Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations *

Hendrik Radke[1], Thorsten Arendt[2], Jan Steffen Becker[1],
Annegret Habel[1], and Gabriele Taentzer[2]

[1] Universität Oldenburg,
{radke,jan.steffen.becker,habel}@informatik.uni-oldenburg.de
[2] Philipps-Universität Marburg,
{arendt,taentzer}@informatik.uni-marburg.de

Abstract. Domain-specific modeling languages (DSMLs) are usually defined by meta-modeling where invariants are defined in the Object Constraint Language (OCL). This approach is purely declarative in the sense that instance construction is not incorporated but has to added. In contrast, graph grammars incorporate the stepwise construction of instances by applying transformation rules. Establishing a formal relation between meta-modeling and graph transformation opens up the possibility to integrate techniques of both fields. This integration can be advantageously used for optimizing DSML definition. Generally, a meta-model is translated to a type graph with a set of nested graph constraints. In this paper, we consider the translation of Essential OCL invariants to nested graph constraints. Building up on a translation of Core OCL invariants, we focus here on the translation of set operations. The main idea is to use the characteristic function of sets to translate set operations to corresponding Boolean operations. We show that a model satisfies an Essential OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint.

Keywords: Meta modeling, Essential OCL, graph constraints, set operations

1 Introduction

Model-based software development causes the need for new, often domain-specific modeling languages (DSMLs) to carry high-level knowledge about the software. Nowadays, DSMLs are typically defined by meta-models following purely the declarative approach. In this approach, language properties are specified by the Object Constraint Language (OCL) [1]. Constructive aspects, however, such as

* This work is partly supported by the German Research Foundation (DFG), Grants HA 2936/4-1 and TA 2941/3-1 (Meta modeling and graph grammars: integration of two paradigms for the definition of visual modeling languages).

generating instances [2,3] for, e.g., testing of model transformations, and recognizing applied edit operations [4] are useful as well to obtain a comprehensive language definition. A constructive way to specify languages, especially textual ones, are grammars. Graph grammars have shown to be suitable and natural to specify (domain-specific) visual languages in a constructive way [5]. They can be used for instance generation, for example.

DSML definition should come along with supporting tools such as model editors and model version management tools. The use of graph grammars for language definition has lead to the idea of generating edit operations from meta-models. In [4], model change recognition as well as model patching are lifted to recognizing and packaging edit operations to patches. To adapt such a general approach to domain-specific needs, complete sets of edit operations have to be specified being able to build up and destroy all models of a DSML. The automatic generation of edit operations from a given meta-model would be of great help.

Given a meta-model, instance generation has been considered by several approaches in the literature. Most of them are **logic-oriented** as, e.g., [2,6]. They translate class models with OCL constraints into logical facts and formulas. Logic approaches such as Alloy [7] can be used for instance generation, as done, e.g., in [6]: After translating a class diagram to Alloy, an instance can be generated or it can be shown that no instances exist. This generation relies on the use of SAT solvers and can also enumerate all possible instances. All these approaches have in common that they translate class models with OCL constraints into logical facts and formulas forgetting about the graph properties of class models and their instances.

In contrast, **graph-based** approaches translate OCL constraints to graph patterns or graph constraints. Following this line, models and meta-models (without OCL constraints) are translated to instance and type graphs. I.e., graph-based approaches keep the graph structure of models as units of abstraction, hence, graph axioms are satisfied by default. In [8], we started to formally translate OCL constraints to nested graph constraints [9]. In this paper, we continue this translation and focus on set operations such as `select`, `collect`, `union` and `size`. Resulting graph constraints can be further translated to application conditions of transformation rules [9]. Especially this work can be advantageously used to translate meta-models (with OCL constraints) to edit operations with all necessary pre-conditions. Meanwhile, Bergmann [10] has implemented a translator of OCL constraints to graph patterns. The focus of that work, however, is not a formal translation but an efficient implementation of constraint checking.

Since graph-based approaches rely on (type and object) graphs, they support flat object sets as the only form of OCL collections to be translated to. In language definition, however, often neither a specific order nor the number of duplicate values is crucial, but the collection of distinct values (see also [6]). Moreover, OCL translation is restricted to a simpler form of meta-model specified by EMOF [12], hence OCL considerations are restricted to Essential OCL being closer to supporting technologies such as the Eclipse Modeling Framework.

Furthermore, considerations are restricted to a first-order, two-valued logic, as done for graph constraints, i.e., the translation is straitened to the corresponding OCL features. However, existing meta-model specifications have shown that this sub-language covers the substantial part to specify well-formedness rules in OCL that are first-order. Since the focus of OCL usage is DSML definition, we further restrict our translation to OCL invariants.

The **contributions** of this paper are the following:

(1) We continue the *translation of OCL* started in [8] and focus on set operations such as `select`, `collect`, `union` and `size`. The main idea for translating constraints with set operations is to use the *characteristic function of sets* which assigns each set operation its corresponding Boolean operation.

(2) We introduce a compact notion of graph conditions, so-called *lax conditions*. They permit the translation of a substantial part of Essential OCL invariants to graph constraints of comparable complexity. Hence, they present a new graphical representation of OCL invariants being slightly more abstract since several navigation paths can be combined in graphs and set operations are reduced to Boolean operations. Lax conditions are extensively used in the OCL translation.

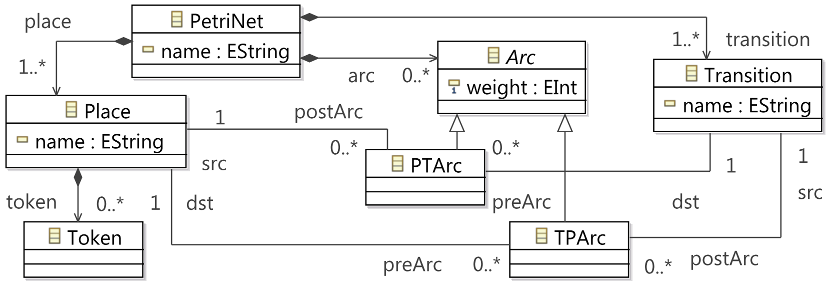
(3) The translation of Essential OCL invariants to nested graph constraints is shown to be *correct*, i.e., a model satisfies an Essential OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint. The aim of this work is to establish a formal relation between meta-modeling and the theory of graph transformation. New contributions in modeling language engineering may be expected by advantageously combining concepts and techniques from both fields.

This paper is structured as follows: The next section presents Essential OCL focusing on set operations. Section 3 recalls typed attributed graphs and graph morphisms as well as nested graph conditions. It also introduces lax conditions as compact notion of graph conditions. Section 4 presents our main contribution of this paper, the translation of Essential OCL invariants to nested graph constraints, more precisely to lax conditions. Section 5 compares to related work and Section 6 concludes the paper. Note that this paper comes along with a long version [11] containing further information about this work, especially the correctness proof.

2 Essential OCL Invariants

In this section, we recall Essential OCL presenting a small example first and formally defining the syntax and semantics thereafter. For illustration purposes, we use the following meta-model for Petri nets.

Example 1. A Petri net (*PetriNet*) is composed of several places (*Place*) and transitions (*Transition*). Arcs between places and transitions are explicit. *PTArc* and *TPArc* are respectively representing place-to-transition arcs and transition-to-place ones. An arc is annotated with a weight. A place can have an arbitrary number of incoming (*preArc*) and outgoing (*postArc*) arcs. In order to model dynamic aspects, places need to be marked with tokens (*Token*).



Despite of multiplicities, this meta-model allows to build inappropriate instances, e.g., one can model a Petri net without any tokens. Therefore, the meta-model has to be complemented with invariants formulated in OCL, e.g.: *There is at least one place in a Petri net having at least one token.*

1. `context PetriNet inv: self.place -> exists(p:Place | p.token -> notEmpty())` or alternatively
2. `context PetriNet inv: self.place -> select(p:Place | p.token -> notEmpty()) -> notEmpty()` or alternatively
3. `context PetriNet inv: self.place -> collect(p:Place | p.token) -> notEmpty()`.

Essential OCL. The Object Constraint Language (OCL) [1] is a formal language used to describe expressions on object-oriented models being consistent to either the Meta Object Facility (MOF) [12] or the Unified Modeling Language (UML) specifications of the OMG. These expressions typically specify invariant conditions that must hold for the system being modeled (see Example 1) or queries over objects described in a model. Whereas our preceding work [8] concentrates on a restricted version of OCL, called Core OCL, that addresses the OCL type system, navigation concepts, and the usage of invariants, we now widen our approach to Essential OCL. According to [1], Essential OCL is "…the minimal OCL required to work with EMOF". Essential MOF (EMOF) is a subset of MOF that allows to define simple meta-models using simple concepts.

The translation presented in this paper covers a substantial part the OCL specification. Compared to [8], we now support a significant number of set operations (e.g., `select`, `collect`, `includesAll`, and `union`). In contrast to the OCL specification, we use a two-valued logic. Furthermore, and the only kind of collections we consider are sets which seem to conform well with using OCL for metamodeling (i.e., we do not consider bags, sequences, ordered sets, and tuples).

Formalization. We describe the semantics of Essential OCL based on the formal definitions included in the OCL specification [1], Annex A being based on the doctoral thesis by Richters [13]. Due to space limitations, we recall the main definitions and concepts only. For deeper considerations, we refer to the long version of this paper [11] as well as to the documents mentioned above. As a first preliminary step, we define an *object model* representing the EMOF-based meta-model types as follows.

Definition 1 (Object Model). Let $DSIG = (S, OP)$ be a data signature with $S = \{Integer, Real, Boolean, String\}$ and corresponding operation symbols OP. An *object model* over $DSIG$ is a structure $M = (CLASS, ENUM, ATT, ASSOC, associates, r_{src}, r_{tgt}, multiplicities, \prec)$ consisting of finite sets of classes ($CLASS$), enumerations ($ENUM$), and associations between classes ($ASSOC$), a family of attributes for each class (ATT), functions for mapping each association to a pair of participating classes ($associates$), to a source respectively target role name (r_{src} and r_{tgt}), and to a multiplicity specification for each association end ($multiplicities$), and finally a partial order on $CLASS$ reflecting its generalization hierarchy ($\prec$).

Since the evaluation of an OCL invariant requires knowledge about the complete context of an object model at a discrete point in time, we recall the definition of a *system state* of an object model M as follows.

Definition 2 (System State). A *system state* of an object model M is a structure $\sigma(M) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$ consisting of a finite set of class objects (σ_{CLASS}), functions assigning attribute values to each class object for each attribute (σ_{ATT}), and a finite set of links connecting class objects (σ_{ASSOC}). The set $States(M)$ consists of all system states $\sigma(M)$ of M.

Based on the formal definition of an object model, the underlying type system (*signature*) for expressions in Essential OCL is defined as follows:

Definition 3 (Signature). A *signature* over an object model M is a structure $\Sigma_M = (T_M, \leq_M, \Omega_M)$. T_M is a set of types consisting of basic types S, all class types $CLASS$, all enumeration types $ENUM$, the collection type *Set(t)* for an arbitrary $t \in T_M$, and *OclAny* as super type of all other types except for *Set(t)*. $\leq_M$ is partial order on T_M representing a type hierarchy. Ω_M is a set of operations on T_M consisting of OP, ATT, appropriate association end operations, set operations such as $isEmpty$, $includesAll$, $size$, and $union$, and operations equality ($=$) and non-equality ($\neq$) for all types $t \in T_M$. The *semantics of a data signature* is based on sets and functions. It is fully presented in [11].

Definition 4 (Essential OCL Expressions). Let $\Sigma_M = (T_M, \leq_M, \Omega_M)$ be a signature over an object model M. Let $Var = \{Var_t\}_{t \in T_M}$ be a family of variable sets indexed by types $t \in T_M$. The family of *Essential OCL expressions* over Σ_M is given by $Expr = \{Expr_t\}_{t \in T_M}$ representing sets of expressions. Expressions in $Expr$ are `VariableExpressions` $v \in Expr_t$ for each variable $v \in Var_t$, `OperationExpressions` $e := \omega(e_1, \cdots, e_n) \in Expr_t$ for each operation symbol $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_M$ and for all $e_i \in Expr_{t_i}(1 \leq i \leq n)$, `IfExpressions`: $e :=$ **if** e_1 **then** e_2 **else** $e_3 \in Expr_{Boolean}$ for all $e_1, e_2, e_3 \in Expr_{Boolean}$, `TypeExpressions` such as $e.oclIsTypeOf(t') \in Expr_{Boolean}$ for $e \in Expr_t$ and some types t' and t, and `IteratorExpressions` such as $s \to forAll(v \mid b) \in Expr_{Boolean}$ and $s \to select(v \mid b) \in Expr_{Set(t)}$ for $s \in Expr_{Set(t)}, v \in Var_t$, and $b \in Expr_{Boolean}$. The *semantics of an Essential OCL expression* $e \in Expr_t$ is a function $I[\![e]\!] : Env \to I(t)$ with Env being pairs of system states and variable assignments and $I(t)$ the set of elements of type t. The complete semantics definition can be found in the long version of this paper [11].

As mentioned above, we concentrate on invariants being formulated in Essential OCL. Therefore, we consider invariants and OCL constraints as synonyms in the remainder of this paper.

Definition 5 (Essential OCL Invariant). An *Essential OCL invariant* is a Boolean OCL expression with a free variable $v \in Var_C$ where C is a classifier type. The concrete syntax of an invariant is: `context v:C inv : <expr>`. The set $Invariant_M$ denotes the set of all Essential OCL invariants over M.

Remark 1. An invariant `context v:C inv: expr` is equivalent to expression `C.allInstances -> forAll(v|expr)`. Consequently, the semantics of an invariant is equal to the semantics of the equivalent Essential OCL expression.

3 Nested Graph Constraints

In the following, we recall the main ingredients of typed, attributed graphs. Their formal definition is presented in [14] and recalled in [11]. They form the basis to define typed attributed nested graph constraints. Attributed graphs as considered here allow to attribute nodes only while the original version [14] supports also the attribution of edges.

Definition 6 (Attributed graph). An *A-graph* $G = (G_V, G_D, G_E, G_A, src_G, tgt_G, src_A, tgt_A)$ consists of sets G_V and G_D, called graph and data nodes (or vertices), respectively, G_E and G_A, called graph and node attribute edges, respectively, and source and target functions for graph and attribute edges. A-graph morphisms are defined componentwise. Let $DSIG = (S, OP)$ be a data signature with a family X of variables, and $T_{DSIG}(X)$ the term algebra w.r.t. $DSIG$ and X. An attributed graph is is a tuple $AG = (G, D, \Phi)$ where G is an A-graph, D is a $DSIG$-algebra with $\sum_{s \in S} D_s = G_D$, and Φ is a finite set of $DSIG$-formulas[3] with free variables in X. An *attributed graph morphism* between two attributed graphs consists of an A-graph morphism and a $DSIG$-homomorphism such that codomain formulas follow from corresponding domain formulas.

[3] $DSIG$-formulas are meant to be $DSIG$-terms of sort BOOL.

This definition is closely related to symbolic graphs [15]. Attributed graphs in the sense of [16] correspond to attributed graphs with an empty sets of formulas.

Definition 7 (Typed attributed graph). An *attributed type graph* ATGI $= (TG, Z, \{false\})$ consists of an A-graph TG and a final $DSIG$-algebra Z and a simple (i.e. containing neither multiple edges nor loops) inheritance graph I. The *(inheritance) clan* of a type is the set of all its sub-types (including itself); the clan of a node (a graph) is the clan of its type (all its node's types). A *typed attributed graph* $(AG, type)$ over ATGI, short ATGI-*graph*, consists of an attributed graph AG and a *morphism type* : $AG \to$ ATGI. Given two ATGI-graphs $AG^1 = (G^1, type^1)$ and $AG^2 = (G^2, type^2)$, an ATGI-*morphism* $f\colon AG^1 \to AG^2$ is an attributed graph morphism such that $type^2 \circ f = type^1$.

Typed attributed graphs and morphisms form a category. In [8], attributed graphs over attributed type graphs with inheritance [14] are considered as well.

Graph conditions [17,18] are nested constructs which can be represented as trees of morphisms equipped with quantifiers and Boolean connectives. In the following, we introduce ATGI-conditions as injective conditions over ATGI-graphs[4], closely related to attributed graph constraints [15] and E-conditions [19]. Graph conditions are implemented e.g. in the systems AGG, GROOVE, and GrGen.

Definition 8 (Nested graph conditions). A *(nested) graph condition* on typed attributed graphs, short *condition*, over a graph P is of the form *true* or $\exists(a, c)$ where $a\colon P \to C$ is an injective morphism and c is a condition over C. Boolean formulas over conditions over P yield conditions over P, that is, for conditions c, c_i $(i \in I)$ over P, $\neg c$ and $\bigwedge_{i\in I} c_i$ are conditions over P. Conditions over the empty graph $\emptyset$ are called *constraints*. In the context of rules, conditions are called *application conditions*.

Notation. Graph conditions may be written in a more compact form: $\exists a$ abbreviates $\exists(a, true)$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, *false* abbreviates $\neg true$, $\bigvee_{i\in I} c_i$ abbreviates $\neg \bigwedge_{i\in I} \neg c_i$, $c \Rightarrow c'$ abbreviates $\neg c \vee c'$, $c \Leftrightarrow c'$ abbreviates $(c \Rightarrow c') \wedge (c' \Rightarrow c)$, and $c \veebar c'$ abbreviates $(c \wedge \neg c') \vee (\neg c \wedge c')$.

The satisfaction of a condition is established by the presence and absence of certain morphisms from the graphs within the condition to the tested graph. The presented *injective* satisfiability notion restricts these morphisms to be injective: no identification of nodes and edges is allowed. In this way, explicit counting such as the existence/non-existence of n nodes is easily expressible.

Definition 9 (Semantics). *Satisfiability* of a condition over P by an injective morphism $p\colon P \to G$ is inductively defined as follows: p satisfies *true*. $p\colon P \to G$ satisfies $\exists(P \overset{a}{\to} C, c)$ if there exists an injective morphism $q\colon C \to G$ such that $p = q \circ a$ and q satisfies c.

$$\begin{tikzcd}
\exists\ P \arrow[rr, "a"] \arrow[dr, "p"'] & {=} & C \arrow[dl, "q"] & \triangleleft c \\
& G &
\end{tikzcd}$$

[4] A graph condition is *injective* if it is built by injective morphisms.

For Boolean formulas over conditions, the semantics is as usual: p satisfies $\neg c$ if p does not satisfy c, and p satisfies $\bigwedge_{i\in I} c_i$ if p satisfies each c_i $(i \in I)$. We write $p \models c$ if $p\colon P \to G$ satisfies the condition c over P. *Satisfiability* of a constraint, i.e. a condition over the empty graph $\emptyset$, by a graph is defined as follows: A graph G satisfies a constraint c, short $G \models c$, if the injective morphism $p\colon \emptyset \to G$ satisfies c. Two conditions c and c' over P are *equivalent*, denoted $c \equiv c'$, if, for all injective morphisms $p\colon P \to G$, $p \models c$ iff $p \models c'$.

The definition of conditions is very rigid. In the following, we will be more flexible and consider so-called lax conditions based on inclusions.

Definition 10 (Lax conditions). A *lax condition* on typed attributed graphs is of the form *true* or $\exists(C, c)$ where C is a graph and c is a lax condition. Boolean formulas over lax conditions yield lax conditions. $\exists(C)$ abbreviates $\exists(C, \mathit{true})$.

Convention. Lax conditions are drawn as follows: Graphs in lax conditions are drawn in a standard way: Nodes are depicted by rectangles $\boxed{\text{v:T}}$ carrying the node name v (or, more general, a set of names) and its type T inside. In the case of $\{u, v\}$, we write $u = v$ inside the rectangle. Edges are drawn by arrows pointing from the source to the target node and the edge label is placed next to the arrow. Inclusions are given by the names of the nodes: Two occurrences of v in different graphs of the lax condition, e.g. $\exists(\boxed{\text{v}}, \exists(\boxed{\text{v}}, c))$ or $\exists(\boxed{\text{u}}, \exists(\boxed{\text{u=v}}))$, mean that they are in inclusion relation.

The semantics of lax conditions is defined by the semantics of conditions. For this purpose, we "complete" lax conditions to conditions.

Construction (From lax conditions to conditions). For a graph P and a lax condition d, Complete(P, d) denotes the condition over P, inductively defined as follows:

$$\begin{tikzcd}
\emptyset \arrow[r, dotted] \arrow[d, dotted] & C' \arrow[d, "b"] & \triangleleft \arrow[d, phantom, "c"] \\
P \arrow[r, "a"'] & C & \triangleleft
\end{tikzcd}$$

Complete$(P, \mathit{true}) = \mathit{true}$.
Complete$(P, \exists(C', c)) = \bigvee_{(a,b)\in\mathcal{F}} \exists(P \overset{a}{\to} C, \text{Complete}(C, c))$
where $\mathcal{F} = \{(a, b) \mid (a, b)$ jointly surjective, a, b inclusions.$\}$.[5]
Complete$(P, \neg c) = \neg$Complete(P, c).
Complete$(P, \wedge_{i\in J} c_i) = \wedge_{i\in J}$Complete$(P, c_i)$.

Definition 11 (Semantic of lax conditions). *Satisfiability* of a lax condition is defined by the satisfiability of the corresponding condition: For an injective morphism $p\colon P \to G$ and a lax condition c, $p \models c$ iff $p \models$ Complete(P, c). Two lax conditions c and c' are *equivalent*, denoted $c \equiv c'$, if, the corresponding conditions are equivalent.

[5] A pair of morphisms (a, b) is *jointly surjective* if, for each $x \in C$, there is a preimage $y \in P$ with $a(y) = x$ or a preimage $z \in C'$ with $b(z) = x$.

By definition, lax conditions and nested graph conditions have the same expressive power.

Example 2. The lax condition $\exists(\boxed{\mathrm{u}}, \exists(\boxed{\mathrm{v}}, \exists(\boxed{\mathrm{u}}\xrightarrow{\mathrm{role}}\boxed{\mathrm{v}})))$ means that there exist two nodes and an edge of type `role` in between. Its completion over the empty graph $\emptyset$ yields the condition $\exists(\emptyset \to \boxed{\mathrm{x}}, \exists(\boxed{\mathrm{u}} \to \boxed{\mathrm{u}}\boxed{\mathrm{v}}, \exists(\boxed{\mathrm{u}}\boxed{\mathrm{v}} \to \boxed{\mathrm{u}}\xrightarrow{\mathrm{role}}\boxed{\mathrm{v}})) \vee \exists(\boxed{\mathrm{u}} \to \boxed{\mathrm{u{=}v}}, \mathrm{false})) \equiv \exists(\emptyset \to \boxed{\mathrm{u}}, \exists(\boxed{\mathrm{u}} \to \boxed{\mathrm{u}}\boxed{\mathrm{v}}, \exists(\boxed{\mathrm{u}}\boxed{\mathrm{v}} \to \boxed{\mathrm{u}}\xrightarrow{\mathrm{role}}\boxed{\mathrm{v}})))$. It is equivalent to lax conditions $\exists(\boxed{\mathrm{u}}\ \boxed{\mathrm{v}}, \exists(\boxed{\mathrm{u}}\xrightarrow{\mathrm{role}}\boxed{\mathrm{v}}))$ and $\exists(\boxed{\mathrm{u}}\xrightarrow{\mathrm{role}}\boxed{\mathrm{v}})$.

Since lax conditions can be transformed into conditions automatically, lax conditions are also called conditions somewhat ambiguously.

The following equivalences can be used to simplify lax conditions.

Fact 1 (Equivalences). Let $C_1 \oplus_P C_2$ denote the gluing or pushout of C_1 and C_2 along P and let $\mathcal{P}$ denote the set of all intersections of C_1 and C_2.

(E1) (a) $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2)$.
(b) $\exists(C_1, \exists(C_2)) \equiv \exists(C_1{+}C_2)$ if C_1 and C_2 are clan-disjoint[6].
(c) $\exists(C_1, \exists(C_2)) \equiv \exists(C_2)$ if $C_1 \subseteq C_2$ and $\equiv \exists(C_1)$ if $C_2 \subseteq C_1$.

(E2) (a) $\exists(C_1, \exists(C_2) \wedge \exists(C_3)) \equiv \exists(C_1, \bigvee_{P\in\mathcal{P}} \exists(C_2 \oplus_P C_3))$, if for all node names occuring in both C_2 and C_3, a node with that name already exists in C_1.
(b) $\exists(C_1) \wedge \exists(C_2) \equiv \exists(C_1 + C_2)$ if C_1 and C_2 are clan-disjoint and have disjoint sets of node names.

(E3) $\exists(\boxed{\mathrm{u{:}T}}, \exists(C) \wedge \exists(\boxed{\mathrm{u{=}v{:}T}})) \equiv \exists(\boxed{\mathrm{u{:}T}}, \exists(C[u{=}v]))$ provided that either u or v does not exist in C and $C[u{=}v]$ is the graph obtained from C by renaming u by $u = v$.

4 Translation of Essential OCL Invariants

To translate Essential OCL invariants, we first show how to translate the type information of meta-models, i.e. object models, to attributed type graphs with inheritance [14]. Thereafter, system states are translated to typed attributed graphs. Having these ingredients available, our main contribution, the translation of Essential OCL invariants is presented and illustrated by several examples. Finally, the correctness of the translation is stated.

Type and state correspondences. To translate Essential OCL invariants to nested graph constraints, we relate an object model M to an attributed type graph $ATGI$. Correspondence relation $corr_{type}$ relates classes of M to graph vertices of $ATGI$, attributes to attribute vertices and associations to graph edges of $ATGI$. Data signatures of M and $ATGI$ are almost the same. The only

[6] Two graphs C_1 and C_2 are *clan-disjoint* if the clans of the types of C_1 and C_2 are disjoint. For graphs C_1 and C_2, $C_1{+}C_2$ denotes the disjoint union.

difference are enumerations of M which are mapped to new sorts for type graphs as well as to new equality and inequality operations.

Given such a type correspondence $corr_{type}$, a system state $\sigma(M)$ corresponds to an attributed graph AG typed over ATGI if there is a *state correspondence relation* $corr_{state}$ bijectively relating classes to graph vertices, attributes to attribute vertices, and links to graph edges of AG.

The formal definitions for these correspondences can be found in [11].

Methodology of the translation. In the following, we present the translation of a substantial part of Essential OCL to nested conditions. This translation is shown to correspond to the one given earlier in [8] and furthermore, it is proven to be correct in [11].

- The translation proceeds along the abstract syntax tree of the OCL constraint. For example, given `a->union(b)->notEmpty()`, we first translate `notEmpty`, followed by `union` and then its arguments `a` and `b`.
- The set operations themselves are translated with the characteristic function in mind, e.g. the characteristic function of `a->union(b)` is the disjunction of the characteristic functions of `a` and `b`: $v \in A \cup B$ iff $v \in A \vee v \in B$. Navigation expressions, which yield a single object, are treated like single-element sets.
- When translating an OCL operation which yields a set of objects (translation tr_S), we pass a single node as an extra parameter serving as representative of the set: $tr_S(\texttt{a->union(b)}, \boxed{\text{v:T}}) := tr_S(\texttt{a}, \boxed{\text{v:T}}) \vee tr_S(\texttt{b}, \boxed{\text{v:T}})$.

Representing sets by their characteristic function allows us to translate OCL set operations without a special set construct in the conditions. For example, we can express `expr1->exists(v:T | expr2)` as "there exists an object v of type T such that v is element of the set described by `expr1` *and* v satisfies `expr2`", and `expr1->forall(v:T | expr2)` as "for all nodes v of type T, *if* v is in the set described by `expr1` *then* v also satisfies `expr2`". Sets A and B are equal if every node v is in A iff it is in B. The idea behind `select` is to restrict the set of nodes described by `expr1` such that each node v′ satisfying `expr1` also satisfies `expr2`. $tr_S(\texttt{expr1->collect(v:T|expr2)}, \boxed{\text{v':T'}})$ is true iff there is a node v that is (a) contained in the set described by `expr1` and (b) the relation between v and v′ given by `expr2` is satisfied. For `T.allInstances()`, the characteristic function is true for all nodes which are of type T.

Without loss of generality, we assume variable names to be unique in OCL expressions. This can easily be ensured by giving each variable a different name, e.g. `self.a->collect(v | v.b)->exists(v | expr)` becomes `self.a->collect(v | v.b)->exists(v' | expr)`.

The translation consists of several parts: Invariants are translated by function tr_I. OCL expressions yielding a Boolean as result are translated by tr_E. We use tr_N for expressions yielding single objects and tr_S for expressions yielding collections (i.e., sets) of objects.

Definition 12 (Constraint translation). Let M be an object model as defined above with ATGI = $corr_{type}(M)$ being the corresponding attributed type graph. Let t : $Expr \to T$ be a typing function which returns the type of an OCL expression. Let $\mathrm{Invariant}_M$ be the set of Essential OCL invariants over M and $\mathrm{GraphCondition}_{\mathrm{ATGI}}$ be the set of all graph constraints as defined in Definition 8. The *translation functions*

- invariant translation tr_I: $\mathrm{Invariant}_M \to \mathrm{GraphCondition}_{\mathrm{ATGI}}$,
- expression translation tr_E: $Expr_{Boolean} \to \mathrm{GraphCondition}_{\mathrm{ATGI}}$,
- navigation translation tr_N: $Expr_{\mathrm{C}} \times \mathrm{Graph}_{\mathrm{ATGI}} \to \mathrm{GraphCondition}_{\mathrm{ATGI}}$ with $\mathrm{C} \in CLASS$,
- and set translation tr_S: $Expr_{Set} \times \mathrm{Graph}_{\mathrm{ATGI}} \to \mathrm{GraphCondition}_{\mathrm{ATGI}}$

are defined as follows:

Let `expr`, `expr1` and `expr2` be OCL expressions, $\mathrm{u}, \mathrm{v}, \mathrm{v}'$ names of nodes (i.e. variables), $\mathrm{T} = \mathrm{t(v)}$ denote the type of v and likewise $\mathrm{T}' = \mathrm{t(v')}$, `attr1` and `attr2` be attribute names, op $\in \{<, >, \leq, \geq, =, <>\}$ a comparison operator, and role be a role of a class. Then

1. (a) $tr_I(\texttt{context C inv: expr}) := \forall(\boxed{\mathrm{self{:}C}}, tr_E(\texttt{expr}))$
   (b) $tr_I(\texttt{context var:C inv: expr}) := \forall(\boxed{\mathrm{var{:}C}}, tr_E(\texttt{expr}))$
2. Translation of Boolean operators is unambiguous: $tr_E(\texttt{not expr}) := \neg tr_E(\texttt{expr})$, $tr_E(\texttt{expr1 and expr2}) := tr_E(\texttt{expr1}) \wedge tr_E(\texttt{expr2})$ and similar for operators `true`, `or`, `implies` and `if`.
3. (a) $tr_E(\texttt{expr1->exists(v:T | expr2)}) :=$
   $\exists(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \wedge tr_E(\texttt{expr2}))$
   (b) $tr_E(\texttt{expr1->forall(v:T|expr2)}) :=$
   $\forall(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \Rightarrow tr_E(\texttt{expr2}))$
4. $tr_E(\texttt{expr1->includesAll(expr2)}) :=$
   $\forall(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr2}, \boxed{\mathrm{v{:}T}}) \Rightarrow tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}))$
   where $\mathrm{t}(\texttt{expr1}) = \mathrm{t}(\texttt{expr2}) = \texttt{Set(T)}$.
   The translation of `excludesAll` is analogous.
5. $tr_E(\texttt{expr->notEmpty()}) := \exists(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr}, \boxed{\mathrm{v{:}T}}))$
6. $tr_E(\texttt{expr->size() >= } n) := \exists(\boxed{\mathrm{v}_1\mathrm{:T}} \cdots \boxed{\mathrm{v}_n\mathrm{:T}}, \bigwedge_{i=1}^{n} tr_S(\texttt{expr}, \boxed{\mathrm{v}_i\mathrm{:T}}))$
   where n is an integer constant ≥ 0, $\mathrm{t}(\texttt{expr}) = \texttt{Set(T)}$ and $\mathrm{v}_1, \ldots, \mathrm{v}_n$ are fresh variables of type T.
7. (a) $tr_E(\texttt{expr1 = expr2}) := \exists(\boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \wedge \mathrm{tr}_N(\texttt{expr2}, \boxed{\mathrm{v{:}T}}))$
   if $\mathrm{t}(\texttt{expr1}) = \mathrm{t}(\texttt{expr2}) = \mathrm{T}$ for some class T,
   (b) $tr_E(\texttt{expr1 = expr2}) := \forall(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \Leftrightarrow tr_S(\texttt{expr2}, \boxed{\mathrm{v{:}T}}))$
   if $\mathrm{t}(\texttt{expr1}) = \mathrm{t}(\texttt{expr2}) = \texttt{Set(T)}$ for some class T.
8. $tr_E(\texttt{expr.attr1 op con}) := \exists(\boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v{:}T}}) \wedge \exists(\boxed{\begin{array}{c}\mathrm{v{:}T} \\ \hline \mathrm{attr1\ op\ con}\end{array}}))$
   where con is a constant and $\mathrm{t}(\texttt{expr}) = \mathrm{T}$ for some class T.

9. $tr_E(\texttt{expr1.attr1 op expr2.attr2}) :=$
   $\exists(\boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr1}, \boxed{\begin{array}{c}\mathrm{v{:}T}\\ \hline \mathrm{attr1\ op\ x}\end{array}}) \wedge tr_N(\texttt{expr2}, \boxed{\begin{array}{c}\mathrm{v{:}T}\\ \hline \mathrm{attr2 = x}\end{array}})) \vee$[7]
   $\exists(\boxed{\mathrm{v{:}T}}\ \boxed{\mathrm{v'{:}T'}}, tr_N(\texttt{expr1}, \boxed{\begin{array}{c}\mathrm{v{:}T}\\ \hline \mathrm{attr1\ op\ x}\end{array}}) \wedge tr_N(\texttt{expr2}, \boxed{\begin{array}{c}\mathrm{v'{:}t(v')}\\ \hline \mathrm{attr2 = x}\end{array}}))$
   where $\mathrm{t}(\texttt{expr1}) = \mathrm{T}$, $\mathrm{t}(\texttt{expr2}) = \mathrm{T}'$, $\mathrm{t}(\mathrm{x}) = \mathrm{t}(\texttt{attr1}) = \mathrm{t}(\texttt{attr2})$ and x, v and v' are fresh variables.
10. (a) $tr_E(\texttt{expr.oclIsKindOf(T)}) := \exists(\boxed{\mathrm{v{:}T'}} \hookrightarrow \boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v{:}T'}}))$
    (b) $tr_E(\texttt{expr.oclIsTypeOf(T)}) :=$
    $\exists(\boxed{\mathrm{v{:}T'}} \hookrightarrow \boxed{\mathrm{v{:}T}}, \bigwedge_{T'' \in clan(T)}^{T'' \neq T} \neg\exists(\boxed{\mathrm{v{:}T}} \hookrightarrow \boxed{\mathrm{v{:}T''}}) \wedge tr_N(\texttt{expr}, \boxed{\mathrm{v{:}T'}}))$
    where $\mathrm{T}' = \mathrm{t}(\texttt{expr})$ and $\mathrm{T} \in clan(\mathrm{T}')$.
11. $tr_N(\texttt{expr.oclAsType(T)}, \boxed{\mathrm{v{:}T}}) := \exists(\boxed{\mathrm{v{:}T'}} \hookrightarrow \boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v{:}T'}}))$
    where $\mathrm{T}' = \mathrm{t}(\texttt{expr})$ and $\mathrm{T} \in clan(\mathrm{T}')$
12. (a) $tr_N(\texttt{v}, \boxed{\mathrm{v'{:}T}}) := \exists(\boxed{\mathrm{v{=}v'{:}T}})$ if `v` is a variable,
    (b) If `role` has a multiplicity of 1, $tr_N(\texttt{expr.role}, \boxed{\mathrm{v{:}T}}) :=$
    $\exists(\boxed{\mathrm{v'{:}T'}} \xrightarrow{\mathrm{role}} \boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v'{:}T'}}))$ if $\mathrm{T}' \notin clan(\mathrm{T})$ and
    $\exists(\boxed{\mathrm{v'{:}T'}} \xrightarrow{\mathrm{role}} \boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v'{:}T'}})) \vee \exists(\boxed{\mathrm{v{:}T}}\circlearrowleft{\scriptstyle\mathrm{role}}, tr_N(\texttt{expr}, \boxed{\mathrm{v{:}T}}))$ else.
    (c) If `role` has a multiplicity > 1, $tr_S(\texttt{expr.role}, \boxed{\mathrm{v{:}T}}) :=$
    $\exists(\boxed{\mathrm{v'{:}T'}} \xrightarrow{\mathrm{role}} \boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v'{:}T'}}))$ if $\mathrm{T}' \notin clan(\mathrm{T})$ and
    $\exists(\boxed{\mathrm{v'{:}T'}} \xrightarrow{\mathrm{role}} \boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr}, \boxed{\mathrm{v'{:}T'}})) \vee \exists(\boxed{\mathrm{v{:}T}}\circlearrowleft{\scriptstyle\mathrm{role}}, tr_N(\texttt{expr}, \boxed{\mathrm{v{:}T}}))$ else,
    where v' is a fresh variable and $\mathrm{t}(\texttt{expr}) = \mathrm{T}'$[8].
13. $tr_S(\texttt{expr1->select(v:T | expr2)}, \boxed{\mathrm{v'{:}T}}) :=$
    $tr_S(\texttt{expr1}, \boxed{\mathrm{v'{:}T}}) \wedge tr_E(\texttt{expr2})\{\mathrm{v}/\mathrm{v}'\}$ where $\texttt{expr2}\{\mathrm{v}/\mathrm{v}'\}$ means replacing `v` in `expr2` with $\texttt{v}'$.
    The translation of `reject` proceeds analogously.
14. (a) $tr_S(\texttt{expr1->collect(v:T | expr2)}, \boxed{\mathrm{v'{:}T'}}) :=$
    $\exists(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \wedge tr_S(\texttt{expr2}, \boxed{\mathrm{v'{:}T'}}))$ if `expr2` yields a set, and
    (b) $tr_S(\texttt{expr1->collect(v:T | expr2)}, \boxed{\mathrm{v'{:}T'}}) :=$
    $\exists(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \wedge tr_N(\texttt{expr2}, \boxed{\mathrm{v'{:}T'}}))$ if `expr2` yields an object.
15. $tr_S(\texttt{expr1->union(expr2)}, \boxed{\mathrm{v{:}T}}) := tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \vee tr_S(\texttt{expr2}, \boxed{\mathrm{v{:}T}})$
    Transformations for `intersect`, `-` (set difference) and `symmetricDifference` are analogous, using $a \wedge b$, $a \wedge \neg b$ and $a \veebar b$ instead of $a \vee b$, respectively.
16. $tr_S(\texttt{T.allInstances()}, \boxed{\mathrm{v{:}T}}) := \exists(\boxed{\mathrm{v{:}T}})$
17. $tr_S(\texttt{Set\{expr1, ..., exprN\}}, \boxed{\mathrm{v{:}T}}) :=$
    $tr_N(\texttt{expr1}, \boxed{\mathrm{v{:}T}}) \vee \cdots \vee tr_N(\texttt{exprN}, \boxed{\mathrm{v{:}T}})$
    where $\texttt{expr1}, \ldots, \texttt{exprN}$ are OCL expressions of type T.

[7] The part before $\vee$ is omitted if $clan(\mathrm{t}(\texttt{expr1})) \cap clan(\mathrm{t}(\texttt{expr2})) = \emptyset$, and the part after $\vee$ is omitted if $\texttt{expr1} = \texttt{expr2}$.

[8] Case (a) presents the final step in a chain of navigations, while cases (b) and (c) present the navigation to single nodes and sets of nodes, respectively. Translations (b) and (c) are identical, since single nodes are treated as single-element sets.

Further translations of Essential OCL constraints can be derived from equivalences of OCL expressions. Most of these equivalences follow from basic set theory and logic axioms, cf. Richters [13]. Such equivalences include operations `includes`, `excludes`, `including`, `excluding`, `<>`, `isEmpty`, `expr->size op n` for `op` in `>,=,<=,<,<>`, `any` and `one`.

Example 3. To demonstrate our approach, we translate the second alternative of invariant *There is at least one place in a Petri net having at least one token* presented in Example 1. Note that translating each alternative leads to the same graph constraint, as shown in [11].

$$tr_I(\texttt{context PetriNet inv:}$$
$$\quad \texttt{self.place->select(p:Place|p.token->notEmpty())->notEmpty()}) =^{1}$$
$$\forall(\boxed{\text{self:PN}}, tr_E(\texttt{self.place->select(p:Place|p.token->notEmpty())->notEmpty()})) =^{5}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, tr_S(\texttt{self.place->select(p:Place|p.token->notEmpty())}, \boxed{\text{p:Pl}}))) =^{13}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, tr_S(\texttt{self.place}, \boxed{\text{p:Pl}}) \wedge tr_E(\texttt{p.token->notEmpty()}))) =^{5}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, tr_S(\texttt{self.place}, \boxed{\text{p:Pl}}) \wedge \exists(\boxed{\text{t:Tk}}, tr_S(\texttt{p.token}, \boxed{\text{t:Tk}})))) =^{12}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p:Pl}}) \wedge \exists(\boxed{\text{t:Tk}}, \exists(\boxed{\text{p:Pl}}\xrightarrow{\text{token}}\boxed{\text{t:Tk}})))) \equiv^{E1,E2}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p:Pl}}\xrightarrow{\text{token}}\boxed{\text{t:Tk}}))$$

An index above the = sign refers to the translation rule used; an index at the equivalence sign ≡ refers to the used equivalence rule of Proposition 1.

Example 4 (Further invariant translations).
The name of a transition is not empty.

$$tr_I(\texttt{context Transition inv: self.name <> ''}) = \forall(\boxed{\text{self:Tr}}, \exists(\boxed{\begin{array}{c}\text{self:Tr}\\ \hline \text{name <> ' '}\end{array}}))$$

There is no isolated place.

$$tr_I(\texttt{context Place inv:self.preArc->notEmpty() or self.postArc->notEmpty()}) =$$
$$\forall(\boxed{\text{self:Pl}}, \exists(\boxed{\text{self:Pl}}\xrightarrow{\text{preArc}}\boxed{\text{v:TPArc}}) \vee \exists(\boxed{\text{self:Pl}}\xrightarrow{\text{postArc}}\boxed{\text{w:PTArc}}))$$

Each two places of a Petri net have different names.

$$tr_I(\texttt{context PetriNet inv:}$$
$$\quad \texttt{self.place->forAll(p1,p2:Place | p1<>p2 implies p1.name <> p2.name)}) =$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}}\begin{array}{l}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}\\ \xrightarrow[\text{place}]{}\boxed{\text{p2:Pl}}\end{array}) \Rightarrow \exists(\boxed{\begin{array}{c}\text{p1:Pl}\\ \hline \text{name<>x}\end{array}}\boxed{\begin{array}{c}\text{p2:Pl}\\ \hline \text{name=x}\end{array}}))$$

The translations of Core OCL constraints in [8] (in this paper denoted tr') and the translation tr of Essential OCL constraints are closely related, as stated by the following proposition.

Proposition 1 (Translations of Core and Essential OCL). For every Core OCL constraint `expr`, $tr'(\texttt{expr}) \equiv tr(\texttt{expr})$.

Proof. The proof of this proposition is given in [11]. □

To show that the translation of Essential OCL invariants is correct, we consider their semantics and the semantics of graph constraints. If an invariant holds for a system state, the corresponding graph constraint is fulfilled by the corresponding graph.

Theorem 1 (Correct Translation of Essential OCL invariants). Given an object model M and its corresponding attributed type graph ATGI $= corr_{type}(M)$, for all Essential OCL invariants $inv \in dom(tr_I)$ and all environments $(\sigma, \beta) \in Env$,

$$I[\![\texttt{inv}]\!](\sigma, \beta) = true \text{ iff } G = corr_{state}(\sigma) \models tr_I(inv).$$

Proof. The proof of this theorem is given in [11]. □

Limitations. Since we focus on the use of OCL within DSML definitions, we restrict our translation to *invariants*. Therefore, we do not consider expression `oclIsNew` that is mainly used within post-condition specifications of operations.

Because graph-based approaches rely on (type and object) graphs, they support *flat object sets* as the only form of OCL collections to be translated. Consequently, we do not translate expressions related to further collection types (e.g., `Sequence`) such as `sortedBy` and `isUnique` as well as expressions related to hierarchical sets (e.g., `flatten`) and sets of primitive values (e.g., `sum`).

Since graph constraints are restricted to a *first-order, two-valued logic*, our OCL translation is straightened to corresponding OCL features, focusing on the equivalence of constraints to *true* in our proofs. Therefore, we do not consider types `void` and `invalid` as well as expressions like `oclIsUndefined` and `iterate` which is not first order.

Finally, there are a few additional OCL features which have not been covered by our OCL translation but will be in future work. These are, e.g., non-recursive operation calls, as used in model queries, and `LetExpressions` which may be iteratively replaced by their bodies with potential variable replacement. Also, set operations `any` and `one` are not handled yet.

5 Related Work

In the literature, there are several approaches to translate OCL to formal frameworks. Most of them are logic-oriented; they translate class models with OCL invariants into logical facts and formulas. An overview on the significant logic-oriented approaches is given in [8]. The advantage of the logic-oriented approaches is that there are a number of established theorem provers which can be used.

In contrast to logic-oriented approaches, graph-based approaches translate OCL constraints to graph patterns or graph constraints. Pennemann has shown in [20] that a theorem prover for graph conditions works more efficient than theorem provers for logical formulas being applied to graph conditions. The key idea is here that graph axioms are always satisfied by default when using a theorem prover for graph conditions. Lambers and Orejas [21] have shown that this theorem prover is also complete. Bergmann [10] has translated OCL constraints to graph patterns. He considers a pretty similar subset of OCL than we do (except of OCL expression not being first-order), and in fact, the way of translation shows a lot of similarities. The focus of that work, however, is not a formal translation but an efficient implementation of constraint checking which is tested at example constraints.

6 Conclusion

Translating Essential OCL invariants to nested graph constraints opens up a way to construct application conditions of transformation rules ensuring consistency already during transformations [9]. This missing link between meta-modeling and transformation systems may be advantageously used by new applications such as test model generation as well as recognition and auto-completion of model editing operations. The backward translation of graph conditions to OCL may also be interesting, e.g., to weakest pre-conditions in OCL as proposed in [22]. In future work, we plan to implement the presented translation of OCL to application conditions in the context of the Eclipse Modeling Framework and Henshin [23], a model transformation environment based on graph transformation concepts, and to apply it in various forms.

Acknowledgement. We are grateful to the anonymous referees for their helpful comments on a draft version of this paper.

References

1. OMG: Object Constraint Language. `http://www.omg.org/spec/OCL/`
2. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). (2007) 547–548
3. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and System Modeling **8**(4) (2009) 479–500
4. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In Denney, E., Bultan, T., Zeller, A., eds.: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, IEEE (2013) 191–201
5. Bardohl, R., Minas, M., Schürr, A., Taentzer, G.: Application of Graph Transformation to Visual Languages. In: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2. World Scientific (1999) 105–180

6. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: Model Driven Engineering Languages and Systems - 15th Int. Conference, MODELS 2012, Proceedings. Volume 7590 of LNCS., Springer (2012) 415–431
7. Jackson, D.: Alloy Analyzer website (2012) http://alloy.mit.edu/.
8. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From Core OCL Invariants to Nested Graph Constraints. In: Graph Transformations (ICGT 2014). Volume 8571 of LNCS. (2014) 97–112 Extended version at: http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk/pdfbi/bi2014-01.pdf.
9. Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. Mathematical Structures in Computer Science **19** (2009) 245–296
10. Bergmann, G.: Translating OCL to Graph Patterns. In Dingel, J., Schulte, W., Ramos, I., Abraho, S., Insfran, E., eds.: Model-Driven Engineering Languages and Systems (MoDELS). Volume 8767 of LNCS. Springer (2014) 670–686
11. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations: Long version (2015) Available at: http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk/pdfbi/bi2015-01.pdf.
12. OMG: Meta Object Facility. http://www.omg.org/spec/MOF/
13. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin (2002)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental Theory of Typed Attributed Graph Transformation based on Adhesive HLR Categories. Fundamenta Informaticae **74(1)** (2006) 31–61
15. Orejas, F.: Symbolic Graphs for Attributed Graph Constraints. J. Symb. Comput. **46**(3) (2011) 294–315
16. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. Springer (2006)
17. Rensink, A.: Representing first-order logic by graphs. In: Graph Transformations (ICGT'04). Volume 3256 of LNCS. (2004) 319–335
18. Habel, A., Pennemann, K.H.: Nested constraints and application conditions for high-level structures. In: Formal Methods in Software and System Modeling. Volume 3393 of LNCS. (2005) 293–308
19. Poskitt, C.M., Plump, D.: Hoare-Style Verification of Graph Programs. Fundamenta Informaticae **118(1-2)** (2012) 135–175
20. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, Universität Oldenburg (2009)
21. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Graph Transformation (ICGT 2014). Volume 8571 of LNCS. (2014) 17–32
22. Richa, E., Borde, E., Pautet, L., Bordin, M., Ruiz, J.F.: Towards Testing Model Transformation Chains Using Precondition Construction in Algebraic Graph Transformation. In: AMT 2014–Analysis of Model Transformations Workshop Proceedings. (2014) 34–43
23. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In: Model Driven Engineering Languages and Systems, 13th International Conference, MoDELS 2010, Oslo, Norway. Proceedings. Volume 6394 of LNCS., Springer (2010) 121–135