our approach, `Invariants` can be expressed by triple patterns. The meta-concept `TriplePattern` is further explained as below.

4.1.2. Meta-concepts to Specify Rules

The meta-concepts for rule specifications are shown as in Figure 2(B).

Pattern. As mentioned above, the meta-concept `Pattern` represents a set of snapshots, each of which includes instances of meta-classes and links (`Reference`) between them.

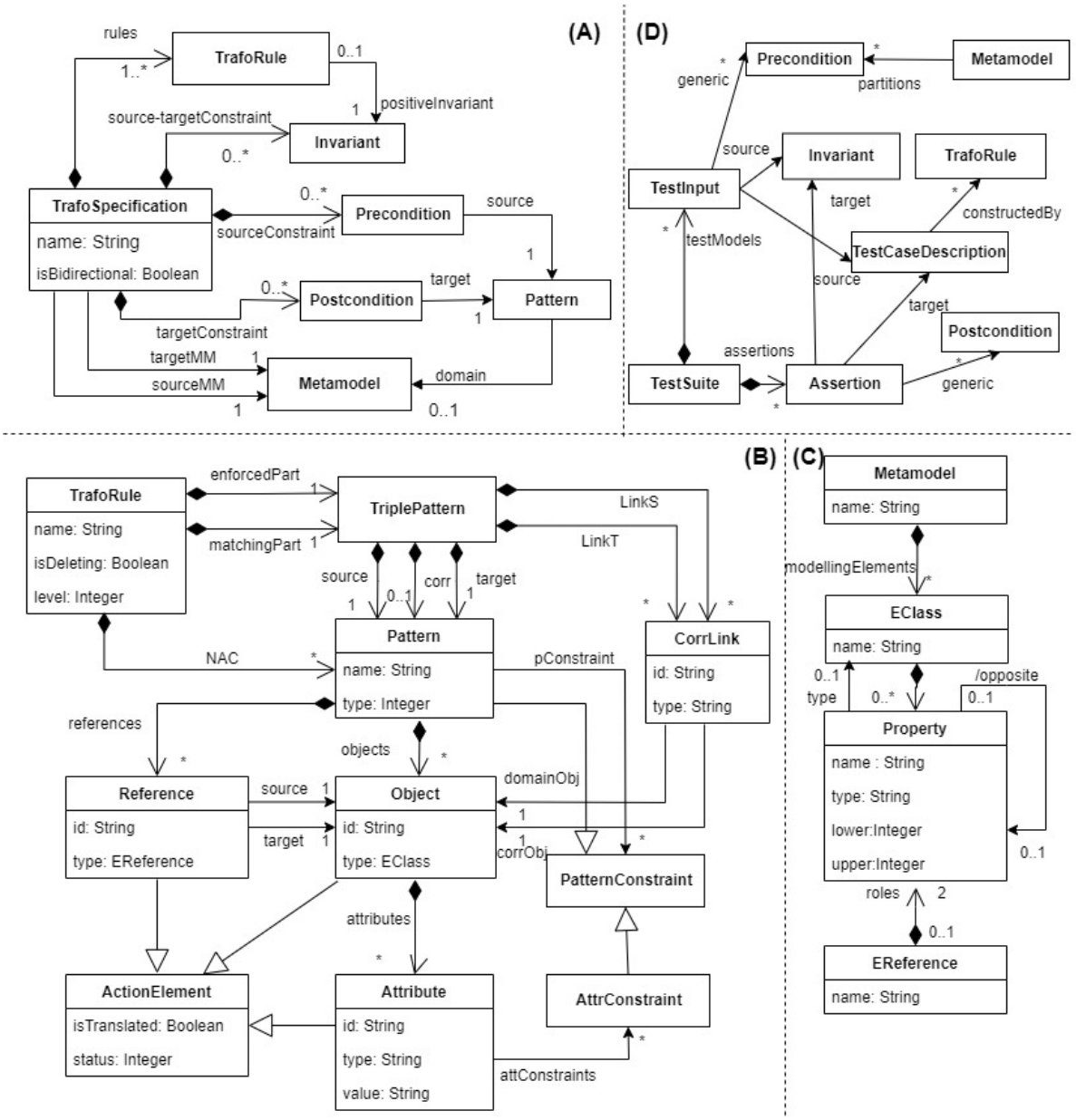

Figure 2. Metamodel of the language TC4MT.

A pattern might be attached with constrains (`PatternConstraint`) as restrictions on the snapshots of the pattern. A `PatternConstraint` can be a `Pattern` and often expressed in the form of OCL conditions. Each `Object` includes a set of `Attributes`, each of which might be restricted by `AttrConstraints`, including value assignment or type binding.

TriplePattern. A `TriplePattern` consists of three patterns (the source, the target, and the optional corresponding pattern) that can are connected together by links.

TrafoRule. The meta-concept is to represent transformation rules as functional units of a model transformation. A `TrafoRule` can be seen as an extension of an invariant by adding actions to manipulate model elements when executing transformation rules. Actions are often included in the specification of the time-dependent behavioral semantics of the transformation.

Matching & Enforcing. A transformation rule consists of negative application conditions (`NAC`) and two triple patterns (`matchingPart` and `enforcedPart`). The patterns is used to match and enforce the rule, i.e., applying the rule for a graph rewriting step, as explained in Section 4.2.

4.1.3. Meta-concepts to Capture Model Structure

Figure 2(C) shows MOF meta-concepts `EClass`, `Property`, and `EReference` to represent model elements: Each `EClass` consists of `Properties` that describe the instance's data values. Each `EReference` consists of two `EClasses` at the two ends associated with different roles. Each role corresponds to a property on the opposite side represented by the opposite association between the two `EClasses`.

4.1.4. Meta-concepts to Express Test Suites

The meta-concepts to represent test suites are shown in Figure 2(D). A `TestSuite` includes test cases, each of which contains test-input models (`TestInput`) and output assertions (`Assertion`). The test inputs can be defined based on preconditions, the source pattern of invariants, and the source part of transformation rules within test case descriptions (`TestCaseDescription`). Output assertions can be directly derived from the postconditions, the target pattern of invariants, and the target part of transformation rules. Preconditions as discussed in [12, 23, 24] allow us to define partitioning conditions based on the type of class's attributes and association multiplicities.

4.2. Operational Semantics

This section aims to provide an operational semantics of TC4MT. Basically, transformation rules in a TC4MT specification that are referred to as so-called positive invariants expressing valid relationships between source and target models. The execution semantics of the TC4MT specification is mapped to a transformation system of a corresponding triple graph grammar.

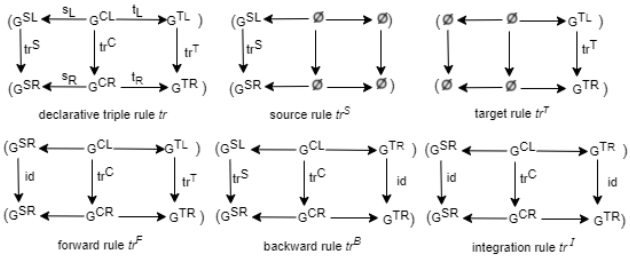

Figure 3. The operational rules are derived from a declarative TGG rule.

Definition 1. (Flattening Triple Graph) *A triple graph $G = (G^S \xleftarrow{s_G} G^C \xrightarrow{t_G} G^T)$, the flattening construction FG of graph G is a plain graph defined by the disjoint union $FG = G^S + G^C + G^T + LinkS(G) + LinkT(G)$. The FG consists of the following components.*

- *Three subgraphs G^S, G^T, and G^C are the source, target, and correspondence graphs, respectively. They are plain graphs with edges and nodes (the edge and node sets of a graph G are denoted G_E and G_V, respectively). The subgraph G^C only contains*

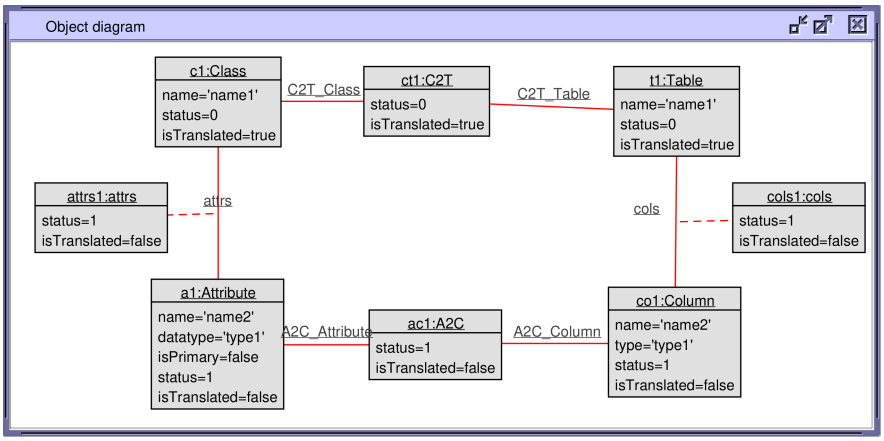

Figure 4. The declarative structure of the triple rule *Attr2Column*.

nodes but not edges. A graph G with edge set GE and node set GV

- *The additional edges in* $LinkS(G) = \{(x,y)|x \in G_V^C, y \in G_V^S, s_G(x) = y\}$ *define the mappings* s_G *from the correspondence graph to the source graph.*
- *The additional edges in* $LinkT(G) = \{(x,y)|x \in G_V^C, y \in G_V^T, t_G(x) = y\}$ *define the mappings* t_G *from the correspondence graph to the target graph.* $\forall x \in G_V^C \mid \exists(y_1,y_2),((x,y_1) \in LinkS(G)),((x,y_2) \in LinkT(T)) \Leftrightarrow (y_1 \in G_V^S) \wedge (y_2 \in G_V^T) \wedge (s_G(x) = y_1) \wedge (t_G(x) = y_2))$.

Definition 2. (Transformation Rule) *A transformation rule* $tr = (L \longrightarrow R, AC) = ((G^{SL} \leftarrow G^{CL} \rightarrow G^{TL}) \longrightarrow (G^{SR} \leftarrow G^{CR} \rightarrow G^{TR}), AC)$ *consists of two flattened triple graph* $L = (G^{SL}, G^{CL}, G^{TL}, LinkS(L), LinkT(L))$, $R = (G^{SR}, G^{CR}, G^{TR}, LinkS(R), LinkT(R))$ *of the same flattened triple graph. The triple graphs for the LHS and the RHS of the rule are marked and distinguished by two attributes isTranslated and status of each node.* $AC = (AC^{SL}, AC^{CL}, AC^{TL})$ *is the application condition of the rule, where* AC^{SL}, AC^{CL}, *and* AC^{TL} *are constraints on the* G^{SL}, G^{CL}, *and* G^{TL} *of the rule, respectively.*

Definition 3. (Transformation Specification) *A transformation specification is a tuple* $TS = (P_{pre}, P_{post}, P_{inv}, TR)$, *where* P_{pre} (P_{post}) *are the preconditions (postconditions) of the transformation,* P_{inv} *are invariants, and TR are rules.*

Note that a transformation rule in this definition corresponds to a triple rule (a.k.a. a TGG rule) of a triple graph grammar. The pre- and postconditions of a transformation specification are conditions on a particular domain w.r.t. a metamodel represented as a type graph [11, 25]. Invariants and transformation rules are represented by typed triple graphs. A type triple graph can be seen as a combination of two plain graphs for source and target parts and additional edges to connect them with each other. This composite graph is referred to as a flattening triple graph.

As depicted in Figure 3, a declarative TGG rule express a co-evolution relationship between the source and target models. Besides, there exist operational TGG rules derived from a declarative TGG rule for different transformation scenarios, e.g., forward, backward, and integration transformation scenarios. An integration transformation aims to define trace elements of a corresponding graph to relate the source and target models.

Example. Figure 4 describes the declarative TGG rule *Attr2Column* for the example transformation. The operational rules derived from the *Attr2Column* rule for the forward, backward, and integration scenario are shown as in Figures 5, 6, and 7, respectively.

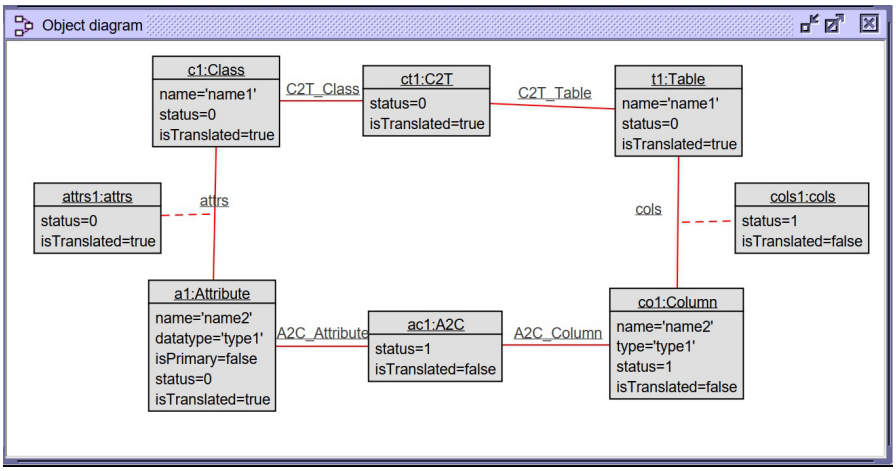

Figure 5. The forward TGG rule *Attr2Column*.

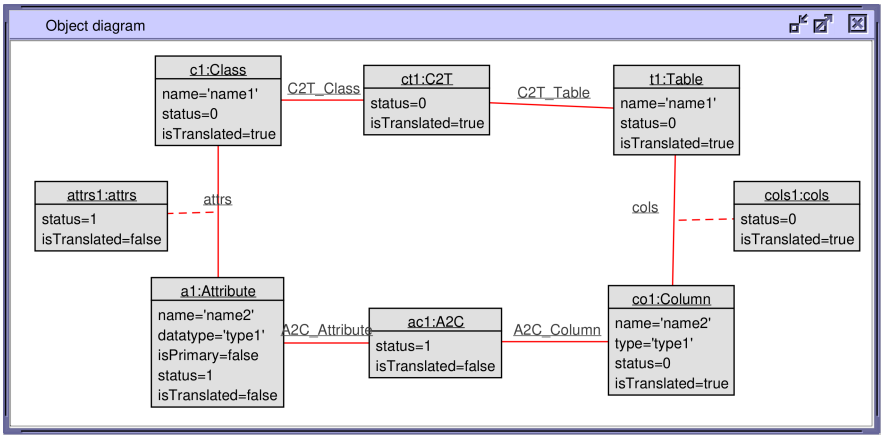

Figure 6. The backward TGG rule *Attr2Column*.

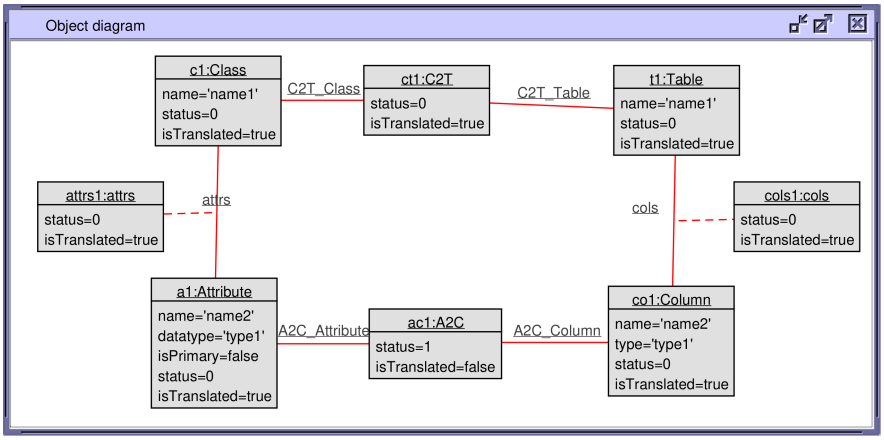

Figure 7. The integration TGG rule*Attr2Column*.

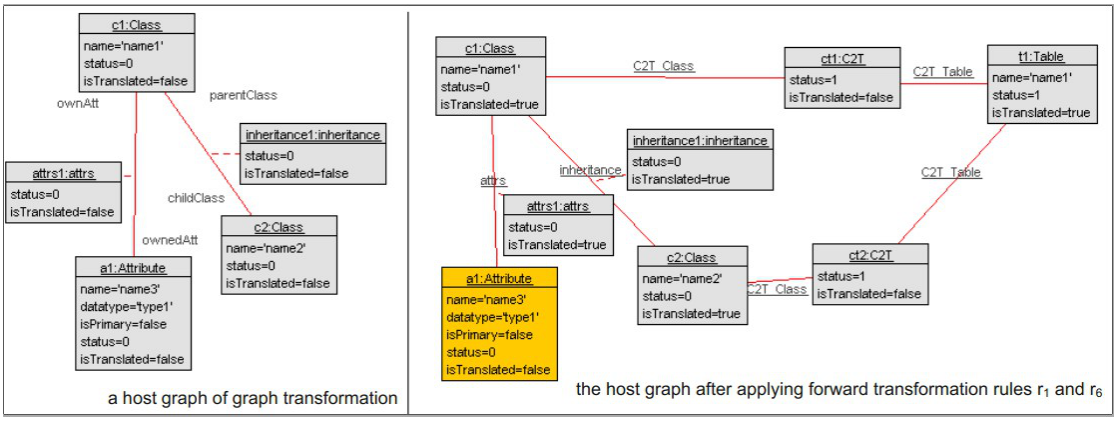

Figure 8. The application of the operational rule *Attr2Column*.

We explain how a model transformation scenario is realized by applying operational TGG rules as follows. Given an initial target graph so-called the host graph G and a set of derived rules. At initial step, all the elements of G have the state $isTranslated = false$. Applying an operational rule on the G would replace the LHS structure found in G (by matching the rule) with the corresponding RHS structure. This rule application also updates the value of marked attributes of nodes in the host graph. The transformation scenario finishes when all elements of G are marked by setting $isTranslated = true$. Note that the application conditions (AC) of a rule are used as the pre- and postcondition of each transformation step. Application conditions might be divided into two kinds: positive application conditions (PACs) and negative ones (NACs). PACs (NACs) require (forbid) the presence of certain structures in the graph for the rule to be applied. Each PAC / NAC also consists of three parts (SL, CL, and TL) corresponding to the parts of AC.

Figure 8 shows on the RHS the current state of the host graph, resulted from applying two rules r_1^F (Class2Table) and r_6^F (SubClass2Table). These rules are specified similarly to the rule r_2 (Attr2Column) as shown in Figure 5. Due to the limited space the specification of these rules is not shown in this paper. Instead of that, we focus on explaining how the r_2^F is applied on this situation. At the current state (w.r.t. the graph on the RHS of Figure 8), the transformation system has not reached the ending state yet because $a1$ has not been transformed ($isTranslated = false$). Searching for an executable rule will find a match for the r_2^F rule, as shown in Figure 6. Note that before matching a rule, each nodes of the LHS could switch its $isTranslated$ from `false` to `true` and its *status* from `1` to `0`. In this case, on the host graph $a1.isTranslated$ becomes `true` so that the rule r_2^F could be matched. The match includes the nodes on the host graph c1, a1, ct1, and t1 corresponding to the LHS of the rule. The transition will end when all elements of the host graph reaching the state $isTranslated = true$.

Note that the settings for the $isTranslated$ is changeable for each derived operational rule. For example, with forward transformation all the elements within the LHS (G^{SR}, G^{CL}, G^{TL}) would have $isTranslated = true$, and all the elements within the RHS have $isTranslated = false$. The attribute $isTranslated$ of all NAC's elements is set to `true` as well. Each node of the host graph is defined with another attribute *status* in order to track rule applications:

- $status = 0$ if the element belongs to either the LHS (i.e., unchanged when applying the rule).
- $status = 1$ if the element belongs to the RHS and is newly created by the rule application.

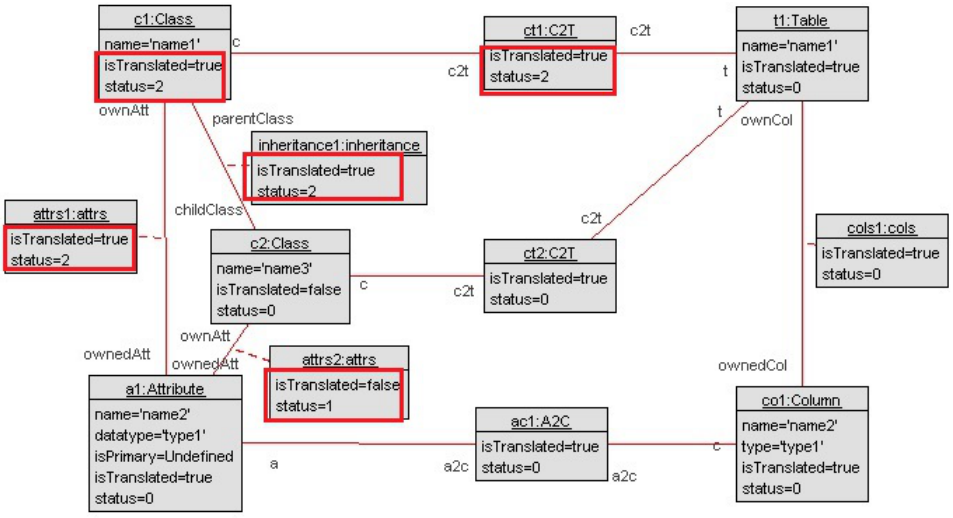

Figure 9. The declarative triple rule *FlatteningInheritance*.

- $status = 2$ if the element is deleted by the rule application, i.e., it belongs to the RHS and does not belong to the LHS.
- $status = -1$ if the element is mentioned in a positive application condition of the rule or the required conditions of the transformation, i.e., positive pre- and postconditions and invariants.
- $status = -2$ if the element is represented in a negative application condition of the rule as well as the forbidden conditions of the transformation, i.e., negative pre- and postconditions and invariants.

The key idea of a forward (backward, integration, and co-evolution) transformation using TGGs is to preserve the given source (target or both) model and to add the missing target and correspondence elements. Original TGG rules have been defined as non-deleting triple rules [11], however, within the context of QVT [26], in many cases, model elements should be removed using deleting rules. In TC4MT, we could specify deleting rules by assigning these attribute-value pairs on a deleted elements: $status = 2$ and $isTranslated = true$.

Example. The rule *FlatteningInheritance* in Figure 9 flattens the inheritance relationships in a class model by first copying the entire attributes ($a1$) of the superclass ($c1$) to attributes of its subclass ($c2$), and then removing the superclass and its attributes.

For forward transformation scenarios, the AC^{SL} of a rule (i.e., AC on G^{SL} of the rule) is part of the precondition of the rule, while the AC^{TL} (i.e., AC on G^{TL} of the rule) becomes part of the postcondition of the rule. A similar mechanism is applicable for backward and integration transformations.

4.3. Concrete Syntax

Requirements of a TC4MT specification are expressed by graph patterns of a typed attributed graph. For example, Figure 10 depicts the declarative rule *PriAttribute2Column*. Figures 13 and 14 depict a negative precondition and a negative invariant, respectively.

5. Quality Assurance of Transformations

Based on our contract-based specification method of transformations, this section introduces several techniques to ensure quality of

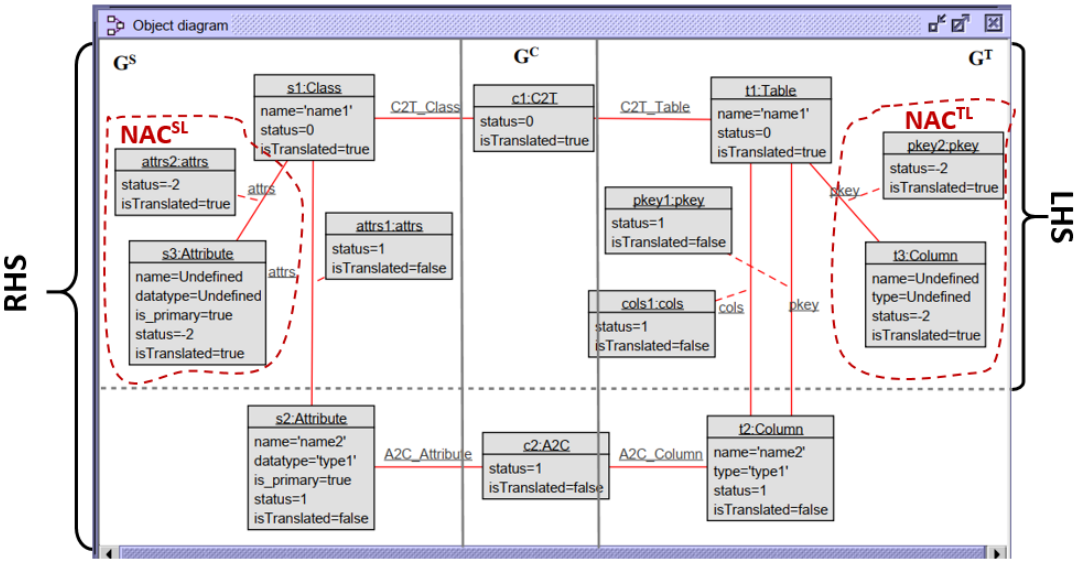

Figure 10. The transformation rule *PriAttribute2Column* in TC4MT.

transformations. First, we explain how to maintain the consistency among contracts by checking their well-formedness. Then, we provide means to on-the-fly verify the fulfillment of contracts during a transformation execution. Finally, we outline a specification-driven testing framework for model transformations.

5.1. Checking the Well-formedness of Contracts

A TC4MT transformation specification is formally represented using contracts in the form of graph patterns. We need to guarantee the well-formedness of contracts for consistency so that such a following error does not occur: A contract may never be satisfied when there exists a negative precondition pattern occurred in the source pattern of a positive invariant, or a precondition included within another precondition. In order to statically detect such errors, based on the formal semantics of TC4MT, we introduce a technique reasoning on i) *metamodel coverage*; ii) *redundancies*; and iii) *contradictions* on contracts, as detailed in the following.

First, obtaining *metamodel coverage* means that any element of source and target metamodels of a transformation is mentioned in at least one contract of the transformation specification. This allows us to check the application domain coverage of the specification. The graphical representation of a TC4MT specification allows us to quickly detect whether an element of source/target metamodels has been referenced by any pattern of the specification and how the element is used (i.e., in positive or negative patterns). All the elements of source and target metamodels as shown in Figure 1 are referenced by at least one transformation contract so that full metamodel coverage is achieved.

Second, *redundancies* in a contract set can be detected by a static analysis. A redundant pattern can be safely dropped to produce a simpler, cleaner set of contracts with the same semantics. Table 1 shows specific situations of redundant contract sets: i) If there exists an inclusion relationship between two graph constructs specifying positive preconditions (or postconditions), then the graph covered within the larger graph will be redundant; ii) If there exists two mutually exclusive negative preconditions (or postconditions), then the graph with the larger graph structure will be redundant; iii) If there exists two negative invariants that share the same source (target)

Table 1. Redundancy detection in specification

Contracts	**Redundancy**
Given two positive pre/postconditions p_1, p_{p2}	$p_1 \subseteq p_2 \Rightarrow p_1$ is redundancy
Given two negative pre/postconditions p_1, p_{p2}	$p_1 \subseteq p_2 \Rightarrow p_2$ is redundancy
Given two positive invariants (rules) $I_1 = \left(p_1^s, p_1^t, p_1^c, LinkS_1, LinkT_1\right)$ $I_2 = \left(p_2^s, p_2^t, p_2^c, LinkS_2, LinkT_2\right)$	$\left(p_1^S = p_2^S\right) \wedge \left(p_1^T \subseteq p_2^T\right) \Rightarrow I_1$ is redundancy $\left(p_1^S \subseteq p_2^S\right) \wedge \left(p_1^T = p_2^T\right) \Rightarrow I_2$ is redundancy
Given two negative invariants $I_1 = \left(p_1^s, p_1^t, p_1^c, LinkS_1, LinkT_1\right)$ $I_2 = \left(p_2^s, p_2^t, p_2^c, LinkS_2, LinkT_2\right)$	$\left(p_1^S = p_2^S\right) \wedge \left(p_1^T \subseteq p_2^T\right) \Rightarrow I_2$ is redundancy $\left(p_1^S \subseteq p_2^S\right) \wedge \left(p_1^T = p_2^T\right) \Rightarrow I_1$ is redundancy

graph structure and the remaining target (source) graph samples are mutually inclusive, the invariant with the larger target graph structure will is redundant; and iv) If two declaration rules have the same source graph pattern, then the rule with a smaller target graph sample will be redundant, otherwise, if two rules have the same target graph pattern but have an inclusive source graph pattern, then the rule having a larger sample of the source graph will be redundant.

Finally, conflicts between contracts can also be statically detected. For example, a conflict can occur if a negative pre- or post-condition is included within a positive pre- or post-condition. The conflict between the two preconditions (affirmative and negative) makes it impossible to exist a model that satisfies the positive contract without violating the negative contract.

5.2. Checking the Fulfillment of Contracts

The paper provides a technique to on-the-fly verify contracts in TC4MT during a transformation execution. The basic idea is to translate contracts of a TC4MT specification into OCL constraints. The Object Management Group (OMG) [13] introduced the Meta-Object Facility (MOF) standard to define metamodels and the Object-Constraint-Language (OCL) standard to define model's constraints in terms of predicate logic. Graphical representation of contracts facilitates the transformation analysis and design while at the transformation implementation phase, OCL contracts are easier to integrated in the transformation implementation languages designed according to the MOF standard such as QVTr [26], ATL [27], RTL [28], and ETL [19]. Classifying terms is an effective instrument used to automatically validate input models and output models of a model transformation as regarded in [29, 30].

```
1  [not]
2  O1type.allInstances()->exists(O1|...
3    O2type.allInstances()->exists(O2|...
4      Ontype.allInstances()->exists(On|
5        conditions(O1,O2...On)...)
```

Figure 11. The OCL schema to translate preconditions.

```
1  O1type.allInstances()->exists(O1|...
2    Ontype.allInstances()->exists(On|
3      conditions(O1,...On)...)
4  and
5  [not]
6  Oktype:allInstances()->exists(Ok|...
7    Omtype.allInstances()->exists(Om|
8      conditions(Ok,...Om)...)
```

Figure 12. The OCL schema to translate invariants.

First, preconditions are translated into boolean OCL constraints as source classifying terms using the OCL schema shown in Figure 11. As illustrated in Figure 11, to translate

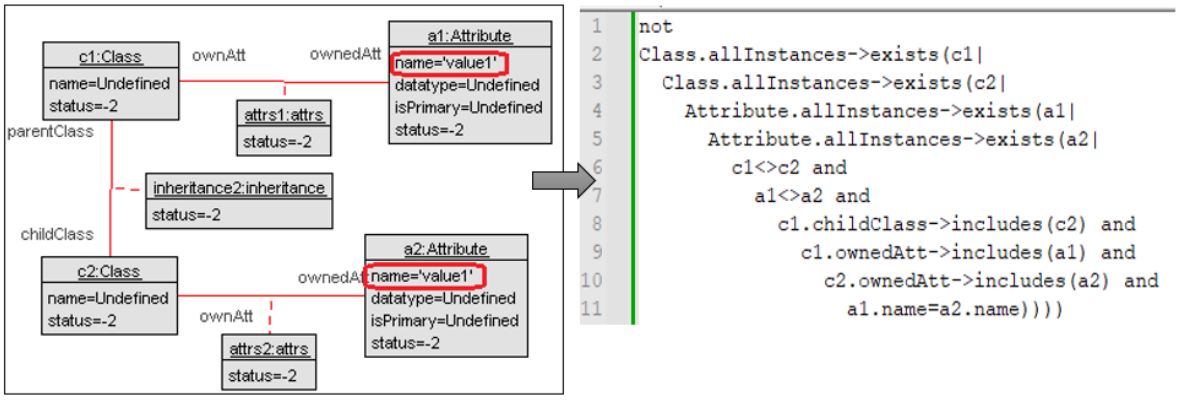

Figure 13. Translating a precondition into the corresponding OCL constraint.

graph patterns into OCL expressions, the OCL schema is matched with the graph pattern of each contract. In the case of negative preconditions, i.e., preconditions contain objects with the attribute $status = -2$, the negation operator **not** appears at the first line of the schema. The function *conditions* is used to check constraints on the underlying objects and their properties: If there exist two objects $o1$ and $o2$ with the same type ($type_{o1} = type_{o2}$) then the condition $o1 <> o2$ will be added. The association between two objects will be translated into a corresponding condition, either $o1.role_2 \rightarrow includes(o2)$ or $o2.role_1 \rightarrow includes(o1)$. We omit the condition to check if an object attribute is undefined. The other OCL constraints of a graph pattern will be included in the function *conditions*. An equal comparison between object attribute values excluding object attributes *status* and $isTranslated$ will be created to represent the $AttributeConstraint$ attribute value constraints, the OCL schema is interested in the above properties and ignores attributes with unspecified value ($Undefined$). Also, if the contract has additional OCL binding expressions for complex expressions, the expression will be added to the *conditions* function of the resulting OCL expression. The partial expressions in *conditions* are combined together by the operator *and*.

Example. Figure 13 illustrates translating a precondition to an OCL expression. Note that since both pre- and postconditions specify constraints on a single modeling domain, the translation of postconditions is performed in a similar way.

Second, we use the OCL schema as shown in Figure 12 to translate invariants into boolean OCL expressions. The OCL scheme for invariants consists of two parts linked together by the *and* operator (line 4). If the source-target contract is a negative invariant, the *not* operator is added before the boolean OCL expression corresponding to the target graph element (line 5). The two parts of the generated OCL expression are created respectively from source and target graphs similarly to the translation of pre- and post-conditions. Figure 14 illustrates the translation of an invariant to two boolean OCL constraints linked by the relational operator 'AND'.

Finally, we explain how to check the correctness of a model transformation with OCL expressions translated from contracts: i) For each pair of OCL expressions translated from positive invariants, if the source model satisfies the OCL expression w.r.t. the source graph, then the target model must also satisfy the OCL expression w.r.t. the target graph; ii) For each pair of OCL expressions translated from negative invariants, if the source model satisfies the OCL expression w.r.t. the source graph, the target model must not satisfy the OCL expression w.r.t. the target graph; and iii) The transformation is correct only if these test conditions are fulfilled.

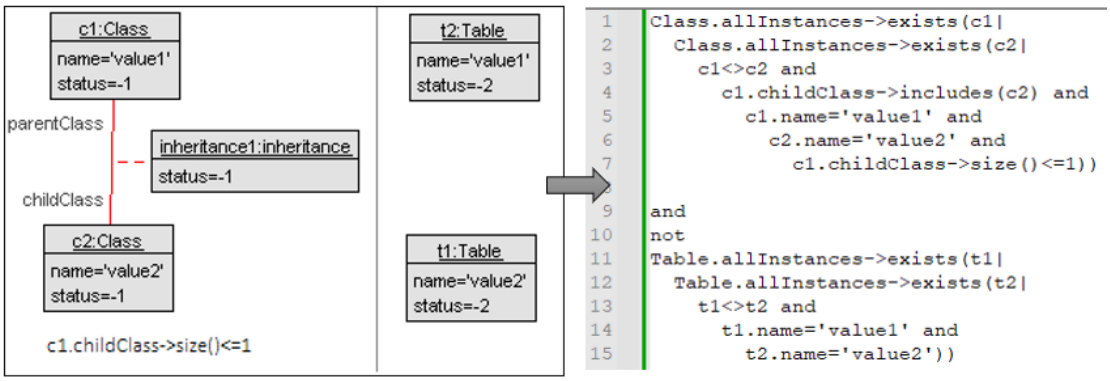

Figure 14. Translating an invariant into the corresponding OCL constraint.

5.3. *Specification-Driven Testing for Model Transformations*

We provide a method to test model transformations as outlined in Figure 15. First, the language TC4MT is offered for the designer to specify transformation requirements. The developer takes as input the TC4MT specification and employs a platform-dependent implementation language (including QVTr [26], ATL [27], and RTL [28]) to implement the transformation, as shown in *label 2a*. The framework also supports the feature to take such a transformation implementation as input in order to obtain a corresponding specification in TC4MT, as illustrated in *label 2b*. Based on a formal semantic of the high-level specification language like TC4MT, it opens a possibility to employ High-Order Transformations (HOTs) as discussed in [14] to realize the bidirectional transformation (w.r.t. *label 2b*).

Second, two methods are introduced to analyze a transformation specification for the test case generation. The first method (as shown in *label 3a*) is partition analysis on the input and output modeling spaces of a model transformation based on metamodels and behavior contracts. Then partition conditions are combined together to generate efficient test cases (avoid duplication and redundancy) with objectives (as shown in *label 4a*): i) maximizing metamodel coverage; ii) maximizing specification coverage; and iii) minimizing number of test cases. The second method is to analyze the transformation rules specifying the dynamic view of a model transformation, to detect rule dependencies that impact on the applicability of the rule chain (as shown in *label 3b*). Applying a grammar testing approach to the TGG of the TC4MT transformation specification, detected rule dependencies can be used to structure applicable rule chains for constructing test case descriptions or inapplicable rule sequences for the exception/robustness testing of model transformations (as shown in *label 4b*). Test case descriptions are graph patterns which also could be translated into boolean OCL constraints similar to graph patterns representing contracts.

Third, in both above testing methods, the input test conditions are expressed by boolean OCL constraints that can be combined according to the test criteria then used to generate test models automatically using OCL constraint solvers (e.g. KodKod [31]), as shown in *labels 5a* and *5b*. Depending on the quality property to be tested, as depicted in *labels 6a* and *6b*, test oracles are constructed by combining boolean OCL constraints representing OCL assertions on expected output models. The benefit of OCL assertions is that they can be combined with the relational operators to form complex output conditions as well as used to evaluate actual output models using model validation tools (e.g. USE model validator [29]), as shown in *label 7*.

Finally, depending on the input test conditions and the corresponding output test oracles

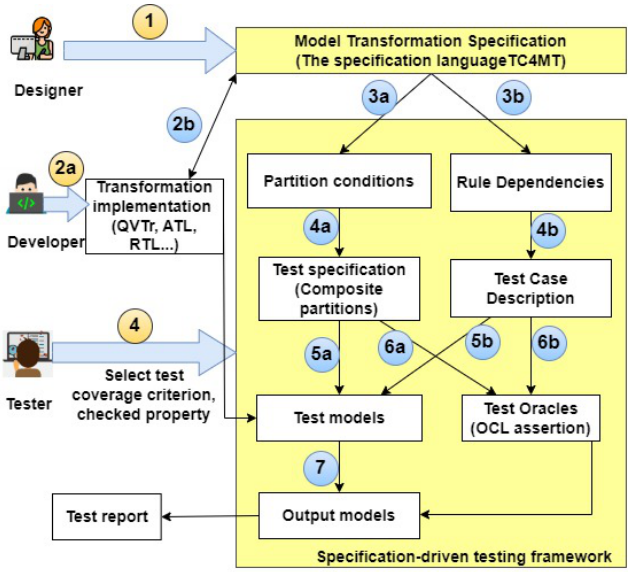

Figure 15. A specification-driven testing framework for model transformations.

designed based on the quality objective to be tested, a test report will be created. The test report indicates passed or failed test cases of the transformation implementation. Failure case analysis helps transformation developers debug the transformation implementation or review the correctness of other artifacts. Details of specification-based testing methods are discussed in our other works.

6. Tool Support

We develop a support tool on the UML-based Specification Environment (USE) [29]. The USE editor allows to represent a transformation definition in the form of a UML class diagram enriched with OCL constraints. Graph patterns of transformation contracts are represented by UML object diagrams conforming to the UML class diagram.

As shown in Figure 10, the pattern of the *PriAttribute2Column* is represented as a flattened triple graph in the form of an object diagram. The preconditions, postconditions, negative invariants, and transformation rules in TC4MT for the CD2RDBM transformation are also captured by object diagrams. These object diagrams are created using the GUI or the Simple OCL-like Imperative Language (SOIL) of the USE editor.

We also develop a plugin for USE as shown in Figure 16 to analyze the TC4MT specification of a transformation in order to define test conditions. First, the window *MetamodelAnalysis* (*red label 1*) helps us to automatically generate source classifying terms [30] from the partition analysis on the source metamodel. Then, the window *SpecificationAnalysis* (*red label 2*) allows us to load patterns of preconditions, postconditions, and invariants (including transformation rules as positive invariants) and translate them into classifying term using the OCL schemes. Finally, with the windows *RuleAnalysis* (*red label 3*) we could load rule specifications and analyze rule dependencies for the test specification.

7. Evaluation

We consider TC4MT as a specification language and adapt from [32] the following three criteria to evaluate it: i) *expressiveness*; ii) *formal definition*; and iii) *usability*. The first one

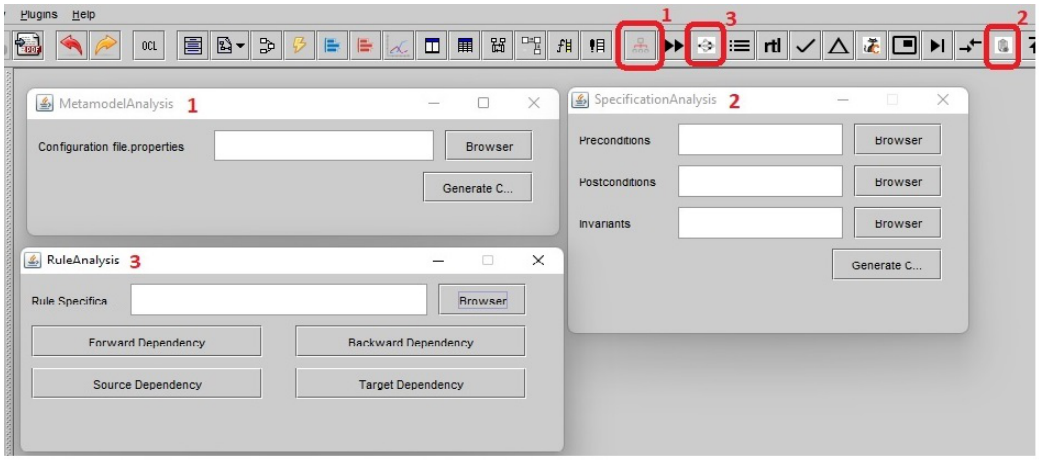

Figure 16. The specification analysis tool.

expressiveness concerns the ability of a specification language to fully and accurately express various aspects of problems within its application domain. The second one *formal definition* concerns with providing a body of knowledge for other activities of the development process, especially in the context of activity automation. The third one *usability* refers to the ability of the language to support stakeholders in terms of intuitive interface capabilities, the availability of tools and environments, and familiarity with the syntax.

7.1. Expressiveness

Expressiveness is one of the most cited properties of model transformation languages as regarded in [33]. This is also a key motivation in the driving force behind the design of domain-specific languages. A language should be able to fully and exactly express all properties of interest of its application domain including complex applications. To evaluate expressive power, we first consider the capability of fully specifying the functional coverage of a model transformation which includes input constraints, output constraints, and input-output constraints. Similar to generic software, functional requirements of a model transformation system also are observed in both data and behavior views which should be specified for requirements analysis purposes. Besides, a transformation specification language needs to support the specification of complex cases of application problems in its application domain. The work in [34] proposed the feature model that makes explicit the different design choices for model transformations with complex features such as bidirectional transformation, tracing, synchronization, and deleting rules.

Based on the functional coverage and the feature model of model transformations, we propose a set of criteria to compare the expressive ability of transformation specification languages as shown in Table 2. Based on the functional coverage and the feature model of model transformations, we propose a set of criteria to compare the expressive ability of transformation specification languages as shown in Table 2: i) Source and target constraints as explained in Section 4.1; ii) Source-target relations: a support for describing the correspondence between source and target models; iii) Element actions, i.e., the actions as explained in the definition of the meta-concept TrafoRule in Section 4.1; iv) Bidirectional: a support for bidirectional transformations; v) Tracing: a support for defining trace elements to maintain the correspondence between source and target models; vi) Rule organization: a support for defining constraints on the order of rule applications; vii) Contract-based, i.e., offering means

Table 2. Compare the expressiveness of TC4MT with that of other specification languages

Features	**TC4MT (2021)**	**Pamomo (2011)**	**DSLTrans (2014)**	**DelTa (2014)**	**TL (2020)**	**MTP (2015)**	**Tract (2012)**
Source constraints	x	x	x	x	x	x	x
Target constraints	x	x	x	x	x	x	x
Source-target relations	x	x	x	x	x	x	x
Element actions	x			x	x	x	
Bidirectional	x	x		x	x		x
Tracing	x	x		x			x
Rule organization			x	x	x		
Contract-based	x	x	x				x

for an on-flying verification. Current specification languages often allow expressing constraints on source models, target models, and relationships between them using the contract-based approach [1–3, 5]. As explained in Section 3, the TC4MT expresses not only behavioral contracts but also protocol contracts for transformations. Therefore, TC4MT could allow us to specify the ordinal relationship between rules, similarly to the proposals explained in DSLTrans [3] and DelTa [4]. Besides, with a formal semantics based on TGG, the TC4MT could maintain the correspondence between source and target models by using trace elements. This also makes it possible to specify bidirectional model transformations and support the model transformation debugging as discussed in [1, 5]. The TL language also supports bidirectional transformation specification, but unlike the other languages using tracing elements in graph transformation rules to express bidirectional transformations, TL uses OCL-based statements to express forward and backward requirement separately.

7.2. Formal Definition

The formal definition of a language is extremely important, especially when the language is handled automatically. There can be varying degrees of formality, resulting in more or less extended possibilities for error detection, validation, verification, and compilation. The most well-known formal property of a programming language or a specification language is that of having a formally defined syntax, which defines the set of allowed expressions. Formal semantics provides a basis to formally define and verify properties such as confluence, deterministic, the equivalence of specifications, and correctness of implementation for a specification.

Accordingly, a specification language must have a solid foundation based on widely accepted mathematical theory, and the different parts of the specification must be compatible with each other. There should be a formally defined mapping between any element of the specification to the mathematical domain that defines its semantics. Specifically, a language specification should have: a formally described syntax; a formally defined interpretation model; and a formally defined mapping between them.

Our specification language TC4MT has all these three characteristics. The syntax of the TC4MT language is formally defined based on the syntax of attribute-typed graphs and the algebra graph transformation technique using rewrite graph rules. We also provide mappings between the meta-modeling and graph typing techniques and mappings between model transformations and corresponding TGG transformations. Again, the TC4MT specification language's semantics is formally defined based on triple graph grammar that is widely used to specify and implement graph transformation systems.

Table 3. Compare the usability of TC4MT with that of other specification languages

Features	**TC4MT (2021)**	**Pamomo (2011)**	**DSLTrans (2014)**	**DelTa (2014)**	**TL (2020)**	**MTP (2015)**	**Tract (2012)**
Supporting MOF	x	x	x	x	x	x	x
Textual syntax	x	x	x	x	x		x
Graphical syntax	x	x	x	x		x	
Pattern-based	x	x	x	x	x		x
Not own symbology		x	x	x	x		

7.3. Usability

The specification language environment should be oriented towards the use by humans. So, specification languages should contain appropriate notations for each stakeholder. A transformation specification language should represent requirements as a knowledge base for other tasks in the development process such as analysis, design, verification, and validation. Table 3 describes the result of comparing TC4MT with other implementation-independent specification languages about usability. The notation **x** in each cell indicates the corresponding feature (listed by the first column) is provided by the corresponding language (listed by the first row).

Current transformation specification languages often support the MOF standard, in which requirements are represented by a structure like the UML class diagram. However, specification approaches regarded in [1, 3, 4] add more graphical symbols to represent patterns instead of using pure UML class diagram structures like TC4MT as well as [6]. Besides, current works often employ OCL to express constraints on models. The works in [2, 5, 7] propose using expression patterns in OCL to express requirements of a transformation. TC4MT and several other specification methods regarded in [1, 24, 35] employ graph patterns in order to express requirements and support the translation of graph patterns into OCL expressions for different purposes, including implementation and verification of transformations.

8. Conclusion

This paper has introduced TC4MT as a high-level transformation specification language and provided a formal semantics for it. We explained the use of the TC4MT specification to verify transformations. Moreover, the TC4MT language allows the designer to precisely specify requirements of a transformation as contracts at all three levels: type contracts, behavioral contracts, and protocol contracts. The TC4MT language is defined based on graph patterns to visually represent transformation contracts. The TGG-based formal semantics of the TC4MT language provides an efficient means for verifying the quality properties of model transformations.

In the context of MDE, “Every thing is a model, even transformations” is a main engineering principle [14]. This poses the need to automatically generate an implementation for a transformation from a specification at a high level like in TC4MT. In future work, we plan to realize the goal by developing high-order transformations between the specification language TC4MT and implementation languages.

Acknowledgements

This work has been supported by Vietnam National University, Hanoi under Project No. QG.20.54. We wish to thank the anonymous reviewers for numerous insightful feedback on the first version of this paper.

References

[1] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, A Visual Specification Language for Model-to-Model Transformations, in: Proc. Int. Symp. Visual Languages and Human-Centric Computing (VL/HCC), IEEE Computer Society, 2010, pp. 119–126, https://doi.org/10.1109/VLHCC.2010.25.

[2] K. Lano, S. Fang, S. K. Rahimi, *TL*: An Abstract Specification Language for Bidirectional Transformations, in: Proc. 23rd Int. Conf. Model Driven Engineering Languages and Systems (MODELS), ACM, 2020, pp. 77:1–77:10, https://doi.org/10.1145/3417990.3419223.

[3] G. M. K. Selim, L. Lucio, J. R. Cordy, J. Dingel, B. J. Oakes, Specification and Verification of Graph-Based Model Transformation Properties, in: Proc. 7th Int. Conf. Graph Transformation (ICGT), Springer, 2014, pp. 113–129, https://doi.org/10.1007/978-3-319-09108-2_8.

[4] H. Ergin, E. Syriani, Towards a Language for Graph-Based Model Transformation Design Patterns, in: Proc. 7th Int. Conf. Theory and Practice of Model Transformations (ICMT), Vol. 8568 of LNCS, Springer, 2014, pp. 91–105, https://doi.org/10.1007/978-3-319-08789-4_7.

[5] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, L. Hamann, Formal Specification and Testing of Model Transformations, in: Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Vol. 7320 of LNCS, Springer, 2012, pp. 399–437, https://doi.org/10.1007/978-3-642-30982-3_11.

[6] A. P. Magalhaes, A. M. S. Andrade, R. S. P. Maciel, On the Specification of Model Transformations through a Platform Independent Approach, in: H. Xu (Ed.), Proc. 27th Int. Conf. Software Engineering and Knowledge Engineering (SEKE), KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015, pp. 558–561, https://doi.org/10.18293/SEKE2015-053.

[7] E. Cariou, N. Belloir, F. Barbier, N. Djemam, OCL Contracts for the Verification of Model Transformations, Electron. Commun. Eur. Assoc. Softw. Sci. Technol., Vol. 24, 2009, pp. 1–15, https://doi.org/10.14279/tuj.eceasst.24.326.

[8] M. Asztalos, L. Lengyel, T. Levendovszky, Formal Specification and Analysis of Functional Properties of Graph Rewriting-Based Model Transformation, Softw. Test. Verification Reliab., Vol. 23, No. 5, 2013, pp. 405–435, https://doi.org/10.1002/stvr.1502.

[9] H. Ehrig, C. Ermel, U. Golas, F. Hermann, Graph and Model Transformation - General Framework and Applications, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2015, https://doi.org/10.1007/978-3-662-47980-3.

[10] J. Greenyer, E. Kindler, Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars, Softw. Syst. Model., Vol. 9, No. 1, 2010, pp. 21–46, https://doi.org/10.1007/s10270-009-0121-8.

[11] A. Schürr, Specification of Graph Translators with Triple Graph Grammars, in: Proc. 20th Int. Conf. Graph-Theoretic Concepts in Computer Science (WG), 1994, pp. 151–163, https://doi.org/10.1007/3-540-59071-4_45.

[12] S. Sen, B. Baudry, J. Mottu, Automatic Model Generation Strategies for Model Transformation Testing, in: Proc. 2nd Int. Cong. Theory and Practice of Model Transformations (ICMT@TOOLS), Vol. 5563 of LNCS, Springer, 2009, pp. 148–164, https://doi.org/10.1007/978-3-642-02408-5_11.

[13] W. Tang, Meta Object Facility, in: Encyclopedia of Database Systems, Springer US, 2009, pp. 1722–1723, https://doi.org/10.1007/978-0-387-39940-9_914.

[14] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Second Edition, Springer Cham, 2017, https://doi.org/10.1007/978-3-031-02549-5.

[15] P. Giner, V. Pelechano, Test-Driven Development of Model Transformations, in: Proc. 12th Int. Conf. Model Driven Engineering Languages and Systems (MODELS), Vol. 5795 of LNCS, Springer, 2009, pp. 748–752, https://doi.org/10.1007/978-3-642-04425-0_61.

[16] J. S. Cuadrado, Towards Interactive, Test-driven Development of Model Transformations, Journal of Object Technology, Vol. 19, No. 3, 2020, pp. 3:1–12, https://doi.org/10.5381/jot.2020.19.3.a18.

[17] J. A. Agirre, G. Sagardui, L. Etxeberria, Model Transformation by Example Driven ATL Transformation Rules Development Using Model Differences, in: A. Holzinger, J. Cardoso, J. Cordeiro, T. Libourel, L. A. Maciaszek, M. van Sinderen (Eds.), Proc. 9th Int. Conf. Software Technologies (ICSOFT), Vol. 555 of CCIS, Springer, 2014, pp. 113–130, https://doi.org/10.1007/978-3-319-25579-8_7.

[18] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, O. M. dos Santos, Engineering Model Transformations with TransML, Softw. Syst. Model., Vol. 12, No. 3, 2013, pp. 555–577, https://doi.org/10.1007/s10270-011-0211-2.

[19] D. S. Kolovos, Establishing Correspondences between Models with the Epsilon Comparison Language, in: Proc. 5th Int. Conf. Model Driven Architecture - Foundations and Applications (ECMDA-FA),

Vol. 5562 of LNCS, Springer, 2009, pp. 146–157, https://doi.org/10.1007/978-3-642-02674-4_11.

[20] B. Meyer, Applying 'Design by Contract', Computer, Vol. 25, No. 10, 1992, pp. 40–51, https://doi.org/10.1109/2.161279.

[21] A. Beugnard, J. Jézéquel, N. Plouzeau, Making Components Contract Aware, Computer, Vol. 32, No. 7, 1999, pp. 38–45, https://doi.org/10.1109/2.774917.

[22] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, D. Varró, Survey and Classification of Model Transformation Tools, Softw. Syst. Model., Vol. 18, No. 4, 2019, pp. 2361–2397, https://doi.org/10.1007/s10270-018-0665-6.

[23] F. Fleurey, B. Baudry, P. Muller, Yves Le Traon, Qualifying Input Test Data for Model Transformations, Software and System Modeling, Vol. 8, No. 2, 2009, pp. 185–203, https://doi.org/10.1007/s10270-007-0074-8.

[24] C. A. González, J. Cabot, Test Data Generation for Model Transformations Combining Partition and Constraint Analysis, in: Proc. 7th Int. Cong. Theory and Practice of Model Transformations (ICMT), Vol. 8568 of LNCS, Springer, 2014, pp. 25–41, https://doi.org/10.1007/978-3-319-08789-4_3.

[25] L. Fritsche, J. Kosiol, A. Möller, A. Schürr, G. Taentzer, A Precedence-Driven Approach for Concurrent Model Synchronization Scenarios using Triple Graph Grammars, in: Software Engineering 2022, Fachtagung des GI-Fachbereichs Softwaretechnik, Vol. P-320 of LNI, Gesellschaft für Informatik e.V., 2022, pp. 27–28, https://doi.org/10.18420/se2022-ws-005.

[26] O. Document, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 1st Edition, Object Management Group, 2016.

[27] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A Model Transformation Tool, Sci. Comput. Program., Vol. 72, No. 1-2, 2008, pp. 31–39, https://doi.org/10.1016/j.scico.2007.08.002.

[28] D.-H. Dang, M. Gogolla, An OCL-Based Framework for Model Transformations, VNU Journal of Science: Computer Science and Communication Engineering, Vol. 32, No. 1, 2016, pp. 44–57, https://doi.org/10.25073/2588-1086/jcsce.120.

[29] M. Gogolla, F. Büttner, M. Richters, USE: A uml-based specification environment for validating UML and OCL, Sci. Comput. Program., Vol. 69, No. 1-3, 2007, pp. 27–34, https://doi.org/10.1016/j.scico.2007.01.013.

[30] F. Hilken, M. Gogolla, L. Burgueño, A. Vallecillo, Testing Models and Model Transformations Using Classifying Terms, Software and System Modeling, Vol. 17, No. 3, 2018, pp. 885–912, https://doi.org/10.1007/s10270-016-0568-3.

[31] R. V. D. Straeten, J. P. Puissant, T. Mens, Assessing the Kodkod Model Finder for Resolving Model Inconsistencies, in: Proc. 7th Int. Cong. Modelling Foundations and Applications (ECMFA), Vol. 6698 of LNCS, Springer, 2011, pp. 69–84, https://doi.org/10.1007/978-3-642-21470-7_6.

[32] J. Bruijning, Evaluation and Integration of Specification Languages, Comput. Networks, Vol. 13, 1987, pp. 75–89, https://doi.org/10.1016/0169-7552(87)90092-4.

[33] S. Götz, M. Tichy, T. Kehrer, Dedicated Model Transformation Languages vs. General-purpose Languages: A Historical Perspective on ATL vs. Java, in: S. Hammoudi, L. F. Pires, E. Seidewitz, R. Soley (Eds.), Proc. 9th Int. Conf. Model-Driven Engineering and Software Development (MODELSWARD), SciTePress, 2021, pp. 122–135, https://doi.org/10.5220/0010340801220135.

[34] K. Czarnecki, S. Helsen, Feature-Based Survey of Model Transformation Approaches, IBM Syst. J., Vol. 45, No. 3, 2006, pp. 621–646, https://doi.org/10.1147/sj.453.0621.

[35] J. Cabot, R. Clarisó, E. Guerra, J. de Lara, Verification and Validation of Declarative Model-to-model Transformations through Invariants, J. Syst. Softw., Vol. 83, No. 2, 2010, pp. 283–302, https://doi.org/10.1016/j.jss.2009.08.012.