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

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

* 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).

Object Constraint Language (OCL) [1]. Constructive aspects, however, such as 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 [11], 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. The translation of graph constraints to application conditions of rules is sketched in Section 5. Section 6 compares to related work and Section 7 concludes the paper.

2 Essential OCL Invariants

In this section, we recall Essential OCL presenting a small example first and formally defining the syntax and semantics thereafter, according to the work by Richters [12] that went into the OCL specification by the OMG [1].

2.1 An example meta-model including OCL invariants

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*).

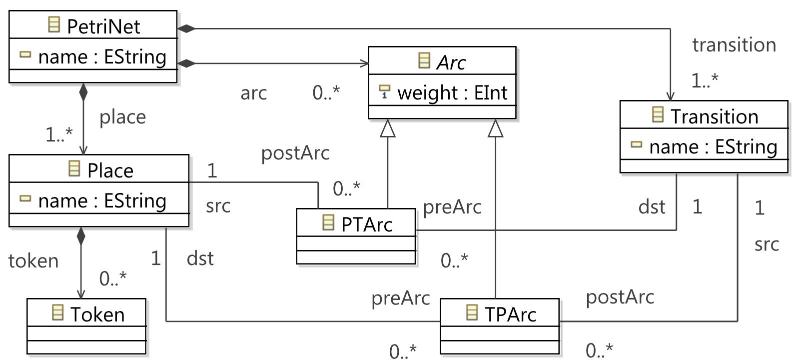


Fig. 1. Meta-model for Petri nets (adapted from [13])

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.:

1. The name of a transition is not empty.
   `context Transition inv: self.name <> ''`
2. There is no isolated transition.
   `context Transition inv: self.preArc -> notEmpty() or self.postArc -> notEmpty()` or alternatively
   `context PetriNet inv: self.transition -> forAll(t:Transition | t.preArc -> notEmpty() or t.postArc -> notEmpty())`
3. There is no isolated place.
   (a) `context Place inv: self.preArc -> notEmpty() or self.postArc -> notEmpty()` or alternatively
   (b) `context Place inv: PTArc.allInstances() -> collect(src) -> union(TPArc.allInstances() -> collect(dst)) -> includes(self)`
4. Each two places of a Petri net have different names.
   (a) `context PetriNet inv: self.place -> forAll(p1:Place | self.place -> forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))` or alternatively

(b) `context PetriNet inv: self.place -> forAll(p1:Place,p2:Place | p1 <> p2 implies p1.name <> p2.name)`

5. There is at least one place in a Petri net having at least one token.
   (a) `context PetriNet inv: self.place -> exists(p:Place | p.token -> notEmpty())` or alternatively
   (b) `context PetriNet inv: self.place -> select(p:Place | p.token -> notEmpty()) -> notEmpty()` or alternatively
   (c) `context PetriNet inv: self.place -> collect(p:Place | p.token) -> notEmpty()` or alternatively
   (d) `context PetriNet inv: Token.allInstances() -> notEmpty()` [3]
6. The weight of an arc is positive.
   `context Arc inv: self.weight >= 1`
7. There is at least one transition that can be fired, i.e., all PTArcs targeting this transition must have a weight less or equal to the token number of their source places.
   ```
   context PetriNet inv: self.transition -> exists(t:Transition |
   t.preArc -> forAll(a:PTArc | a.weight <= (a.src.token -> size())))
   ```

In the following, we list further OCL invariants which conform to the Petri net meta-model in Figure 1. Please note, that these invariants may be not appropriate to model proper Petri nets. Instead, we use them to demonstrate additional translations of invariants to nested graph constraints compared to those presented in the preceding paper [8].

8. Each Petri net has at least two places.
   `context Petrinet inv: self.place->size() >= 2`
9. Each place is both source and destination of corresponding place-to-transition respectively transition-to-place arcs.
   ```
   context PetriNet inv:
   (self.place = (PTArc.allInstances() -> collect(src))) and
   (self.place = (TPArc.allInstances() -> collect(dst)))
   ```

2.2 OCL language description

In the following, we give a more detailed but rather informal description of the OCL language. After summarizing the OCL type system, we discuss issues with respect to navigating OCL expressions and dealing with the three-valued logic in OCL. Finally, we present selected operations on the OCL collection types.

[3] We assume, that a Petri net model consists of only one single instance of type `PetriNet` representing the root element of the model with respect to the containment hierarchy.

The OCL type system The type system in OCL mainly consists of three categories: custom types, predefined types, and template types. *Custom types* are either class types or enumeration types defined by the user in the corresponding meta-model. For example, the Petri net meta-model shown in Figure 1 defines the custom class types `PetriNet`, `Transition`, `Place`, `Token`, `PTArc`, `TPArc`, and `Arc` [4]. For all custom types, OCL provides basic operations like equality (`=`) and inequality (`<>`) as used in invariant 1 of Example 1 (`self.name <> ''`). *Predefined types* are `Integer`, `Real`, `String`, and `Boolean`, called *primitive data types*. They are used as attribute types in meta-models, as for example in attribute `Arc::weight::Integer` (see Figure 1) [5]. Again, basic operations depending on the concrete type are provided. For instance, the invariants of Example 1 use (in)equality operators on String (`self.name <> ''` in invariant 1), logical operations on Boolean (logical `or` in invariant 2), and relational operators on Integer (`self.weight >= 1` in invariant 6). Furthermore, OCL has two special predefined types representing the top (`OclAny`) and bottom (`OclVoid`) elements of the corresponding type hierarchy. *Template types* are `Collection(T)` and `Tuple(`T_1`,`T_2`)` whose parameters `T`, T_1, and T_2 are applied to other types. Please note, that collection is an abstract type. Its concrete subtypes are `Set`, `OrderedSet`, `Bag`, and `Sequence` and differ with respect to frequency and ordering of the contained elements. In this paper, we concentrate on sets and bags only since we consider graph structures which, in the basic sense, do not include ordering features.

Navigating OCL expressions In OCL expressions, object structures can be traversed using the so-called *dot notation*. Accessible elements are objects (i.e., class instances) and their features (i.e., attributes respectively opposite association ends). Depending on the feature's multiplicity (for example, `1` and `0..1` on the one hand, `1..*` and `0..*` on the other hand), a navigation results either in a single-valued return type (i.e., custom or predefined type) or in a multi-valued type, more precisely in a set. In invariant 1 of Example 1, e.g., navigation `self.name` [6] results in a single value of predefined type `String` whereas in invariant 3 (a) navigation `self.preArc` [7] yields a set of type `PTArc`. If in a given Petri net no such incoming arc exists, the navigation from the corresponding transition results in an *empty* set whereas, in the case of multiplicity `0..1`, the absence of an appropriate value yields `null` representing the only value of bottom type `OclVoid`. Further navigation from a multi-valued result using a second dot yields a bag-valued result. For example, navigating `somePetriNet.arc.weight` would return a bag since different arcs could have the same weight which consequently has to be returned multiple times. Please note, that this kind of navigation is only a shorthand for the `collect` operation described later on in this section.

[4] Please note, that class `Arc` is abstract as can be seen by its name written in italics. As a consequence, it is impossible to model instance elements having `Arc` as type.

[5] The meta-model in Figure 1 uses the EMF representation `EInt` of type `Integer`.

[6] In this invariant, variable `self` represents an instance of type `Transition`.

[7] Again, variable `self` represents an instance of type `Transition`.

Logic in OCL We mentioned above that (1) OclVoid is a subtype of any custom and predefined type, i.e., it is also a subtype of the predefined type Boolean, and that (2) OclVoid consists of value null. As a consequence, OCL type Boolean comes along with a three-valued logic, i.e., Boolean={true,false,null}. The following operations are provided: and, or, not, xor, and implies. Among others, we use the logical or in invariant 2 and the implication in invariant 4 of Example 1. Moreover, OCL has a universal quantifier forAll and an existential quantifier exists, both in the spirit of first order logic. Consequently, both quantifiers range over finite collections only and cannot be used, for example, on all instances of the type Integer or String [14]. Invariant 4 uses the universal quantifier to express that for each pair of places within the Petri net the corresponding names are distinct: forAll(p1,p2:Place | p1 <> p2 implies p1.name <> p2.name). The existential quantifier is used in invariants 5 (a) and 7, for example.

OCL collection type operations In this section, we give a rough overview on some selected but substantial predefined collection type operations which are called by the arrow-notation (for example, someSet->foo()). They can be categorized into construction, conversion, filter, extraction, and Boolean operations. *Construction operations* are either explicit type constructors like Set{...} and Bag{...} or one of the implicit constructors including(e) and excluding(e). An implicit constructor takes an element e as parameter and adds it to a given collection (including) respectively removes all occurrences of it from a given collection (excluding). *Conversion operations* like asSet() and asBag() allow to convert one collection kind into any of the other three collection kinds. *Filter operations* like select(BExp), reject(BExp), and any(BExp) are used to filter collection elements according to the evaluation of the Boolean expression BExp inside the brackets. For example, somePetriNet.arc->select(weight=1) filters all those arcs from the arc set of a given Petri net carrying the standard weight 1 whereas somePetriNet.arc->any(weight=1) non-deterministically returns one such arc. We use the selection operation in invariant 5 (b) of Example 1. *Extraction operations* extract some information from the given collection except for Boolean values. Examples of this kind of operations are size(), collect(BExp), and union(Collection(T)). size() returns the number of elements within the collection. collect(...) can be used to construct new collections (with potentially other type elements) from existing ones. For example, somePetriNet.arc->collect(weight) returns a bag (!) of Integer values. The operations collect and size are used in invariants 5 (c) and 7, respectively. Finally, there are many operations returning Boolean values. For checking the existence of elements within a collection, operations isEmpty() respectively notEmpty() can be used. We use the latter one in invariant 3 (a) of Example 1, for example. In order to test membership in collections the operations includes(e) and excludes(e) testing on single elements e as well as includesAll(Collection(T)) and excludesAll(Collection(T)) for testing element collections are available.

2.3 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) [11] 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. Considering EMOF as underlying structure means that the type system we address in this paper enhances the one in the preceding paper [8] by also considering enumeration types and allowing arbitrary multiplicities on association ends, i.e., multiplicities range between lower and upper bounds. As a consequence, this leads to single objects instead of sets of objects for upper bound 1 multiplicities (`0..1` and `1..1`, respectively) which is now also considered in the translation to graph constraints. However, we differ from the EMOF type system in two minor issues. On the one hand, we do not consider class operations since our aim is to translate invariants only [8]. On the other hand, for simplicity reasons it is still appropriate that roles are the default ones indicating source and target.

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 meta-modeling (i.e., we do not consider bags, sequences, ordered sets, and tuples).

Abstract syntax Figure 2 shows the substantial part of the Essential OCL meta-model we consider in this paper. The meta-classes are embedded in the corresponding EMOF meta-model whose meta-classes have a gray-colored background. As illustrated in the left part of Figure 2, the EMOF type system is extended by the special type `AnyType` and by the specific collection type `SetType`. An invariant (respectively `Constraint`) on a `Classifier` is defined by an `ExpressionInOCL` that owns a contextual `Variable` which is named *self* in most cases. Since OCL is a strongly typed language [9], the context variable is typed by the constrained element of the invariant whereas the invariant itself has type *Boolean* which is a concrete implementation of `PrimitiveType`. The concrete specification of an invariant is given by a subclass of `OclExpression`. In general, such a subclass is either

[8] We do not consider pre-/postconditions of class operations, for example.

[9] Indeed, all meta-classes are direct or indirect sub classes of `TypedElement` (see right part of Figure 2).

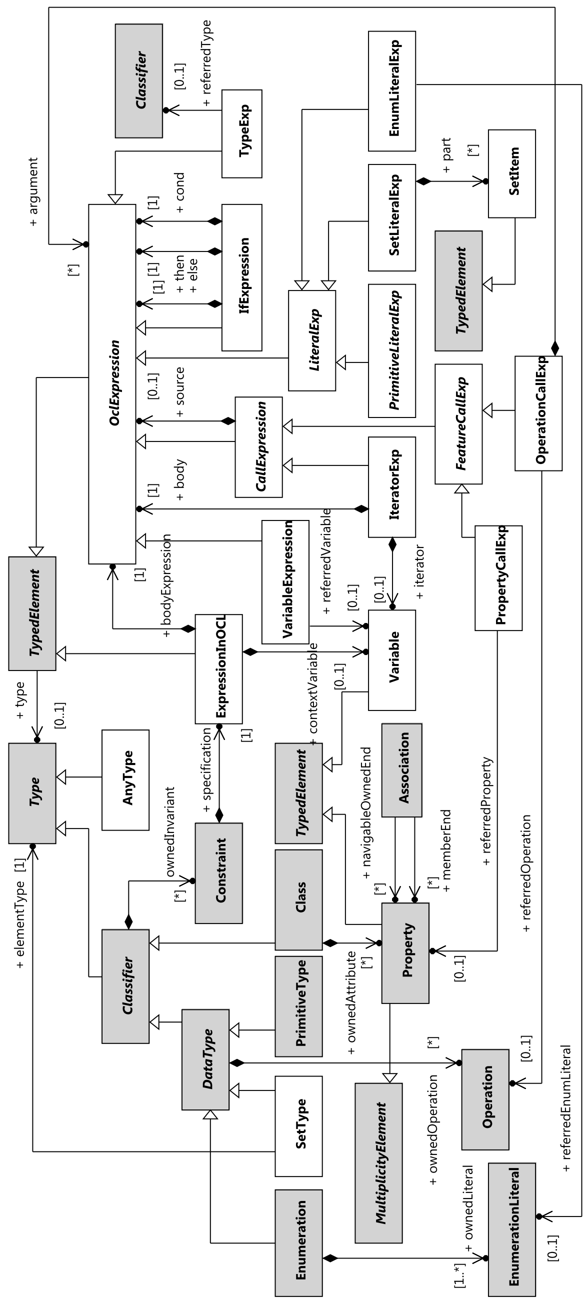


Fig. 2. The considered part of the Essential OCL meta-model

- a `VariableExpression` to refer to a variable,
- a `PrimitiveLiteralExp` to refer to a primitive type literal, e.g., String *foo*,
- an `OperationCallExp` to refer to an operation of a primitive type like the addition of integers, or to a set type operation like *isEmpty*,
- a `PropertyCallExp` to enable navigation to class attributes (typed by a primitive type or an enumeration) or to association ends (typed by a class), both represented as instances of meta-type `Property`,
- a `SetLiteralExp` to refer to a set of model elements which are represented by meta-type `SetItem`,
- an `EnumLiteralExp` to refer to an enumeration literal,
- a `TypeExp` to provide type checking and type casting,
- an `IfExpression` to provide conditional expressions, or finally
- an `IteratorExp` representing a looping execution on each element of a given set (used in *exists* and *forAll*).

Semantics 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 [12]. We prefer this formalization, in contrast to the UML-based specification in the main part, since it is more suitable for proving the semantic preservation of our translation later on in this paper. Due to space limitations, we recall the main definitions and concepts only. For deeper considerations, we refer 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 $M = (CLASS, ENUM, ATT, ASSOC, associates, r_{src}, r_{tgt}, multiplicities, \prec)$ where

- $CLASS$ is a finite set of classes,
- $ENUM$ is a finite set of enumerations where each enumeration $E \in ENUM$ is associated with a non-empty but finite set of enumeration literals by function $literals(E) = \{e_1^E, \ldots, e_n^E\}$,
- $ATT = \{ATT_c\}_{c \in CLASS}$ is a family of attributes $att : c \to (S \cup ENUM)$ of class c,
- $ASSOC$ is a set of associations,
- $associates : ASSOC \to (CLASS \times CLASS)$ is a function that maps each association to a pair of participating classes,
- $r_{src}, r_{tgt} : ASSOC \to String$ are functions that map each association to a source respectively target role name with $r_{src}(assoc) = c_1$ and $r_{tgt}(assoc) = c_2$ for each $assoc \in ASSOC$ with $associates(assoc) = (c_1, c_2)$,
- $multiplicities : ASSOC \to (\mathcal{P}(\mathbb{N}_0) \times \mathcal{P}(\mathbb{N}_0))$ [10] is a function assigning each association end a multiplicity specification with $multiplicities(assoc) =$

[10] $\mathcal{P}(\mathbb{N}_0)$ denotes the power set of the natural numbers. However, we consider intervals with lower and upper bound only.

(M_1, M_2), $M_1 \neq \{0\}$ [11], and $M_2 \neq \{0\}$ for each $assoc \in ASSOC$ with $associates(assoc) = (c_1, c_2)$,
– and $\prec$ is a partial order on $CLASS$ reflecting its generalization hierarchy.

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. Informally, a system state consists of a set of class objects, functions assigning attribute values to each class object for each attribute, and a finite set of links connecting class objects within the model.

Definition 2 (System State). A *system state* of an object model M is a structure $\sigma(M) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$ where

– for each class $c \in CLASS$, $\sigma_{CLASS}(c)$ is a finite subset of the (infinite) set of object identifiers $oid(c) = \{\underline{c}_1, \underline{c}_2, \dots\}$,
– for each attribute $att : c \to t \in ATT_c^{\prec}$, $\sigma_{ATT}(att) : \sigma_{CLASS}(c) \to I(t)$ is an operation from class objects to some interpretation of type $t \in (S \cup ENUM)$ where $ATT_c^{\prec} := \bigcup_{c \prec c'} ATT_{c'}$ represents the set of all owned and inherited attribute symbols of a class c,
– for each association $assoc \in ASSOC$ with $associates(assoc) = (c_1, c_2)$, $\sigma_{ASSOC}(assoc) \subset \sigma_{CLASS}^{\prec}(c_1) \times \sigma_{CLASS}^{\prec}(c_2)$ is a finite set of links connecting objects where $\sigma_{CLASS}^{\prec}(c) := \bigcup_{c' \prec c} \sigma_{CLASS}(c')$ is the set of all objects with type or super type c. Furthermore, $\sigma_{ASSOC}(assoc)$ must meet both multiplicity specifications for *assoc*:
$\forall i, j \in \{1, 2\}, i \neq j, \forall l = (o_1, o_2) \in \sigma_{ASSOC}(assoc) :$
$|\{l' = (o'_1, o'_2) | l' \in \sigma_{ASSOC}(assoc) \wedge o_i = o'_i\}| \in M_j$
with $multiplicities(assoc) = (M_1, M_2)$.

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 $\Sigma_M = (T_M, \leq_M, \Omega_M)$ where

– T_M is a set of types consisting of
  • all basic types (S in $DSIG$),
  • all object types (for each $c \in CLASS$ there is an object type $t_c \in T_M$),
  • all enumeration types $E \in ENUM$,
  • 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)*. [12]
– $\leq_M$ is partial order on T_M representing a type hierarchy over T_M, where

[11] Since an association end with both lower bound and upper bound set to 0 does not really make sense.
[12] T_M reflects the type hierarchy in the left part of Figure 2.

    - $Integer \leq_M Real$,
    - $t_c \leq_M t_{c'}$ if $c \prec c'$, and
    - $t \leq_M OclAny$ for all $t \in \hat{T}$ with $\hat{T}$ representing the set of all basic, enumeration, and object types in T_M.
- and Ω_M is a set of operations on T_M consisting of
    - an exhaustive set of predefined operations on primitive data types such as comparison operations on Integer, implication on Boolean, etc.,
    - operations $allInstances_{t_c}$ for obtaining all objects of type t_c,
    - operations $a : t_c \rightarrow t$ to access type attributes,
    - operations $c' : t_c \rightarrow t_{c'}$ with $assoc \in ASSOC$, $associates(assoc) = (c, c')$, $multiplicities(assoc) = (M_c, M_{c'})$, and $M_{c'} \subseteq \{0, 1\}$ to access single-valued navigable association ends of a given type t_c,
    - operations $c' : t_c \rightarrow Set(t_{c'})$ with $assoc \in ASSOC$, $associates(assoc) = (c, c')$, $multiplicities(assoc) = (M_c, M_{c'})$, and $M_{c'} \nsubseteq \{0, 1\}$ to access multi-valued navigable association ends of a given type t_c,
    - operations on sets ($isEmpty$, $notEmpty$, $includes$, $includesAll$, $excludes$, $excludesAll$, $including$, $excluding$, $size$, $union$, $-$, $intersection$, and $symmetricDifference$),
    - the constructor $mkSet_t$ for creating a set with elements of type t,
    - and operations equality ($=$) and non-equality ($\neq$) for all types $t \in T_M$.

Definition 4 (Semantics of a Data Signature). The *semantics of a data signature* $\Sigma_M = (T_M, \leq_M, \Omega_M)$ over an object model M is a structure $I(\Sigma_M) = (I(T_M), I(\leq_M), I(\Omega_M))$ where

- $I(T_M)$ assigns each $t \in T_M$ an interpretation $I(t)$, e.g.,
    - $I(Real) = \mathbb{R}$,
    - $I(t_c) = \sigma^{\prec}_{CLASS}(c)$,
    - $I(E) = literals(E)$,
    - $I(Set(t)) = F(I(t))$ where $F(I(t))$ is the set of all finite subsets of $I(t)$,
    - and $I(OclAny) = \bigcup_{t \in \hat{T}} I(t)$.
- $I(\leq_M)$ implies for all types $t, t' \in T_M$ that $I(t) \subset I(t')$ if $t \leq_M t'$,
- and $I(\Omega_M)$ assigns each operation $\omega : t_1 \times \cdots \times t_n \rightarrow t \in \Omega_M$ a total function $I(\omega) = I(t_1) \times \cdots \times I(t_n) \rightarrow I(t)$, e.g.,
    - $I(\texttt{42}) = 42$,
    - $I(+_{Integer})(i, j) = i + j$ for integers i and j,
    - $I(allInstances_{t_c}) = \sigma_{CLASS}(c)$,
    - $I(att : t_c \rightarrow t) = \sigma_{ATT}(att)(\underline{c})$ with $\underline{c} \in \sigma_{CLASS}(c)$,
    - $I(c' : t_c \rightarrow t_{c'}) = \underline{c'}$ with $(\underline{c}, \underline{c'}) \in \sigma_{ASSOC}(assoc)$,
    - $I(c' : t_c \rightarrow Set(t_{c'})) = \{\underline{c'} \mid (\underline{c}, \underline{c'}) \in \sigma_{ASSOC}(assoc)\}$,
    - $I(notEmpty(S)) = (S \neq \varnothing)$,
    - $I(mkSet_t(v_1, \ldots, v_n)) = \{v_1, \ldots, v_n\}$ with values $v_i \in I(t)$ for $1 \leq i \leq n$,
    - and $I(=_t)(v_1, v_2) = (v_1 = v_2)$ with values $v_1, v_2 \in I(t)$.

For specifying expressions for Essential OCL we use a data signature over an object model M as defined above ($\Sigma_M = (T_M, \leq_M, \Omega_M)$), a family of variable sets indexed by types $t \in T_M$ ($Var = \{Var_t\}_{t \in T_M}$), and a set of environments $Env = \{\tau \mid \tau = (\sigma, \beta)\}$ with system states σ and variable assignments $\beta : Var_t \to I(t)$ that map variable names to values.

Definition 5 (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 $t \in T_M$. Let $Env = \{\tau \mid \tau = (\sigma, \beta)\}$ be a set of environments with system states σ and variable assignments $\beta : Var_t \to I(t)$ which map variable names to values. The family of *Essential OCL expressions* over Σ_M is given by $Expr = \{Expr_t\}_{t \in T_M}$ representing sets of expressions. The *semantics of an Essential OCL expression* $e \in Expr_t$ is a function $I[\![e]\!] : Env \to I(t)$. Both, syntax and semantics, are defined inductively as follows.

– `VariableExpressions`: $v \in Expr_t$ for each variable $v \in Var_t$.[13] Moreover, $I[\![v]\!](\tau) = \beta(v)$ for each $\tau = (\sigma, \beta) \in Env$.
– `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 $e_i \in Expr_{t_i} (1 \leq i \leq n)$. [14] Moreover, $I[\![\omega(e_1, \cdots, e_n)]\!](\tau) = I(\omega)(\tau)(I[\![e_1]\!](\tau), \cdots, I[\![e_n]\!](\tau))$ for each $\tau \in Env$. Tables 1 to 3 give an overview on the syntax and semantics of concrete operation expressions in Essential OCL [15].
– `IfExpressions`: If $e_1, e_2, e_3 \in Expr_{Boolean}$ then $e :=$ **if** e_1 **then** e_2 **else** e_3 $\in Expr_{Boolean}$. [16] Moreover,

$$I[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!](\tau) = \begin{cases} I[\![e_2]\!](\tau) & \text{if } I[\![e_1]\!](\tau) = true \\ I[\![e_3]\!](\tau) & \text{otherwise} \end{cases}$$

for each $\tau \in Env$. [17]
– `TypeExpressions`: If $e \in Expr_t$ and $t, t' \in T_M$ then
  • $e.oclIsTypeOf(t') \in Expr_{Boolean}$,
  • $e.oclIsKindOf(t') \in Expr_{Boolean}$, and
  • $e.oclAsType(t') \in Expr_{t'}$. [18]

  Moreover,
  • $I[\![e.oclIsTypeOf(t')]\!](\tau) = true$ if $I[\![e]\!](\tau) \in I(t') - \bigcup_{t'' \leq_M t'} I(t'')$,
  • $I[\![e.oclIsKindOf(t')]\!](\tau) = true$ if $I[\![e]\!](\tau) \in I(t')$, and

[13] This means, that a `VariableExpression` refers to a variable, being either a context variable or an iterator variable (see Figure 2).
[14] Operations in Ω_M include: predefined operations on data types (`OperationCallExp`), class attribute operations, navigable association end operations (both `PropertyCallExp`), and constants (`LiteralExp`), see Figure 2.
[15] For primitive types we present selected operations only.
[16] Refers to an `IfExpression` in Figure 2.
[17] Alternatively, we can define the semantics of a conditional expression by using the equivalent logical expression: $I[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!](\tau) = ((I[\![e_1]\!](\tau) \wedge I[\![e_2]\!]) \vee (\neg I[\![e_1]\!] \wedge I[\![e_3]\!]))$.
[18] Refers to a `TypeExpression` in Figure 2.

- $I\llbracket e.oclAsType(t')\rrbracket(\tau) = I\llbracket e\rrbracket(\tau)$ for $t' \leq_M t$ and $I\llbracket e\rrbracket(\tau) \in I(t')$

  for each $\tau \in Env$.

– `IteratorExpressions`: If $s \in Expr_{Set(t)}, v \in Var_t$, $b \in Expr_{Boolean}$, $e_1 \in Expr_{t'}$, and $e_2 \in Exp_{Set(t')}$ then
  - $s \rightarrow exists(v \mid b) \in Expr_{Boolean}$,
  - $s \rightarrow forAll(v \mid b) \in Expr_{Boolean}$,
  - $s \rightarrow select(v \mid b) \in Expr_{Set(t)}$,
  - $s \rightarrow reject(v \mid b) \in Expr_{Set(t)}$,
  - $s \rightarrow collect(v \mid e_1) \in Expr_{Set(t')}$, and
  - $s \rightarrow collect(v \mid e_2) \in Expr_{Set(t')}$. [19] [20] [21]

  Moreover,
  - $I\llbracket s \rightarrow exists(v|b)\rrbracket(\tau) = \begin{cases} false & \text{if } I\llbracket s\rrbracket(\tau) = \varnothing \\ \bigvee_{1\leq i\leq n} I\llbracket b\rrbracket(\sigma, \beta\{v/x_i\}) & \text{if } I\llbracket s\rrbracket(\tau) = \{x_1, \ldots, x_n\} \end{cases}$,
  - $I\llbracket s \rightarrow forAll(v|b)\rrbracket(\tau) = \begin{cases} true & \text{if } I\llbracket s\rrbracket(\tau) = \varnothing \\ \bigwedge_{1\leq i\leq n} I\llbracket b\rrbracket(\sigma, \beta\{v/x_i\}) & \text{if } I\llbracket s\rrbracket(\tau) = \{x_1, \ldots, x_n\} \end{cases}$,
  - $I\llbracket s \rightarrow select(v|b)\rrbracket(\tau) = \{x \mid x \in I\llbracket s\rrbracket(\tau) \wedge I\llbracket b\rrbracket(\sigma, \beta\{v/x\}) = true\}$,
  - $I\llbracket s \rightarrow reject(v|b)\rrbracket(\tau) = \{x \mid x \in I\llbracket s\rrbracket(\tau) \wedge I\llbracket b\rrbracket(\sigma, \beta\{v/x\}) = false\}$,
  - $I\llbracket s \rightarrow collect(v|e_1)\rrbracket(\tau) = \{I\llbracket e_1\rrbracket(\sigma, \beta\{v/x\}) \mid x \in I\llbracket s\rrbracket(\tau)\}$, and
  - $I\llbracket s \rightarrow collect(v|e_2)\rrbracket(\tau) = \bigcup_{x \in I\llbracket s\rrbracket(\tau)} I\llbracket e_2\rrbracket(\sigma, \beta\{v/x\})$,

  for each $\tau \in Env$, where $\beta\{v/x\}$ denotes the substitution of all occurrences of v in β by x.[22]

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 6 (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. The following properties hold for Essential OCL invariants:

---

[19] Refers to an `IteratorExpression` in Figure 2.

[20] Please note, that this formal definition of collect results in sets of values instead of bags possibly yielding duplicate values. This means, that the translation approach presented in this paper thus restricts the expressiveness of collect. However, in many circumstances, not the number of duplicate values is crucial, but the collection of distinct values [6].

[21] Although these expressions operate on sets, they do not represent set operations. Therefore, they are not listed in Tables 2 and 3.

[22] Note, that in [12] and [1] the semantics of iterator expressions is defined in a more common but slightly different way. However, the definition presented here is quite obvious. Nevertheless, the equivalence of both definitions has to be shown.

1. An invariant `context v:C inv: expr` is equivalent to `C.allInstances -> forAll(v|expr)`. As a consequence, the semantics of an invariant is equal to the semantics of the equivalent Essential OCL expression.
2. Navigation shortcuts to collections are not contained in other navigation expressions, e.g., `somePetriNet.place.preArc -> notEmpty()` is replaced by `somePetriNet.place -> collect(p:Place|p.preArc) -> notEmpty()`.
3. Iterator expressions are completed, i.e., the iterator variable is explicitly declared. Moreover, a variable declaration is always complete, i.e., it consists of a variable name and a type name.

	Operation $\omega \in \Omega_M$	**Syntax** $e \in Expr$	**Semantics** $I\llbracket e \rrbracket(\tau)$ with $\tau = (\sigma, \beta) \in Env$
All Types	$=: t \times t \to Boolean$	e_1 `=` $e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_t$	$I\llbracket e_1 \rrbracket(\tau) = I\llbracket e_2 \rrbracket(\tau)$
	$\neq: t \times t \to Boolean$	e_1 `<>` $e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_t$	$I\llbracket e_1 \rrbracket(\tau) \neq I\llbracket e_2 \rrbracket(\tau)$
Primitive Types	$+ : Int \times Int \to Int$	e_1 `+` $e_2 \in Expr_{Integer}$ with $e_1, e_2 \in Expr_{Integer}$	$I\llbracket e_1 \rrbracket(\tau) + I\llbracket e_2 \rrbracket(\tau)$
	$\leq: Real \times Real \to Boolean$	$e_1 \leq e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_{Real}$	$I\llbracket e_1 \rrbracket(\tau) \leq I\llbracket e_2 \rrbracket(\tau)$
	$and : Boolean \times Boolean \to Boolean$	e_1 `and` $e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_{Boolean}$	$I\llbracket e_1 \rrbracket(\tau) \wedge I\llbracket e_2 \rrbracket(\tau)$
	$\omega :\to String$	`'foo'` $\in Expr_{String}$	$'foo'$
Object Types	$allInstances :\to t_c$	$t_c.allInstances() \in Expr_{Set(t_c)}$	$\sigma_{CLASS}(c)$
	$att : t_c \to t_d$	$e_c.att \in Expr_{t_d}$ with $e_c \in Expr_{t_c}$	$\sigma_{ATT}(att)(I\llbracket e_c \rrbracket(\tau))$
	$c' : t_c \to t_{c'}$ with $associates(assoc) = (c, c')$	$e_c.c' \in Expr_{t_{c'}}$ with $e_c \in Expr_{t_c}$	$\underline{c'}$ with $(I\llbracket e_c \rrbracket(\tau), \underline{c'}) \in \sigma_{ASSOC}(assoc)$
	$c' : t_c \to Set(t_{c'})$ with $associates(assoc) = (c, c')$	$e_c.c' \in Expr_{Set(t_{c'})}$ with $e_c \in Expr_{t_c}$	$\{\underline{c'} \mid (I\llbracket e_c \rrbracket(\tau), \underline{c'}) \in \sigma_{ASSOC}(assoc)\}$

Table 1. Syntax and semantics of operation expressions in Essential OCL

Set Operation $\omega \in \Omega_M$	**Syntax** $e \in Expr$	**Semantics** $I\,[\![e]\!]\,(\tau)$ with $\tau = (\sigma, \beta) \in Env$
$size : Set(t) \rightarrow Integer$	$s \rightarrow size() \in Expr_{Integer}$ with $s \in Expr_{Set(t)}$	$\mid I\,[\![s]\!]\,(\tau) \mid$
$isEmpty : Set(t) \rightarrow Boolean$	$s \rightarrow isEmpty() \in Expr_{Boolean}$ with $s \in Expr_{Set(t)}$	$I\,[\![s]\!]\,(\tau) = \varnothing$
$notEmpty : Set(t) \rightarrow Boolean$	$s \rightarrow notEmpty() \in Expr_{Boolean}$ with $s \in Expr_{Set(t)}$	$I\,[\![s]\!]\,(\tau) \neq \varnothing$
$includes : Set(t) \times t \rightarrow Boolean$	$s \rightarrow includes(e) \in Expr_{Boolean}$ with $s \in Expr_{Set(t)}$ and $e \in Expr_t$	$I\,[\![e]\!]\,(\tau) \in I\,[\![s]\!]\,(\tau)$
$excludes : Set(t) \times t \rightarrow Boolean$	$s \rightarrow excludes(e) \in Expr_{Boolean}$ with $s \in Expr_{Set(t)}$ and $e \in Expr_t$	$I\,[\![e]\!]\,(\tau) \notin I\,[\![s]\!]\,(\tau)$
$includesAll : Set(t) \times Set(t) \rightarrow Boolean$	$s \rightarrow includesAll(s') \in Expr_{Boolean}$ with $s, s' \in Expr_{Set(t)}$	$I\,[\![s']\!]\,(\tau) \subseteq I\,[\![s]\!]\,(\tau)$
$excludesAll : Set(t) \times Set(t) \rightarrow Boolean$	$s \rightarrow excludesAll(s') \in Expr_{Boolean}$ with $s, s' \in Expr_{Set(t)}$	$I\,[\![s]\!]\,(\tau) \cap I\,[\![s']\!]\,(\tau) = \varnothing$

Table 2. Syntax and semantics of set operation expressions in Essential OCL (Part 1)

Set Operation $\omega \in \Omega_M$	**Syntax** $e \in Expr$	**Semantics** $I \llbracket e \rrbracket (\tau)$ with $\tau = (\sigma, \beta) \in Env$
$mkSet_t : t \times \cdots \times t \to Set(t)$	$mkSet_t(e_1, \ldots, e_n) \in Expr_{Set(t)}$ with $e_i \in Expr_t$	$\{I \llbracket e_1 \rrbracket (\tau), \ldots, I \llbracket e_n \rrbracket (\tau)\}$
$union : Set(t) \times Set(t) \to Set(t)$	$s \to union(s') \in Expr_{Set(t)}$ with $s, s' \in Expr_{Set(t)}$	$I \llbracket s \rrbracket (\tau) \cup I \llbracket s' \rrbracket (\tau)$
$intersection : Set(t) \times Set(t)$ $\to Set(t)$	$s \to intersection(s') \in Expr_{Set(t)}$ with $s, s' \in Expr_{Set(t)}$	$I \llbracket s \rrbracket (\tau) \cap I \llbracket s' \rrbracket (\tau)$
$- : Set(t) \times Set(t) \to Set(t)$	$s - s' \in Expr_{Set(t)}$ with $s, s' \in Expr_{Set(t)}$	$I \llbracket s \rrbracket (\tau) - I \llbracket s' \rrbracket (\tau)$
$symmetricDifference :$ $Set(t) \times Set(t) \to Set(t)$	$s \to symmetricDifference(s') \in Expr_{Set(t)}$ with $s, s' \in Expr_{Set(t)}$	$(I \llbracket s \rrbracket (\tau) \cup I \llbracket s' \rrbracket (\tau)) -$ $(I \llbracket s \rrbracket (\tau) \cap I \llbracket s' \rrbracket (\tau))$
$including : Set(t) \times t \to Set(t)$	$s \to including(e) \in Expr_{Set(t)}$ with $s \in Expr_{Set(t)}$ and $e \in Expr_t$	$I \llbracket s \rrbracket (\tau) \cup \{I \llbracket e \rrbracket (\tau)\}$
$excluding : Set(t) \times t \to Set(t)$	$s \to excluding(e) \in Expr_{Set(t)}$ with $s \in Expr_{Set(t)}$ and $e \in Expr_t$	$I \llbracket s \rrbracket (\tau) - \{I \llbracket e \rrbracket (\tau)\}$

Table 3. Syntax and semantics of set operation expressions in Essential OCL (Part 2)

3 Nested Graph Constraints

In the following, we recall the formal definition of typed, attributed graphs with node type inheritance as presented in [15]. They form the basis to define typed attributed nested graph constraints.

3.1 Graphs

Attributed graphs as defined here allow to attribute nodes only while the original version [15] supports also the attribution of edges.

Definition 7 (A-graphs). 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: $src_G \colon G_E \to G_V, tgt_G \colon G_E \to G_V$ for graph edges and $src_A \colon G_A \to G_V, tgt_A \colon G_A \to G_D$ for node attribute edges. Given two A-graphs G^1 and G^2, an *A-graph morphism* $f \colon G^1 \to G^2$ is a tuple of functions $f_V \colon G^1_V \to G^2_V, f_D \colon G^1_D \to G^2_D, f_E \colon G^1_E \to G^2_E$ and $f_A \colon G^1_A \to G^2_A$ such that f commutes with all source and target functions, e.g. $f_V \circ src^1_G = src^2_G \circ f_E$. An A-graph morphism f is *injective* if the functions f_V, f_D, f_E, and f_A are injective. An injective morphism $f \colon G \to H$ is an *inclusion* if $n_G(x) \subseteq n_H(f(x))$ for all items $x \in G$.

$$
\begin{array}{ccccccc}
G^1_E & \xrightarrow{src^1_G(tgt^1_G)} & G^1_V & \xleftarrow{src^1_A} & G^1_A & \xrightarrow{tgt^1_A} & G^1_D \\
f_E \downarrow & = & f_V \downarrow & = & f_A \downarrow & = & \downarrow f_D \\
G^2_E & \xrightarrow[src^2_G(tgt^2_G)]{} & G^2_V & \xleftarrow[src^2_A]{} & G^2_A & \xrightarrow[tgt^2_A]{} & G^2_D
\end{array}
$$

We assume that the reader is familiar with the basics of algebraic specification. In [15], Appendix B, a short introduction to algebraic signatures and algebras, including term algebras, quotient term algebras, and final algebras is given. For a deeper introduction see e.g. [16,17]. The definition of attributed graphs generalizes largely the one in [18] by allowing variables and a set of formulas that constrain the possible values of these variables. The definition is closely related to symbolic graphs [19].

Definition 8 (Attributed graphs). Let $DSIG = (S, OP)$ be a data signature, $X = \{X_s\}_{s \in S}$ a family of variables, and $T_{DSIG}(X)$ the term algebra w.r.t. $DSIG$ and X. An *attributed graph* over $DSIG$ and X 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[23] with free variables in X. A set $\{F_1, \ldots, F_n\}$ of formulas can

[23] $DSIG$-formulas are meant to be $DSIG$-terms of sort BOOL. One may consider e.g. a set of literals.

be regarded as a single formula $F_1 \wedge \ldots \wedge F_n$. An attributed graph $AG = (G, D, \emptyset)$ with an empty set of formulas is *basic* and is shortly denoted by $AG = (G, D)$.

Given two attributed graphs AG^1 and AG^2, an *attributed graph morphism* $f\colon AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ of an A-graph morphism $f_G\colon G^1 \to G^2$ and a $DSIG$-homomorphism $f_D\colon D^1 \to D^2$ such that (1) commutes for all $s \in S$, $f_{G,G_D} = \sum_{s\in S} f_{D,s}$, and $\Phi^2 \Rightarrow f(\Phi^1)$ where $f(\Phi^1)$ is the set of formulas obtained when replacing in Φ^1 every variable x in G^1 by $f(x)$. An attributed graph morphism f is *injective* (*an inclusion*) if f_G and f_D are injective (inclusions).

$$\begin{array}{ccc} G^1_D & \longleftarrow & D^1_s \\ {\scriptstyle f_{G,G_D}}\downarrow & (1) & \downarrow{\scriptstyle f_{D,s}} \\ G^2_D & \longleftarrow & D^2_s \end{array}$$

Remark 2. We are interested in the case where D^1_s is a $DSIG$-term algebra and D^2_s is a $DSIG$-algebra (without variables). In this case the $DSIG$-homomorphism assigns values to variables and terms.

Attributed graphs in the sense of [18] correspond to basic attributed graphs. The results for basic attributed graphs can be generalized to arbitrary attributed graphs: attributed graphs and morphisms form the category **AGraphs**. The category has pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization in the sense of [18].

Fact 1 (properties of attributed graphs).

1. Attributed graphs and attributed morphisms form the category **AGraphs**.
2. The category **AGraphs** has pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization.
3. Pushouts are unique up to isomorphism. More precisely, if (H, Φ_H) and $(H', \Phi_{H'})$ are both pushout objects of the same morphisms $K \to R$ and $K \to D$, Then H and H' are isomorphism and Φ_H and $\Phi_{H'}$ are equivalent.
4. For every direct transformation $G \Rightarrow H$ (see Definition 18) via an injective morphism g in basic **AGraphs** and every set of formulas Φ_G, there is some Φ_H such that $(G, \Phi_G) \Rightarrow (H, \Phi_H)$ is a direct transformation in **AGraphs**.

Proof. The proof follows more or less from [18].

1. Straightforward.
2. Let $r\colon K \to R$ and $d\colon K \to D$ be attributed morphisms on basic attributed graphs and Φ_K, Φ_R, Φ_D be the corresponding sets of formulas. By [18], there is a basic attributed graph H and basic attributed morphisms $r'\colon R \to H$ and $h\colon D \to H$ such that (1) is a pushout. Let Φ_H be equivalent to $r'(\Phi_D) \cup h(\Phi_R)$. Then $\Phi_H \Rightarrow r'(\Phi_D)$ and $\Phi_H \Rightarrow h(\Phi_R)$, i.e. r' and h are attributed morphisms. $\mathcal{E}'$-$\mathcal{M}$ pair factorization is straightforward.
3. Let $l\colon K \to L$ and $g\colon L \to G$ be injective attributed morphisms on basic attributed graphs and Φ_K, Φ_L, Φ_G be the corresponding sets of formulas. If

D is a pushout complement of $K \to L \to G$ with morphisms $d\colon K \to D$ and inclusion $l'\colon D \to G$, define Φ_D be equivalent to $(\Phi_G - g(\Phi_L - l(\Phi_K))$. By definition of Φ_D, inclusion l', and $g \circ l = l' \circ d$, we have $\Phi_G \Rightarrow \Phi_G - g(\Phi_L - l(\Phi_K)) \equiv \Phi_D \equiv l'(\Phi_D)$ and $\Phi_D \equiv \Phi_G - g(\Phi_L - l(\Phi_K)) \equiv \Phi_G - g(\Phi_L) + gl(\Phi_K) \Rightarrow gl(\Phi_K)) \equiv l'd(\Phi_K) \Rightarrow d(\Phi_K)$, i.e., l' and d are attributed morphisms. Then statement 3 follows with the help of statement 2.

4. Straightforward.

□

Typed attributed graphs and morphisms form a category that has pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

Fact 2 ([8]). ATGI-graphs and morphisms form the category $\mathbf{AGraphs}_{\mathrm{ATGI}}$ with pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization in the sense of [15].

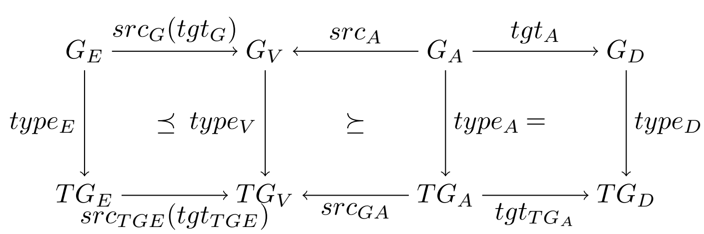
In [8], also typed attributed graphs typed over attributed type graphs with inheritance [20] are considered.

Definition 9 (Typed attributed graph over ATGI**).** An *attributed type graph with inheritance* $\mathrm{ATGI} = (TG, Z, I)$ consists of an A-graph, a final $DSIG$-algebra Z, and a simple [24] inheritance graph I with $I_V = TG_V$. For each node $v \in I_V$, the *inheritance clan* is defined by $clan_I(v) = \{v' \in I_V \mid \exists\ path\ v' \overset{*}{\to} v\ in\ I\}$[25]. If I is discrete[26], ATGI is an *attributed type graph.*

A typed attributed graph $(AG, type)$ over ATGI, short ATGI-*graph*, consists of an attributed graph $AG = (G, D, \Phi)$ and a *clan morphism type* $: AG \to \mathrm{ATGI}$.

A *clan morphism type* consists of typing functions $type_V\colon G_V \to TG_V$, $type_D\colon G_D \to TG_D$ for nodes, $type_E\colon G_E \to TG_E$, $type_A\colon G_A \to TG_A$ for edges, and the unique final $DSIG$-homomorphism $type_{DSIG}\colon D \to Z$ such that:

- $type_V \circ src_{GE} \preceq clan_I \circ src_{TGE} \circ type_E$[27]
- $type_V \circ tgt_{GE} \preceq clan_I \circ tgt_{TGE} \circ type_E$
- $type_V \circ src_{GA} \preceq clan_I \circ src_{TGA} \circ type_A$
- $type_D \circ tgt_{GA} = tgt_{TGA} \circ type_A$
- $type_{DSIG,s} = type_{D|D_s}$ for all $s \in S$.



[24] A graph is *simple* if it has neither multiple edges nor loops.

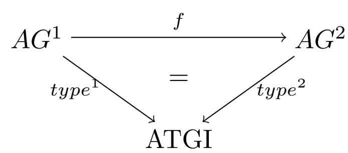
[25] $v' \overset{*}{\to} v$ in I stands for a directed path in I from v' to v of length ≥ 0.

[26] A graph is *discrete* if the edge set is empty.

[27] For functions $f\colon A \to B, g\colon A \to clan_I(B)$, $f \preceq g$ means $f(x) \in clan_I(g(x))$ for all $x \in A$ where $clan_I(B) = \{clan(v) \mid v \in B\}$.

A clan morphism *type* is *injective* (*an inclusion*) if $type_V$, $type_E$, and $type_{DSIG}$ are injective (inclusions).

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 $type^2 \circ f = type^1$.



Fact 3 (properties of typed attributed graphs).

1. ATGI-graphs and ATGI-morphisms form the category $\mathbf{AGraphs}_{\mathrm{ATGI}}$.
2. The category has pushouts.
3. For every pushout complement D of $K \to L \to G$ in basic **AGraphs**, there is a pushout complement $(D, type_D)$ of $(K, type_K) \to (L, type_L) \to (G, type_G)$ in $\mathbf{AGraphs}_{\mathrm{ATGI}}$.
4. For every direct transformation $G \Rightarrow H$ in **AGraphs** and every typing function $type_G$, there is a some $type_H$ such that $(G, type_G) \Rightarrow (H, type_H)$ a direct transformation in $\mathbf{AGraphs}_{\mathrm{ATGI}}$.

Proof. The first statement is straightforward. The other statements follow directly from [18], Lemma 13.13. □

3.2 Nested Graph Constraints

Graph conditions [21,22] 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[28], closely related to attributed graph constraints [19] and E-conditions [23]. Graph conditions are implemented e.g. in the systems AGG, GROOVE, and GrGen.

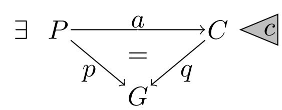
Definition 10 (Nested graph conditions). A *(nested) graph condition* on typed attributed graphs, short *condition*, over a graph P is of the form *true* or $\exists(a, c)$ where $a\colon P \to C$ is an injective morphism and c is a condition over C. Boolean formulas over conditions over P yield conditions 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 $(c \wedge \neg c') \vee (\neg c \wedge c')$.

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

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 11 (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.



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.

Lax conditions are built from *true* and arbitrary connections $\exists(C, c)$ between a graph C and a lax condition c. Lax conditions may be built from conditions as follows: Without loss of generality, conditions are based on inclusions. For each inclusion in a condition, the domain is not represented whenever it can unambiguously inferred, e.g. $\forall(C_1, \exists(C_2, c)) := \forall(\emptyset \to C_1, \exists(C_1 \to C_2, c))$. Inclusions are given by the names of nodes (and edges), e.g. $\exists(\boxed{\text{u}}, \exists(\boxed{\text{v}}, \exists(\boxed{\text{u}} \xrightarrow{\text{role}} \boxed{\text{v}})))$. Nodes of graph are decorated by a set of names, e.g. $n_G(v) = \{u, v\}$, written as $u = v$, where the index G refers to the graph in consideration.

Definition 12 (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 $\exists(C, \mathit{true})$.

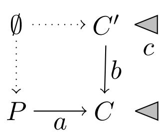
Example 2. $\exists(\boxed{\text{u}}, \exists(\boxed{\text{v}}, \exists(\boxed{\text{u}} \xrightarrow{\text{role}} \boxed{\text{v}})))$ is a lax condition, meaning that there exists a node and a node and an edge of type `role` between them.

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 $n(v)$) and its type T inside. In the case of $n(v) = \{u, v\}$, 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.

In the following, the graphs in consideration are equipped by an injective *name function* n_G assigning a set of names to each item such that and each item is in its name set and different items have disjoint name sets, i.e. $x \in n_G(x)$ and $x \neq y$ implies $n_G(x) \cap n_G(y) = \emptyset$ for all items x, y in G[29]. Moreover, the definition of an inclusion is extended to these graphs as follows: An injective morphism $f\colon G \to H$ is an *inclusion* if $n_G(v) \subseteq n_H(f_V(v))$ for all nodes $v \in \mathrm{V}_G$ and $f_E(e) = e$ for all $e \in E_G$.

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[30]). For a graph P and a lax condition d, $\mathrm{Complete}(P, d)$ denotes the condition over P, inductively defined as follows:



$\mathrm{Complete}(P, \mathit{true}) = \mathit{true}$.
$\mathrm{Complete}(P, \exists(C', c)) = \bigvee_{(a,b)\in\mathcal{F}} \exists(P \stackrel{a}{\to} C, \mathrm{Complete}(C, c))$
where $\mathcal{F} = \{(a, b) \mid (a, b)$ jointly surjective, a, b inclusions.$\}$.[31]
$\mathrm{Complete}(P, \neg c) = \neg\mathrm{Complete}(P, c)$.
$\mathrm{Complete}(P, \wedge_{i\in J} c_i) = \wedge_{i\in J}\mathrm{Complete}(P, c_i)$.

Example 3. The completion of the lax condition $\exists(\boxed{u}, \exists(\boxed{v}, \exists(\boxed{u}\xrightarrow{\text{role}}\boxed{v})))$ over the empty graph $\emptyset$ yields the condition $\exists(\emptyset \to \boxed{u}, \exists(\boxed{u} \to \boxed{u}\,\boxed{v}, \exists(\boxed{u}\,\boxed{v} \to \boxed{u}\xrightarrow{\text{role}}\boxed{v})))$, meaning that there exist two nodes together with a connecting edge of type `role`.

In more detail:

$$
\begin{aligned}
&\mathrm{Complete}(\emptyset, \exists(\boxed{u}, \exists(\boxed{v}, \exists(\boxed{u}\xrightarrow{\text{role}}\boxed{v}))))\\
\equiv\ &\exists(\emptyset \to \boxed{u}, \mathrm{Complete}(\boxed{u}, \exists(\boxed{v}, \exists(\boxed{u}\xrightarrow{\text{role}}\boxed{v}))))\\
\equiv\ &\exists(\emptyset \to \boxed{u}, \exists(\boxed{u} \to \boxed{u}\,\boxed{v}, \mathrm{Complete}(\boxed{u}\,\boxed{v}, \exists(\boxed{u}\xrightarrow{\text{role}}\boxed{v})))\\
&\qquad\quad \vee \exists(\boxed{u} \to \boxed{u{=}v}, \mathrm{Complete}(\boxed{u{=}v}, \exists(\boxed{u}\xrightarrow{\text{role}}\boxed{v}))))\\
\equiv\ &\exists(\emptyset \to \boxed{u}, \exists(\boxed{u} \to \boxed{u}\,\boxed{v}, \exists(\boxed{u}\,\boxed{v} \to \boxed{u}\xrightarrow{\text{role}}\boxed{v}))\\
&\qquad\quad \vee \exists(\boxed{u} \to \boxed{u{=}v}, \mathrm{false}))\\
\equiv\ &\exists(\emptyset \to \boxed{u}, \exists(\boxed{u} \to \boxed{u}\,\boxed{v}, \exists(\boxed{u}\,\boxed{v} \to \boxed{u}\xrightarrow{\text{role}}\boxed{v}))).
\end{aligned}
$$

Definition 13 (Semantic of lax conditions). *Satisfiability* of a lax condition is defined by the satisfiability of the corresponding condition: For an injective

[29] If we don't want to distinguish between nodes and edges we use the notation *item* and $x \in G$ means $x \in G_V$ or $x \in G_E$.

[30] The Complete and the Shift construction in [24] look very similar. While Shift is based on injective morphisms, Complete is restricted on inclusions. Complete is based on empty morphisms and completes lax conditions $\exists(C, c)$ with empty morphism $\emptyset \to C$ with respect to an empty morphism $b\colon\ \to P'$. Instead of the empty morphisms, we write the codomain of the morphisms.

[31] 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 $b(z) = x$.

morphism $p\colon P \to G$ and a lax condition c, $p \models c$ iff $p \models \text{Complete}(P, c)$. Two lax conditions c and c' are *equivalent*, denoted $c \equiv c'$, if, the corresponding conditions are equivalent.

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

Example 4. The lax condition $\exists(\boxed{\text{u}}, \exists(\boxed{\text{v}}, \exists(\boxed{\text{u}} \xrightarrow{\text{role}} \boxed{\text{v}})))$ of Example 3 is equivalent to the lax conditions $\exists(\boxed{\text{u}}\,\boxed{\text{v}}, \exists(\boxed{\text{u}} \xrightarrow{\text{role}} \boxed{\text{v}}))$ and $\exists(\boxed{\text{u}} \xrightarrow{\text{role}} \boxed{\text{v}})$ meaning that there exist two nodes together with a connecting edge of type `role`.

Remark 3. We have the following simple equivalences and non-equivalences.

1. $\exists(\boxed{\text{v}}, \exists(\boxed{\text{v}}, c)) \equiv \exists(\boxed{\text{v}}, c)$ and $\exists(\boxed{\text{u}}, \exists(\boxed{\text{v}}, c)) \equiv \exists(\boxed{\text{u}}\,\boxed{\text{v}}, c) \vee \exists(\boxed{\text{u=v}}, c[u = v])$ where $c[u = v]$ is obtained from c by identifying $\boxed{\text{u}}$ and $\boxed{\text{v}}$ in c (a special case of Fact 4, (E1)(a) below).
2. $\exists(\boxed{\text{u}}, \exists(\boxed{\text{v}}, c)) \not\equiv \exists(\boxed{\text{u}}\,\boxed{\text{v}}, c)$. If u and v are nodes in different graphs of the lax condition without inclusion relation, then, by injective satisfiability, u and v may be mapped differently or identified. If u and v are nodes in the same graph of the lax condition, by injective satisfiability, then have to be mapped differently.

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 4 (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.[32].

(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[33].
  (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.

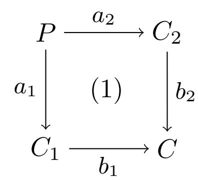
(E3) $\exists(\boxed{\text{u:T}}, \exists(C) \wedge \exists(\boxed{\text{u=v:T}})) \equiv \exists(\boxed{\text{u:T}}, \exists(C[u{=}v]))$ provided that either u or v does not exist in C and $C[u{=}v]$ is the graph obtained from C by renaming u by $u = v$.

Proof. The proof of the equivalences makes use of the Pullback-Pushout-Lemma in [26]: The pushout of the pullback of a pair $(b_1, b_2) \in \mathcal{F}$ leads to the pushout

[32] For constructions of category theory such as pushouts and pullbacks see e.g. [25,15].

[33] 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.

$C_1 \oplus_P C_2$ of C_1 and C_2 along the pullback P. In the following, $\mathcal{P}$ denotes the set of pairs (a_1, a_2) induced by the pairs $(b_1, b_2) \in \mathcal{F}$.

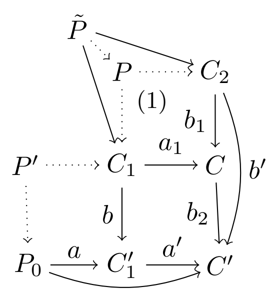


Let $p : P_0 \to G$.

(E1) (a) follows with the help of the definition of Complete:

$$\begin{aligned}
&\exists(C_1, \exists(C_2))\\
\equiv\ & \text{Complete}(P_0, \exists(C_1, \exists(C_2)))\\
\equiv\ & \textstyle\bigvee_{(a,b)\in\mathcal{F}} \exists(a, \text{Complete}(C_1', \exists(C_2, \mathit{true})))\\
\equiv\ & \textstyle\bigvee_{(a,b)\in\mathcal{F}} \exists(a, \bigvee_{(a',b')\in\mathcal{F}'} \exists(a', \text{Complete}(C', \mathit{true})))\\
\equiv\ & \textstyle\bigvee_{(a,b)\in\mathcal{F}} \exists(a, \bigvee_{(a',b')\in\mathcal{F}'} \exists(a', \mathit{true}))\\
\equiv\ & \textstyle\bigvee_{(a,b)\in\mathcal{F}} \bigvee_{(a',b')\in\mathcal{F}'} \exists(a' \circ a)\\
\equiv\ & \textstyle\bigvee_{(a,b)\in\mathcal{F}} \text{Complete}(C_1', \bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2))\\
\equiv\ & \textstyle\text{Complete}(P_0, \bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2))\\
\equiv\ & \textstyle\bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2).
\end{aligned}$$

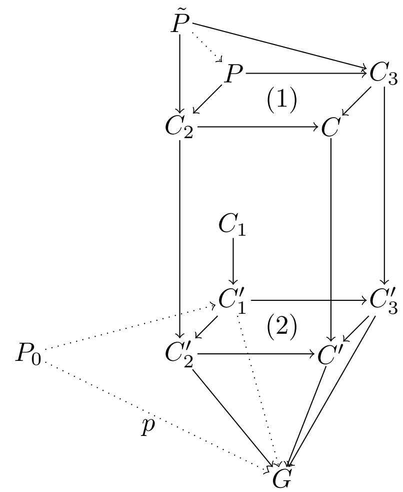
where $\mathcal{F} = \{(a, b)\}$, $\mathcal{F}'$ is the set of pairs $a' : C_1 \to C$, and $b' : C_2 \to C$ such that (a', b') is jointly surjective and a', b' are inclusions, P is the pullback of (a', b'), and C is the pushout of C_1 and C_2 along P. $\tilde{P}$ is the common part of C_1 and C_2, i.e. every pair of injective and jointly surj. (a_1, b_1) such that (1) extended to $\tilde{P}$ commutes, is a pair of inclusions. Given the morphisms (a', b'), some C exists due to $\mathcal{E}$-$\mathcal{M}$ pair factorization.



(b) If C_1 and C_2 are clan-disjoint, then $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2) \equiv \exists(C_1{+}C_2)$ because $\mathcal{F}$ consists of the pair $C_1 \to C_1{+}C_2 \leftarrow C_2$, $\mathcal{P}$ of the pair $C_1 \leftarrow \emptyset \to C_2$ and $C_1 \oplus_\emptyset C_2 = C_1{+}C_2$.

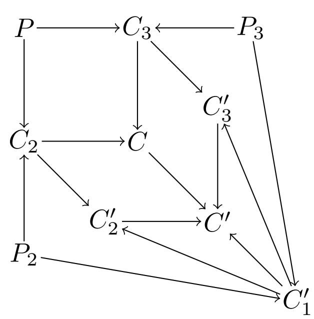
(c) If $C_1 \subseteq C_2$, then C_1 is the pullback of C_1 and C_2 and C_2 is the pushout of C_1 and C_2 along C_1. If $C_2 \subseteq C_1$, then C_2 is the pullback of C_1 and C_2 and C_1 is the pushout of C_1 and C_2 along C_2. Thus, $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2) \equiv \exists(C_2)$ if $C_1 \subseteq C_2$ and $\equiv \exists(C_1)$ if $C_2 \subseteq C_1$.

(E2) follow from the definition of Complete and $\models$. We show both directions separately. For "$\Rightarrow$" consider the commutative diagram below.



Assume $p \models \exists(C_1, \exists(C_2) \wedge \exists(C_3))$. By the definition of Complete, some C'_1, C'_2 and C'_3 exist. Let $\tilde{P}$ be the common part of (C_2, C_3), i.e. in every co-span $C_2 \rightarrow C \leftarrow C_3$ of inj. & jointly surj. morphisms such that (1) extended by $\tilde{P}$ commutes, the morphisms are inclusions. Because all node names that are common in C_2 and C_3 are also contained in C_1, C'_1 is the common part of C'_2 and C'_3. By $\mathcal{E}$-$\mathcal{M}$ pair factorization (consider (1)), some C' exists with $C' \rightarrow G$ injective. By $\mathcal{E}$-$\mathcal{M}$ pair factorization again (consider (2) extended by $\tilde{P}$), some C exists with $C \rightarrow C'$ an inclusion. By definition of Complete, $p \models \exists(C_1, \bigvee_{P \in \mathcal{P}} \exists(C_2 \oplus_P C_3))$.

For the proof's other direction consider the commutative diagram



By definition of Complete, some $P \in \mathcal{P}$, C, C' and C'_1 with $C' \rightarrow G$ exist. Let P_2 and P_3 be the common part of C'_1 and C_2, C_3 respectively. By $\mathcal{E}$-$\mathcal{M}$ pair factorization, C'_2 and C'_3 also exist and with the definiton of $\models$, $p \models \exists(C_1, \exists(C_2) \wedge \exists(C_3))$.

In the case of clan-disjointness of C_1 and C_2, $\exists(C_1) \wedge \exists(C_2) \equiv \exists(\emptyset, \exists(C_1 \wedge \exists(C_2)) \equiv \exists(\emptyset, \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2)) \equiv \exists(\emptyset, \exists(C_1 + C_2)) \equiv \exists(C_1 + C_2)$ because $\mathcal{F}$

consists of the pair $C_1 \rightarrow C_1{+}C_2 \leftarrow C_2$, $\mathcal{P}$ of the pair $C_1 \leftarrow \emptyset \rightarrow C_2$, and $C_1 \oplus_{\emptyset} C_2 = C_1{+}C_2$.

(E3) is a special case of (E2)(a) since $C[u{=}v]^{34} = C \oplus_P \boxed{\text{u=v}}$.

□

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 [15]. 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 shown.

4.1 Type and state correspondences

To translate Essential OCL invariants to nested graph constraints, we have to relate object models to attributed type graphs with inheritance.

Definition 14 (Type Correspondence). Let $DSIG = (S, OP)$ be a data signature with $S = \{Integer, Real, Boolean, String\}$. Let $M = (CLASS, ENUM, ATT, ASSOC, associates, r_{src}, r_{tgt}, multiplicities, \prec)$ be an object model over $DSIG$. We say that M *corresponds to an attributed type graph with inheritance* ATGI $= ((TG, Z), Inh)$ with

- type graph $TG = (TG_V, TG_D, TG_E, TG_A, src_G, tgt_G, src_A, tgt_A)$,
- final $DSIG'$-Algebra Z for $DSIG' = (S', OP')$ with $S' = S \cup ENUM$ and $OP' = OP \cup \{=_{ENUM}, \neq_{ENUM}\}$,
- and inheritance relation Inh,

if there is a *correspondence relation* $corr_{type} = (corr_{CLASS}, corr_{ATT}, corr_{ASSOC})$ with bijective mappings

- $corr_{CLASS} : CLASS \rightarrow TG_V$ such that $\forall c_1, c_2 \in CLASS :$ $c_1 \prec c_2 \iff (corr_{CLASS}(c_1), corr_{CLASS}(c_2)) \in Inh$,
- $corr_{ATT} : ATT \rightarrow TG_A$ with $src_A(corr_{ATT}(att)) = corr_{CLASS}(c)$ for $c \in CLASS$ and $tgt_A(corr_{ATT}(att)) = x$ if $att : c \rightarrow s \in ATT_c$ and $\{x\} = Z_s$ with $s \in S'$,
- $corr_{ASSOC} : ASSOC \rightarrow TG_E$ with $src_G \circ corr_{ASSOC} = corr_{CLASS} \circ pr_1$ and $tgt_G \circ corr_{ASSOC} = corr_{CLASS} \circ pr_2$ with $a \in ASSOC$, $associates(a) = \langle c_1, c_2 \rangle$, $pr_1(a) = c_1$, $pr_2(a) = c_2$, and $c_1, c_2 \in CLASS$.

[34] $C[u = v]$ is the graph C with the nodes named u and v identified.

To show the correctness of our translation, we also need to establish a correspondence relation between system states and typed attributed graphs.

Definition 15 (State Correspondence). Let $DSIG = (S, OP)$ be a data signature with $S = \{Integer, Real, Boolean, String\}$. Let $M = (CLASS, ENUM, ATT, ASSOC, associates, r_{src}, r_{tgt}, multiplicities, \prec)$ be an object model over $DSIG$. Let ATGI $= ((TG, Z), Inh)$ be an attributed type graph with inheritance and with type correspondence $corr_{type}(M) =$ ATGI. We assume that $I(s) = D_s$ for all sorts $s \in S' = S \cup ENUM$.

Given a system state $\sigma(M) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$, it *corresponds to an attributed graph* $AG = (G, D)$ with $G = (G_V, G_D, G_E, G_A, src_G, tgt_G, src_A, tgt_A)$ typed over ATGI by clan morphism *type* if there is a *state correspondence relation* $corr_{state} = (c_{CLASS}, c_{ATT}, c_{ASSOC}) : States(M) \to Graph_{\text{ATGI}}$ defined by the following bijective mappings

- $c_{CLASS} : \sigma_{CLASS} \to G_V$ with $type_{G_V}(c_{CLASS}(o)) = corr_{CLASS}(c)$ with $o \in \sigma_{CLASS}(c)$ and $c \in CLASS$,
- $c_{ATT} : \sigma_{ATT} \to G_A$ with $src_A(c_{ATT}(a)) = c_{CLASS}(o)$ and $tgt_A(c_{ATT}(a)) = d$ as well as $type_{G_A}(c_{ATT}(\sigma_{ATT}(att))) = corr_{ATT}(att)$ and $a \in \sigma_{ATT}(att)$ if $att : c \to s \in ATT_c^{\prec}$, $\sigma_{ATT}(att) : \sigma_{CLASS}(c) \to D_s$, $o \in \sigma_{CLASS}(c)$, $c \in CLASS$ and $\sigma_{ATT}(att)(o) = d$,
- $c_{ASSOC} : \sigma_{ASSOC} \to G_E$ with $src_G \circ c_{ASSOC} = c_{CLASS} \circ pr_1$ and $tgt_G \circ c_{ASSOC} = c_{CLASS} \circ pr_2$ with $l = (o_1, o_2) \in \sigma_{ASSOC}(assoc)$, $pr_1(l) = o_1$, and $pr_2(l) = o_2$. Furthermore, $type_{G_E} \circ c_{ASSOC}(\sigma_{ASSOC}) = corr_{ASSOC}(ASSOC)$.

Figure 3 illustrates the concepts of both correspondences.

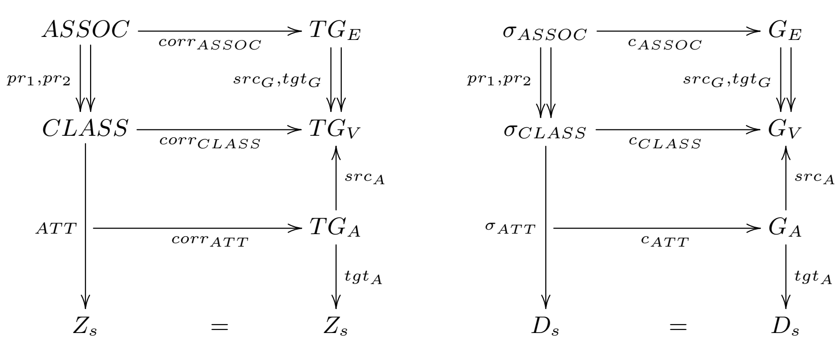


Fig. 3. Type and system state correspondences

4.2 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 .

- 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: $tr_S(\texttt{a->union(b)}, \boxed{\text{v:T}}) := tr_S(\texttt{a}, \boxed{\text{v:T}}) \vee tr_S(\texttt{b}, \boxed{\text{v:T}})$.

As an introducing example let's have a look at OCL expressions of the form `a->exists(v:T | b)` as part of an invariant. We start at the outermost part, that is `exists(v:T | b)`. This is translated in a first step to $\exists(\boxed{\text{v:T}}, tr_E(\texttt{b}))$, where tr_E denotes the translation of a Boolean expression and depends solely on `b`. Now we have to formalize that $\boxed{\text{v:T}}$ comes from the set described by `a`. This is done by giving a predicate $tr_S(\texttt{a}, \boxed{\text{v:T}})$ that describes the set precisely. Because we need the predicate over $\boxed{\text{v:T}}$, we pass $\boxed{\text{v:T}}$ as a parameter to tr_S. So the translation of the whole expression `a->exists(v:T | b)` becomes $\exists(\boxed{\text{v:T}}, tr_E(\texttt{b}) \wedge \mathrm{tr}_S(\texttt{a}, \boxed{\text{v:T}}))$, because $\boxed{\text{v:T}}$ has to fulfill both $tr_E(\texttt{b})$ and $\mathrm{tr}_S(\boxed{\text{v:T}}, \texttt{a})$. However, in our final translation process we join the two steps presented here. To motivate the translation of set expressions, let's assume that `a` in the example is `self.preArc->union(self.postArc)` with `self` of type `Transition` and `v` of type `Arc`. We have $tr_S(\texttt{self.preArc->union(self.postArc)}, \boxed{\text{v:Arc}})$, describing the union of the `preArc` and `postArc` sets. Then $\boxed{\text{v:Tr}}$ is contained in the entire set iff it is connected to `self` via a `preArc` or `postArc` relation. We derive $tr_S(\texttt{self.preArc->union(self.postArc)}, \boxed{\text{v:Arc}}) \equiv \exists(\boxed{\text{self:Tr}} \xrightarrow{\text{preArc}} \boxed{\text{v:Arc}}) \vee \exists(\boxed{\text{self:Tr}} \xrightarrow{\text{postArc}} \boxed{\text{v:Arc}})$. The translation of the overall expression `self.preArc->union(self.postArc)->exists(v:Arc | b)` is $\exists(\boxed{\text{v:Arc}}, tr_E(\texttt{b}) \wedge (\exists(\boxed{\text{self:Tr}} \xrightarrow{\text{preArc}} \boxed{\text{v:Arc}}) \vee \exists(\boxed{\text{self:Tr}} \xrightarrow{\text{postArc}} \boxed{\text{v:Arc}})))$. In general we denote four functions that each translate one type of OCL expression: We use tr_I for the translation of invariants and tr_E, tr_N and tr_S for Boolean expressions, navigation to a single object and expressions yielding a set, respectively. Since we can treat a single object as a set containing one element, the translations of navigation into a set (tr_S) and to a single node (tr_N) are technically the same. However, we distinguish them formally.

- We can express `expr1->exists(v:T | expr2)` as "there exist objects v of type T, such that v is contained in 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 contained in the set described by `expr1` *then* v also satisfies `expr2`”.

- We describe the sets via their characteristic properties, e.g. for `union`, $v \in A \cup B$ iff $v \in A \vee v \in B$. For `T.allInstances()`, the characteristic function is true for all nodes which are of type T. The idea for `select` (point 13 in the definition) is to restrict the set of nodes described by `expr1` by requiring that each node v′ satisfying `expr1` also satisfies `expr2`. The construction for `reject` is analogous. The translation tr_S(`expr1->collect(v:T | expr2)`, $\boxed{\text{v':T'}}$) (point 14) is a condition over v′ that is true iff there is a node v such that (a) v is contained in the set described by `expr1` (i.e. v satisfies tr_S(`expr1`, $\boxed{\text{v:T}}$)) and (b) the relation between v and v′ given by `expr2` is satisfied. This is described by tr_S(`expr2`, $\boxed{\text{v':T'}}$).
- For navigation (point 12), (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. The translations in (b) and (c) are identical, since single nodes are treated as single-element sets.

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 of Essential OCL constraints to nested graph constraints* consists of several parts: Invariants are translated by translation function tr_I. Any OCL expression that yields a Boolean value as result is translated by tr_E. For expressions yielding single objects, we use tr_N, and for expressions yielding collections (i.e., sets) of objects, we use tr_S. The latter two translations take a single node as their second parameter; this node represents the object (or set of objects) yielded by the expression.

Definition 16 (Constraint translation). Let $DSIG = (S, OP)$ be a data signature with $S = \{Integer, Real, Boolean, String\}$. Let $M = (CLASS, ENUM, ATT, ASSOC, associates, r_{src}, r_{tgt}, multiplicities, \prec)$ be an object model over $DSIG$ with ATGI $= corr_{type}(M)$ being the corresponding attributed type graph with inheritance. Let t : $Expr \rightarrow T$ be a typing function which returns the type of an OCL expression. Let Invariant_M be the set of Essential OCL invariants over M and $\text{GraphCondition}_{\text{ATGI}}$ be the set of all graph constraints as defined in Definition 10. The *translation functions*

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

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, $\mathrm{op} \in \{<, >, \leq, \geq, =, <>\}$ 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. (a) $tr_E(\texttt{true}) := \mathit{true}$

   (b) $tr_E(\texttt{not expr}) := \neg tr_E(\texttt{expr})$

   (c) $tr_E(\texttt{expr1 and expr2}) := tr_E(\texttt{expr1}) \wedge tr_E(\texttt{expr2})$

   (d) $tr_E(\texttt{expr1 or expr2}) := tr_E(\texttt{expr1}) \vee tr_E(\texttt{expr2})$

   (e) $tr_E(\texttt{expr1 implies expr2}) := \neg tr_E(\texttt{expr1}) \vee tr_E(\texttt{expr2})$

   (f) $tr_E(\texttt{if cond then expr1 else expr2}) :=$
   $((tr_E(\texttt{cond}) \wedge tr_E(\texttt{expr1})) \vee (\neg tr_E(\texttt{cond}) \wedge tr_E(\texttt{expr2})))$
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. (a) $tr_E(\texttt{expr1->includesAll(expr2)}) :=$
   $\forall(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr2}, \boxed{\mathrm{v{:}T}}) \Rightarrow tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}))$

   (b) $tr_E(\texttt{expr1->excludesAll(expr2)}) :=$
   $\forall(\boxed{\mathrm{v{:}T}}, tr_S(\texttt{expr2}, \boxed{\mathrm{v{:}T}}) \Rightarrow \neg tr_S(\texttt{expr1}, \boxed{\mathrm{v{:}T}}))$

   where $\mathrm{t}(\texttt{expr1}) = \mathrm{t}(\texttt{expr2}) = \texttt{Set(T)}$.
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. $tr_E(\texttt{expr1.attr1 op expr2.attr2}) :=$
   $\exists(\boxed{\mathrm{v{:}T}}, tr_N(\texttt{expr1}, \boxed{\begin{array}{c}\mathrm{v{:}T} \\ \hline \mathrm{attr1\ op\ x}\end{array}}) \wedge tr_N(\texttt{expr2}, \boxed{\begin{array}{c}\mathrm{v{:}T} \\ \hline \mathrm{attr2 = x}\end{array}})) \vee$[35]
   $\exists(\boxed{\mathrm{v{:}T}}\,\boxed{\mathrm{v'{:}T'}}, tr_N(\texttt{expr1}, \boxed{\begin{array}{c}\mathrm{v{:}T} \\ \hline \mathrm{attr1\ op\ x}\end{array}}) \wedge tr_N(\texttt{expr2}, \boxed{\begin{array}{c}\mathrm{v'{:}t(v')} \\ \hline \mathrm{attr2 = x}\end{array}}))$
   where $\mathrm{t}(\texttt{expr1}) = \mathrm{T}$, $\mathrm{t}(\texttt{expr2}) = \mathrm{T}'$, $\mathrm{t(x)} = \mathrm{t}(\texttt{attr1}) = \mathrm{t}(\texttt{attr2})$ and x, v and v' are fresh variables.

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

10. (a) $tr_E(\texttt{expr.oclIsKindOf(T)}) := \exists(\boxed{\text{v:T'}} \hookrightarrow \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v:T'}}))$
    (b) $tr_E(\texttt{expr.oclIsTypeOf(T)}) :=$
    $\exists(\boxed{\text{v:T'}} \hookrightarrow \boxed{\text{v:T}}, \bigwedge_{T'' \in clan(T)}^{T'' \neq T} \neg\exists(\boxed{\text{v:T}} \hookrightarrow \boxed{\text{v:T''}}) \wedge tr_N(\texttt{expr}, \boxed{\text{v:T'}}))$
    where $\mathrm{T}' = \mathrm{t}(\texttt{expr})$ and $\mathrm{T} \in clan(\mathrm{T}')$.
11. $tr_N(\texttt{expr.oclAsType(T)}, \boxed{\text{v:T}}) := \exists(\boxed{\text{v:T'}} \hookrightarrow \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v:T'}}))$
    where $\mathrm{T}' = \mathrm{t}(\texttt{expr})$ and $\mathrm{T} \in clan(\mathrm{T}')$
12. (a) $tr_N(\texttt{v}, \boxed{\text{v':T}}) := \exists(\boxed{\text{v=v':T}})$ if `v` is a variable,
    (b) If `role` has a multiplicity of 1, $tr_N(\texttt{expr.role}, \boxed{\text{v:T}}) :=$
    $\exists(\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v':T'}}))$ if $\mathrm{T}' \notin clan(\mathrm{T})$ and
    $\exists(\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v':T'}})) \vee \exists(\boxed{\text{v:T}} \circlearrowleft^{\text{role}}, tr_N(\texttt{expr}, \boxed{\text{v:T}}))$ else.
    (c) If `role` has a multiplicity > 1, $tr_S(\texttt{expr.role}, \boxed{\text{v:T}}) :=$
    $\exists(\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v':T'}}))$ if $\mathrm{T}' \notin clan(\mathrm{T})$ and
    $\exists(\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v':T'}})) \vee \exists(\boxed{\text{v:T}} \circlearrowleft^{\text{role}}, tr_N(\texttt{expr}, \boxed{\text{v:T}}))$ else,
    where v' is a fresh variable and $\mathrm{t}(\texttt{expr}) = \mathrm{T}'$.
13. (a) $tr_S(\texttt{expr1->select(v:T | expr2)}, \boxed{\text{v':T}}) :=$
    $tr_S(\texttt{expr1}, \boxed{\text{v':T}}) \wedge tr_E(\texttt{expr2})\{\mathrm{v}/\mathrm{v}'\}$
    (b) $tr_S(\texttt{expr1->reject(v:T | expr2)}, \boxed{\text{v':T}}) :=$
    $tr_S(\texttt{expr1}, \boxed{\text{v':T}}) \wedge \neg tr_E(\texttt{expr2})\{\mathrm{v}/\mathrm{v}'\}$
    where $\texttt{expr2}\{\mathrm{v}/\mathrm{v}'\}$ means replacing `v` in `expr2` with $\texttt{v}'$.
14. (a) $tr_S(\texttt{expr1->collect(v:T | expr2)}, \boxed{\text{v':T'}}) :=$
    $\exists(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_S(\texttt{expr2}, \boxed{\text{v':T'}}))$ if `expr2` yields a set, and
    (b) $tr_S(\texttt{expr1->collect(v:T | expr2)}, \boxed{\text{v':T'}}) :=$
    $\exists(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_N(\texttt{expr2}, \boxed{\text{v':T'}}))$ if `expr2` yields an object.
15. (a) $tr_S(\texttt{expr1->union(expr2)}, \boxed{\text{v:T}}) := tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \vee tr_S(\texttt{expr2}, \boxed{\text{v:T}})$
    (b) $tr_S(\texttt{expr1->intersect(expr2)}, \boxed{\text{v:T}}) := tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_S(\texttt{expr2}, \boxed{\text{v:T}})$
    (c) $tr_S(\texttt{expr1 - expr2}, \boxed{\text{v:T}}) := tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge \neg tr_S(\texttt{expr2}, \boxed{\text{v:T}})$
    (d) $tr_S(\texttt{expr1->symmetricDifference(expr2)}, \boxed{\text{v:T}}) :=$
    $tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \veebar tr_S(\texttt{expr2}, \boxed{\text{v:T}})$
16. $tr_S(\texttt{T.allInstances()}, \boxed{\text{v:T}}) := \exists(\boxed{\text{v:T}})$
17. $tr_S(\texttt{Set\{expr1, ..., exprN\}}, \boxed{\text{v:T}}) :=$
    $tr_N(\texttt{expr1}, \boxed{\text{v:T}}) \vee \cdots \vee tr_N(\texttt{exprN}, \boxed{\text{v:T}})$
    where $\texttt{expr1}, \ldots, \texttt{exprN}$ are OCL expressions of type T.

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 [12], Tables 4.4 and 4.5 and page 73.

Definition 17 (further constraint translation).

1. $tr_E(\texttt{expr1->includes(expr2)}) := tr_E(\texttt{expr1->includesAll(Set\{expr2\})})$
   $tr_E(\texttt{expr1->excludes(expr2)}) := tr_E(\texttt{expr1->excludesAll(Set\{expr2\})})$
2. $tr_E(\texttt{expr1->including(expr2)}) := tr_E(\texttt{expr1->union(Set\{expr2\})})$
   $tr_E(\texttt{expr1->excluding(expr2)}) := tr_E(\texttt{expr1 - Set\{expr2\}})$

3. $tr_E($`expr1 <> expr2`$) := tr_E($`not expr1 = expr2`$)$
4. $tr_E($`expr1->isEmpty()`$) := tr_E($`not expr1->notEmpty()`$)$
5. $tr_E($`expr->size() > n`$) := tr_E($`expr->size() >= n+1`$)$
   $tr_E($`expr->size() = n`$) :=$
   $\quad tr_E($`expr->size() >= n and not expr->size() >= n+1`$)$
   $tr_E($`expr->size() <= n`$) := tr_E($`not expr->size() > n`$)$
   $tr_E($`expr->size() < n`$) := tr_E($`not expr->size() >= n`$)$
   $tr_E($`expr->size() <> n`$) := tr_E($`not expr->size() = n`$)$
6. $tr_N($`expr1->any(v|expr2)`$, \boxed{\text{v:T}}) := tr_S($`expr1->select(v|expr2)`$, \boxed{\text{v:T}})$
   $tr_E($`expr1->one(v|expr2)`$) := tr_E($`expr1->select(v|expr2)->size()=1`$)$

where `expr`, `expr1` and `expr2` are OCL expressions and `n` is an integer constant.

4.3 Examples

In the following examples, an index above the $=$ sign refers to the translation rule used; an index at the equivalence sign $\equiv$ refers to the used equivalence rule of Proposition 4.

Example 5. The name of a transition is not empty.

$$tr_I(\texttt{context Transition inv: self.name <> ''}) =^{1}$$
$$\forall(\boxed{\text{self:Tr}}, tr_E(\texttt{self.name <> ''})) =^{8}$$
$$\forall(\boxed{\text{self:Tr}}, \exists(\boxed{\text{self:Tr}}, \exists(\boxed{\begin{array}{c}\text{self:Tr}\\ \hline \text{name <> ''}\end{array}}))) =^{E1}$$
$$\forall(\boxed{\text{self:Tr}}, \exists(\boxed{\begin{array}{c}\text{self:Tr}\\ \hline \text{name <> ''}\end{array}}))$$

Example 6. There is no isolated transition.

$$tr_I(\texttt{context Transition inv: self.preArc->notEmpty() or}$$
$$\quad \texttt{self.postArc->notEmpty()}\ ) =^{1}$$
$$\forall(\boxed{\text{self:TR}}, \text{tr}_E(\texttt{self.preArc->notEmpty() or self.postArc->notEmpty()})) =^{2}$$
$$\forall(\boxed{\text{self:TR}}, \text{tr}_E(\texttt{self.preArc->notEmpty()}) \vee tr_E(\texttt{self.postArc->notEmpty()})) =^{5}$$
$$\forall(\boxed{\text{self:TR}}, \exists(\boxed{\text{v:PTArc}}, tr_S(\texttt{self.preArc}, \boxed{\text{v:PTArc}})) \vee$$
$$\quad \exists(\boxed{\text{w:TPArc}}, tr_S(\texttt{self.postArc}, \boxed{\text{w:TPArc}}))) =^{12}$$
$$\forall(\boxed{\text{self:TR}}, \exists(\boxed{\text{v:PTArc}}, \exists(\boxed{\text{self:TR}} \xrightarrow{\text{preArc}} \boxed{\text{v:PTArc}})) \vee$$
$$\quad \exists(\boxed{\text{w:TPArc}}, \exists(\boxed{\text{self:TR}} \xrightarrow{\text{postArc}} \boxed{\text{w:TPArc}}))) \equiv^{\text{E1b}}$$
$$\forall(\boxed{\text{self:TR}}, \exists(\boxed{\text{self:TR}} \xrightarrow{\text{preArc}} \boxed{\text{v:PTArc}}) \vee \exists(\boxed{\text{self:TR}} \xrightarrow{\text{postArc}} \boxed{\text{w:TPArc}}))$$

Alternatively:

$$tr_I(\texttt{context Petrinet inv: self.transition->forAll(t:Transition |}$$
$$\texttt{t.preArc->notEmpty() or t.postArc->notEmpty()))} =^{1}$$
$$\forall(\boxed{\text{self:PN}}, \text{tr}_E(\texttt{self.transition->forAll(t:Transition|}$$
$$\texttt{t.preArc->notEmpty() or t.postArc->notEmpty())})) =^{3}$$
$$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, tr_S(\texttt{self.transition}, \boxed{\text{t:Tr}}) \Rightarrow$$
$$tr_E(\texttt{t.preArc->notEmpty() or t.postArc->notEmpty()}))) =^{17}$$
$$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, \exists(\boxed{\text{u:PN}}, tr_N(\texttt{self}, \boxed{\text{u:PN}})) \wedge \exists(\boxed{\text{u:Pn}} \xrightarrow{\text{tr}} \boxed{\text{t:Tr}}) \Rightarrow$$
$$tr_E(\texttt{t.preArc->notEmpty() or t.postArc->notEmpty()}))) =^{12}$$
$$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, \exists(\boxed{\text{u:PN}}, \exists(\boxed{\text{u=self:PN}}) \wedge \exists(\boxed{\text{u:Pn}} \xrightarrow{\text{tr}} \boxed{\text{t:Tr}})) \Rightarrow$$
$$tr_E(\texttt{t.preArc->notEmpty() or t.postArc->notEmpty()})) =^{5}$$
$$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, \exists(\boxed{\text{u:PN}}, \exists(\boxed{\text{u=self:PN}}) \wedge \exists(\boxed{\text{u:Pn}} \xrightarrow{\text{tr}} \boxed{\text{t:Tr}})) \Rightarrow$$
$$\exists(\boxed{\text{v1:PTArc}}, tr_N(\texttt{t.preArc}, \boxed{\text{v1:PTArc}}))\vee$$
$$\exists(\boxed{\text{v2:TPArc}}, tr_N(\texttt{t.postArc}, \boxed{\text{v2:TPArc}})))) =^{12}$$
$$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, \exists(\boxed{\text{u:PN}}, \exists(\boxed{\text{u=self:PN}}) \wedge \exists(\boxed{\text{u:Pn}} \xrightarrow{\text{tr}} \boxed{\text{t:Tr}})) \Rightarrow$$
$$\exists(\boxed{\text{v1:PTArc}}, \exists(\boxed{\text{w1:Tr}}, tr_N(\texttt{t}, \boxed{\text{w1:Tr}}) \wedge \exists(\boxed{\text{w1:Tr}} \xrightarrow{\text{preArc}} \boxed{\text{v1:PTArc}})))\vee$$
$$\exists(\boxed{\text{v2:TPArc}}, \exists(\boxed{\text{w2:Tr}}, tr_N(\texttt{t}, \boxed{\text{w2:Tr}}) \wedge \exists(\boxed{\text{w2:Tr}} \xrightarrow{\text{postArc}} \boxed{\text{v2:TPArc}}))))) =^{12}$$
$$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, \exists(\boxed{\text{u:PN}}, \exists(\boxed{\text{u=self:PN}}) \wedge \exists(\boxed{\text{u:Pn}} \xrightarrow{\text{tr}} \boxed{\text{t:Tr}})) \Rightarrow$$
$$\exists(\boxed{\text{v1:PTArc}}, \exists(\boxed{\text{w1:Tr}}, \exists(\boxed{\text{w1=t:Tr}}) \wedge \exists(\boxed{\text{w1:Tr}} \xrightarrow{\text{preArc}} \boxed{\text{v1:PTArc}})))\vee$$
$$\exists(\boxed{\text{v2:TPArc}}, \exists(\boxed{\text{w2:Tr}}, \exists(\boxed{\text{w2=t:Tr}}) \wedge \exists(\boxed{\text{w2:Tr}} \xrightarrow{\text{postArc}} \boxed{\text{v2:TPArc}}))))) \equiv^{E3}$$
$$\forall(\boxed{\text{self:PN}} \xrightarrow{\text{tr}} \boxed{\text{t:Tr}}, \exists(\boxed{\text{t:Tr}} \xrightarrow{\text{preArc}} \boxed{\text{v1:PTArc}}) \vee \exists(\boxed{\text{t:Tr}} \xrightarrow{\text{postArc}} \boxed{\text{v2:TPArc}}))$$

Example 7. There is no isolated place.

$$tr_I(\texttt{context Place inv: self.preArc->notEmpty() or self.postArc->notEmpty()}) =^{1}$$
$$\forall(\boxed{\text{self:Pl}}, \text{tr}_E(\texttt{self.preArc->notEmpty() or self.postArc->notEmpty()}) =^{2}$$
$$\forall(\boxed{\text{self:Pl}}, \text{tr}_E(\texttt{self.preArc->notEmpty()})\vee$$
$$tr_E(\texttt{self.postArc->notEmpty()})) =^{5}$$
$$\forall(\boxed{\text{self:Pl}}, \exists(\boxed{\text{v:TPArc}}, tr_S(\texttt{self.preArc}, \boxed{\text{v:TPArc}}))\vee$$
$$\exists(\boxed{\text{w:PTArc}}, tr_S(\texttt{self.postArc}, \boxed{\text{w:PTArc}}))) =^{12}$$
$$\forall(\boxed{\text{self:Pl}}, \exists(\boxed{\text{v:TPArc}}, \exists(\boxed{\text{self:Pl}} \xrightarrow{\text{preArc}} \boxed{\text{v:TPArc}})) \vee \exists(\boxed{\text{w:PTArc}}, \exists(\boxed{\text{self:Pl}} \xrightarrow{\text{postArc}} \boxed{\text{w:PTArc}}))) \equiv^{E1b}$$
$$\forall(\boxed{\text{self:Pl}}, \exists(\boxed{\text{self:Pl}} \xrightarrow{\text{preArc}} \boxed{\text{v:TPArc}}) \vee \exists(\boxed{\text{self:Pl}} \xrightarrow{\text{postArc}} \boxed{\text{w:PTArc}}))$$

Example 8. Each two places of a Petri net have different names.

$\forall(\boxed{\text{self:PN}}, tr_E(\texttt{self.place->forAll(p1:Place|self.place->forAll(p2:Place|}$
$\quad\texttt{p1<>p2 implies p1.name<>p2.name))))) =^{2\times 3}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, tr_S(\texttt{self.place}, \boxed{\text{p1:Pl}}) \Rightarrow (tr_S(\texttt{self.place}, \boxed{\text{p2:Pl}}) \Rightarrow$
$\quad tr_E(\texttt{p1<>p2 implies p1.name<>p2.name})))) =^{2\times 12}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}) \Rightarrow (\exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p2:Pl}}) \Rightarrow$
$\quad tr_E(\texttt{p1<>p2 implies p1.name<>p2.name})))) =^{2, Def.\ 17}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}) \Rightarrow (\exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p2:Pl}}) \Rightarrow$
$\quad tr_E(\texttt{p1<>p2}) \Rightarrow tr_E(\texttt{p1.name<>p2.name})))) =^{7,9}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}) \Rightarrow (\exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p2:Pl}}) \Rightarrow$
$\quad \neg\exists(\boxed{\text{p:Pl}}, \exists(\boxed{\text{p=p1:Pl}}) \wedge \exists(\boxed{\text{p=p2:Pl}})) \Rightarrow \exists(\begin{array}{|c|}\hline \text{p1:Pl}\\\hline \text{name<>x}\\\hline\end{array}\begin{array}{|c|}\hline \text{p2:Pl}\\\hline \text{name=x}\\\hline\end{array})))) \equiv^{\text{E1a}}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}) \Rightarrow (\exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p2:Pl}}) \Rightarrow$
$\quad \neg\exists(\boxed{\text{p=p1=p2:Pl}}) \Rightarrow \exists(\begin{array}{|c|}\hline \text{p1:Pl}\\\hline \text{name<>x}\\\hline\end{array}\begin{array}{|c|}\hline \text{p2:Pl}\\\hline \text{name=x}\\\hline\end{array})))) \equiv^{A\Rightarrow B\equiv\neg A\vee B}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \neg\exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}) \vee \neg\exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p2:Pl}}) \vee$
$\quad \exists(\boxed{\text{p=p1=p2:Pl}}) \vee \exists(\begin{array}{|c|}\hline \text{p1:Pl}\\\hline \text{name<>x}\\\hline\end{array}\begin{array}{|c|}\hline \text{p2:Pl}\\\hline \text{name=x}\\\hline\end{array}))) \equiv^{\text{DeMorgan, E1c}}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \neg(\exists(\boxed{\text{self:PN}}\begin{smallmatrix}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}\\ \xrightarrow{\text{place}}\boxed{\text{p2:Pl}}\end{smallmatrix}) \wedge \neg\exists(\begin{array}{|c|}\hline \text{p1:Pl}\\\hline \text{name<>x}\\\hline\end{array}\begin{array}{|c|}\hline \text{p2:Pl}\\\hline \text{name=x}\\\hline\end{array})))) \equiv^{\text{DeMorgan}}$

$\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{p1:Pl}}, \neg\exists(\boxed{\text{self:PN}}\begin{smallmatrix}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}\\ \xrightarrow{\text{place}}\boxed{\text{p2:Pl}}\end{smallmatrix}) \vee \exists(\begin{array}{|c|}\hline \text{p1:Pl}\\\hline \text{name<>x}\\\hline\end{array}\begin{array}{|c|}\hline \text{p2:Pl}\\\hline \text{name=x}\\\hline\end{array}))) \equiv^{A\Rightarrow B\equiv\neg A\vee B}$

$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}}\begin{smallmatrix}\xrightarrow{\text{place}}\boxed{\text{p1:Pl}}\\ \xrightarrow{\text{place}}\boxed{\text{p2:Pl}}\end{smallmatrix}) \Rightarrow \exists(\begin{array}{|c|}\hline \text{p1:Pl}\\\hline \text{name<>x}\\\hline\end{array}\begin{array}{|c|}\hline \text{p2:Pl}\\\hline \text{name=x}\\\hline\end{array}))$

Example 9. There is at least one place in a Petri net having at least one token.

$tr_I(\texttt{context PetriNet inv: self.place->exists(p:Place|p.token->notEmpty())}) =^{1}$
$\forall(\boxed{\text{self:PN}}, tr_E(\texttt{self.place->exists(p:Place|p.token->notEmpty())})) =^{3}$
$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, tr_S(\texttt{self.place}, \boxed{\text{p:Pl}}) \wedge tr_E(\texttt{p.token->notEmpty()}))) =^{5}$
$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, tr_S(\texttt{self.place}, \boxed{\text{p:Pl}}) \wedge \exists(\boxed{\text{t:Tk}}, tr_S(\texttt{p.token}, \boxed{\texttt{t:Tk}})))) =^{12}$
$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{p:Pl}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p:Pl}}) \wedge \exists(\boxed{\text{t:Tk}}, \exists(\boxed{\text{p:Pl}}\xrightarrow{\text{token}}\boxed{\text{t:Tk}})))) \equiv^{E1,E2}$
$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}}\xrightarrow{\text{place}}\boxed{\text{p:Pl}}\xrightarrow{\text{token}}\boxed{\text{t:Tk}}))$

Alternatively:

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

Alternatively:

$$tr_I(\texttt{context PetriNet inv: self.place->collect(p:Place|p.token)->notEmpty()}) =^{1}$$
$$\forall(\boxed{\text{self:PN}}, tr_E(\texttt{self.place->collect(p:Place|p.token)->notEmpty()})) =^{5}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{t:Tk}}, tr_S(\texttt{self.place->collect(p:Place|p.token)}, \boxed{\text{t:Tk}}))) =^{14}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{t:Tk}}, tr_S(\texttt{self.place}, \boxed{\text{p:Pl}}) \wedge tr_S(\texttt{p.token}, \boxed{\text{t:Tk}}))) =^{12}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{t:Tk}}, \exists(\boxed{\text{self:PN}} \xrightarrow{\text{place}} \boxed{\text{p:Pl}}) \wedge \exists(\boxed{\text{p:Pl}} \xrightarrow{\text{token}} \boxed{\text{t:Tk}}))) \equiv^{E1,E2}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}} \xrightarrow{\text{place}} \boxed{\text{p:Pl}} \xrightarrow{\text{token}} \boxed{\text{t:Tk}}))$$

Alternatively [36]:

$$tr_I(\texttt{context PetriNet inv: Token.allInstances()->notEmpty()}) =^{1}$$
$$\forall(\boxed{\text{self:PN}}, tr_E(\texttt{Token.allInstances()->notEmpty()})) =^{5}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{v:Tk}}, tr_S(\texttt{Token.allInstances()}, \boxed{\text{v:Tk}}))) =^{16}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{v:Tk}}, \exists(\boxed{\text{v:Tk}}))) \equiv^{E1}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{v:Tk}}))$$

[36] The resulting nested graph constraint slightly differs from those presented above. Please see footnote 3 on page 5.

Example 10. The weight of an arc is positive.

$$tr_I(\texttt{context Arc inv: self.weight >= 1}) =^1$$
$$\forall(\boxed{\text{self:Arc}}, tr_E(\texttt{self.weight >= 1})) =^8$$
$$\forall(\boxed{\text{self:Arc}}, \exists(\boxed{\text{v:Arc}}, tr_N(\texttt{self}, \boxed{\text{v:Arc}}) \wedge \exists(\boxed{\begin{array}{c}\text{v:Arc}\\ \hline \text{weight} \geq 1\end{array}}))) =^{12}$$
$$\forall(\boxed{\text{self:Arc}}, \exists(\boxed{\text{v:Arc}}, \exists(\boxed{\text{v=self:Arc}}) \wedge \exists(\boxed{\begin{array}{c}\text{v:Arc}\\ \hline \text{weight} \geq 1\end{array}}))) \equiv^{E3}$$
$$\forall(\boxed{\text{self:Arc}}, \exists(\boxed{\begin{array}{c}\text{self:Arc}\\ \hline \text{weight} \geq 1\end{array}}))$$

Example 11. Each Petrinet has at least two places.

$$tr_I(\texttt{context Petrinet inv:self.place->size() >= 2}) =^1$$
$$\forall(\boxed{\text{self:PN}}, tr_E(\texttt{self.place->size() >= 2})) =^6$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{v1:Pl}}\,\boxed{\text{v2:Pl}}, tr_S(\texttt{self.place}, \boxed{\text{v1:Pl}}) \wedge tr_S(\texttt{self.place}, \boxed{\text{v2:Pl}}))) =^{12a,12c}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{v1:Pl}}\,\boxed{\text{v2:Pl}}, \exists(\boxed{\text{self:PN}} \xrightarrow{\text{place}} \boxed{\text{v1:Pl}}) \wedge \exists(\boxed{\text{self:PN}} \xrightarrow{\text{place}} \boxed{\text{v2:Pl}}))) \equiv^{E2}$$
$$\forall(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}} \begin{array}{l}\xrightarrow{\text{place}} \boxed{\text{v2:Pl}}\\ \xrightarrow[\text{place}]{} \boxed{\text{v1:Pl}}\end{array}))$$

Example 12. There is at least one transition that can be fired, i.e., all PTArcs targeting this transition must have a weight less or equal to the token number of their source places.

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 is done according to the items in [8, Definition 12] and uses the definition of tr', the equivalences of Fact 4, and the definition of tr. Moreover, the proof for item (11–12) makes use of an induction over the structure of Core OCL constraints.

In more detail: Items 1 and from tr' correspond to item 1 from tr, items 4–8 correspond to 2, 16 and 17 from tr' to 9 and 8 in tr, respectively. The other items are proved as follows; item numbers are taken from Definition 12 in [8].

(3) This just splits up handling of tr'_E and needs no correspondence in tr_E.

(9,10) Note that in items 9,10 from tr', `navExpr` is always of the form `v.role` and `v` and `v.role` refer to distinct nodes.

$tr'_E(\texttt{v.role->notEmpty()})$ (Def. $tr'_E 9$)
$\exists(tr'_N(\texttt{v.role})) =$ (Def. $tr'_N 18$)
$\exists(\emptyset \to \boxed{\text{v:T}} \xrightarrow{\text{role}} \boxed{\text{v':T'}}) \equiv$ (E1b)
$\exists(\emptyset \to \boxed{\text{v':T'}}, \exists(\boxed{\text{v:T}} \xrightarrow{\text{role}} \boxed{\text{v':T'}})) =$ (Def. $tr_N 12$)
$\exists(\emptyset \to \boxed{\text{v':T'}}, tr_N(\texttt{v.role}, \boxed{\text{v:T}})) =$ (Def. $tr_E 5$)
$tr_E(\texttt{v.role->notEmpty()})$
The proof for `isEmpty` is analogous.

(11,12) This is proven by induction over the structure of OCL constraints.
Induction base: $tr'_E(\texttt{true}) = true = tr_E(\texttt{true})$.
Hypothesis: For sub-constraints `expr`, $tr'_E(\texttt{expr}) = tr_E(\texttt{expr})$.
$tr'_E(\texttt{u.role->exists(v:T| expr)}) =$ (Def. $tr'_E 11$)
$\exists(tr'_N(\texttt{u.role}) \sqsupseteq \boxed{\text{v:T}}, tr'_E(\texttt{expr})) =$ (Def. $tr'_N 18$)
$\exists(\boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}} \sqsupseteq \boxed{\text{v:T}}, tr'_E(\texttt{expr})) \equiv$ ([8, Def. $\sqsupseteq$])
$\text{Shift}_{\sqsupseteq}(\boxed{\text{v:T}} \hookrightarrow \boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}},$
$\quad \exists(\boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}} \sqsupseteq \boxed{\text{v:T}}, tr'_E(\texttt{expr}))) \equiv$ ([8, $\text{Shift}_{\sqsupseteq}$])
$\exists(\boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr'_E(\texttt{expr})) \equiv$ (Ind. hyp.)
$\exists(\boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_E(\texttt{expr})) \equiv$ (E1b)
$\exists(\boxed{\text{u:T'}}\,\boxed{\text{v:T}}, \exists(\boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_E(\texttt{expr}))) \equiv$ (E2b,E1b)
$\exists(\boxed{\text{u:T'}}\,\boxed{\text{v:T}}, \exists(\boxed{\text{u:T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}) \wedge tr_E(\texttt{expr})) =$ (Def. $tr_S 12$)
$\exists(\boxed{\text{u:T}}\,\boxed{\text{v:T}}, tr_S(\texttt{u.role}, \boxed{\text{v:T}}) \wedge tr_E(\texttt{expr})) =$ (Def. $tr_E 3$)
$tr_E(\texttt{u.role->exists(v:T| expr)})$
The proof for `forall` proceeds analogously.

(13)
$tr'_E(\texttt{v = u}) =$ (Def. $tr'_E 13a$)
$\exists(\boxed{\text{v:T}}\,\boxed{\text{u:T}} \to \boxed{\text{v=u:T}}) \equiv$ ([8, Footnote 21])
$\text{Shift}(\boxed{\text{v:T}}\,\boxed{\text{u:T}} \to \boxed{\text{v=u:T}}, \exists(\boxed{\text{v:T}}\,\boxed{\text{u:T}} \to \boxed{\text{v=u:T}}))$ (Def. Shift)
$\exists(\boxed{\text{v=u:T}})$ (E1a)
$\exists(\boxed{\text{v:T}}, \exists(\boxed{\text{v:T}}, \exists(\boxed{\text{v:T}}) \wedge \exists(\boxed{\text{u:T}}, \exists(\boxed{\text{u=v:T}}))))$ (Def. $tr_N 12$)
$\exists(\boxed{\text{v:T}}, tr_N(\texttt{v}, \boxed{\text{v:T}}) \wedge tr_N(\texttt{u}, \boxed{\text{v:T}})) \equiv$ (Def. $tr_E 7$)
$tr_E(\texttt{v = u})$.
The proof for `u <> v` proceeds analogously.

(14,15) Let $T' = \mathrm{t}(\texttt{r1}) = \mathrm{t}(\texttt{r2})$ and assume $\texttt{r1} \neq \texttt{r2}$ and $\texttt{v} \neq \texttt{v.r1}, \texttt{v.r2}$[36].
$tr'_E(\texttt{v.r1 = v.r2}) =$ (Def. $tr'_E 14$)
$\exists(\boxed{\text{v:T}} \overset{\text{r1}}{\underset{\text{r2}}{\rightrightarrows}} \boxed{\text{u:T'}}) \equiv$ (E2a)
$\exists(\boxed{\text{u:T'}}, \exists(\boxed{\text{v:T}} \xrightarrow{\text{r1}} \boxed{\text{u:T'}}) \wedge \exists(\boxed{\text{v:T}} \xrightarrow{\text{r2}} \boxed{\text{u:T'}})) =$ (Def. $tr_S 12$)
$\exists(\boxed{\text{u:T}}, \exists(tr_S(\texttt{v.r1}, \boxed{\text{u:T'}})) \wedge \exists(tr_S(\texttt{v.r2}, \boxed{\text{u:T'}}))) =$ (Def. $tr_E 7$)
$tr_E(\texttt{v.r1 = v.r2})$.

(18) Since tr'_N yields graphs and tr_N yields conditions, there can be no direct correspondence between the two. However, tr'_N is used in the construction of

[36] This is a silent assumption in [8]; indeed, tr'_N does not work for models with direct loops in the metamodel.

tr'_E in cases 9 to 12. For these cases, we showed the correspondence. These are all occurrences of tr'_N, so no further proof is necessary here.

(19) Variables v of type T are translated ad-hoc by tr into nodes $\boxed{\text{v:T}}$, which corresponds directly to tr'_V. □

4.4 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.

4.5 Correctness

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 $inv \in \mathrm{Dom}(tr_I)$ and all environments $(\sigma, \beta) \in Env$,

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

Proof. We prove, by induction over the structure of Essential OCL invariants, the more general statement
(1) $I[\![\texttt{expr}]\!](\sigma, \beta) = true \Leftrightarrow p \models tr_E(expr)$,

(2) $I[\![\texttt{expr}]\!](\sigma,\beta) = v \Leftrightarrow p \oplus \mathrm{id}_v \models tr_N(expr, \boxed{\text{v:T}})$[37],

(3) $I[\![\texttt{expr}]\!](\sigma,\beta) = \{v_1,\ldots,v_n\} \Leftrightarrow \forall v \in \{v_1,\ldots,v_n\}.p \oplus \mathrm{id}_v \models tr_S(expr, \boxed{\text{v:T}})$.

Base Case. $I[\![\texttt{context C inv: true}]\!](\sigma,\beta) = \mathit{true} = \forall v \in \sigma_{Class}(C).\mathit{true} = \forall(\boxed{\text{v:C}}, \mathit{true}) = tr_I(\texttt{context C inv: true})$.

Hypothesis. For all subexpressions `expr`, objects $v, v_1, \ldots, v_n$ and morphisms $p\colon \{\boxed{\text{v:T}} \in c_{Class}(\beta(v)) | v \in \mathrm{Dom}(\beta)\} \to corr_{State}(\sigma)$, let statements (1), (2) and (3) be true.

Induction Step.

(1) See the corresponding proof in the extended version of [8]. Case `context C inv: expr` follows as a special case of the above with `var` = `self`.

(2) See the corresponding proof in the extended version of [8].

(3) Let t(`expr1`) = T.

$I[\![\texttt{expr1->exists(v:T| expr2)}]\!](\sigma,\beta)$ (Def. 5)

$\Leftrightarrow I[\![\texttt{expr1}]\!](\sigma,\beta) = \{v_1,\ldots,v_n\} \wedge \bigvee_{1\le i\le n} I[\![\texttt{expr2}]\!](\sigma,\beta\{\texttt{v}/v_i\})$ (set axioms)

$\Leftrightarrow I[\![\texttt{expr1}]\!](\sigma,\beta) = \{v_1,\ldots,v_n\} \wedge$
$\quad \exists v_i \in \{v_1,\ldots,v_n\}.I[\![\texttt{expr2}]\!](\sigma,\beta\{\texttt{v}/v_i\})$ (set axioms)

$\Leftrightarrow \exists v \in \sigma_{Class}(\mathrm{T}).I[\![\texttt{expr1}]\!](\sigma,\beta) \wedge I[\![\texttt{expr2}]\!](\sigma,\beta)$ (Ind. hyp.)

$\Leftrightarrow \exists(\boxed{\text{v:T}} \in c_{Class}(\sigma_{Class}(\mathrm{T})).p \oplus \mathrm{id}_v \models tr_S(\texttt{expr1}, \boxed{\text{v:T}})$
$\quad \wedge p \oplus \mathrm{id}_v \models tr_E(\texttt{expr2}))$ (Def. $\models$)

$\Leftrightarrow p \models \exists(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_E(\texttt{expr2}))$ (Def. 16.3)

$\Leftrightarrow p \models tr_E(\texttt{expr1->exists(v:T| expr2)})$

The proof of `forall` is analoguous.

(4) Let t(`expr1`) = t(`expr2`) = Set(T).

$I[\![\texttt{expr1->includesAll(expr2)}]\!](\sigma,\beta)$ (Def. 5)

$\Leftrightarrow I[\![\texttt{expr2}]\!](\sigma,\beta) \subseteq I[\![\texttt{expr1}]\!](\sigma,\beta)$ (set axioms)

$\Leftrightarrow \forall v \in \sigma(\mathrm{T}).v \in I[\![\texttt{expr2}]\!](\sigma,\beta)$ implies $v \in I[\![\texttt{expr1}]\!](\sigma,\beta)$ (Ind. hyp.)

$\Leftrightarrow \forall \boxed{\text{v:T}} \in c_{Class}(\sigma(\mathrm{T})).p \oplus \mathrm{id}_v \models tr_S(\texttt{expr2}, \boxed{\text{v:T}})$
$\quad$ implies $p \oplus \mathrm{id}_v \models tr_S(\texttt{expr1}, \boxed{\text{v:T}})$ (Def. $\models$)

$\Leftrightarrow p \models \forall(\boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v:T}})$ implies $tr_S(\texttt{expr1}, \boxed{\text{v:T}}))$ (Def. 16.4)

$\Leftrightarrow p \models tr_E(\texttt{expr1->includesAll(expr2)})$

The proof of `excludesall` is analogous.

(5)

$I[\![\texttt{expr->notEmpty()}]\!](\sigma,\beta)$ (Def. 5)

$\Leftrightarrow I[\![\texttt{expr}]\!](\sigma,\beta) \neq \emptyset$ (set axioms)

$\Leftrightarrow \exists v \in \sigma_{Class}(\mathrm{T}).v \in I[\![\texttt{expr}]\!](\sigma,\beta)$ (Ind. hyp.)

$\Leftrightarrow \exists \boxed{\text{v:T}} \in c_{Class}(\sigma_{Class}(\mathrm{T})).p \oplus \mathrm{id}_v \models tr_S(\texttt{expr}, \boxed{\text{v:T}})$ (Def. $\models$)

$\Leftrightarrow p \models \exists(\boxed{\text{v:T}}, tr_S(\texttt{expr}, \boxed{\text{v:T}}))$ (Def. 16.5)

$\Leftrightarrow p \models tr_E(\texttt{expr->notEmpty()})$

[37] For morphisms $p\colon P \to G$, let function composition $p \oplus \mathrm{id}_v$ be the morphism $p'\colon P \oplus \boxed{\text{v:T}} \to G$, with $p'(v) = p(v)$ if $v \in \mathrm{Dom}(p)$ and $p'(v) = v$ otherwise. Note that $P = \emptyset$ for constraints.

(6)
$I[\![\texttt{expr->size() >= } n]\!](\sigma,\beta)$ (Def. 5)
$\Leftrightarrow |\{v \mid I[\![\texttt{expr}]\!](\sigma,\beta)\}| >= n$ (set axioms)
$\Leftrightarrow \exists v_1,\ldots,v_n \in \sigma(\mathrm{T}).\ \bigwedge_{i,j=1,i\neq j}^{n}(v_i \neq v_j)$
$\quad \wedge \bigwedge_{i=1}^{n}(v_i \in I[\![\texttt{expr}]\!](\sigma,\beta))$ (Ind. hyp.)
$\Leftrightarrow \exists \boxed{\mathrm{v}_1\mathrm{:T}} \ldots \boxed{\mathrm{v}_n\mathrm{:T}} \in c_{Class}(\sigma(\mathrm{T})).\ \bigwedge_{i=1}^{n} p \oplus \mathrm{id}_{v_i} \models tr_S(\texttt{expr}, \boxed{\mathrm{v}_i\mathrm{:T}})$ (Def. $\models$)
$\Leftrightarrow p \models \exists(\boxed{\mathrm{v}_1\mathrm{:T}} \ldots \boxed{\mathrm{v}_n\mathrm{:T}}, \bigwedge_{i=1}^{n} tr_S(\texttt{expr}, \boxed{\mathrm{v}_i\mathrm{:T}}))$ (Def. 16.6)
$\Leftrightarrow p \models tr_E(\texttt{expr->size() >= } n)$

(7a) For $\mathrm{t}(\texttt{expr1}) = \mathrm{t}(\texttt{expr2}) = \mathrm{T}$ for some class T,
$I[\![\texttt{expr1 = expr2}]\!](\sigma,\beta)$ (Def. 5)
$\Leftrightarrow I[\![\texttt{expr1}]\!](\sigma,\beta) = I[\![\texttt{expr2}]\!](\sigma,\beta)$ (use variable)
$\Leftrightarrow \exists v \in \sigma_{Class}(\mathrm{T}).v = I[\![\texttt{expr1}]\!](\sigma,\beta) \wedge v = I[\![\texttt{expr2}]\!](\sigma,\beta)$ (Ind. hyp.)
$\Leftrightarrow \exists \boxed{\mathrm{v:T}} \in c_{Class}(\sigma_{Class}(\mathrm{T})).p \oplus \mathrm{id}_v \models tr_N(\texttt{expr1}, \boxed{\mathrm{v:T}})$
$\quad \wedge p \oplus \mathrm{id}_v \models \mathrm{tr}_N(\texttt{expr2}, \boxed{\mathrm{v:T}})$ (Def. $\models$)
$\Leftrightarrow p \models \exists(\boxed{\mathrm{v:T}}, tr_N(\texttt{expr1}, \boxed{\mathrm{v:T}}) \wedge \mathrm{tr}_N(\texttt{expr2}, \boxed{\mathrm{v:T}}))$ (Def. 16.7a)
$\Leftrightarrow p \models tr_E(\texttt{expr1 = expr2})$

(7b) For $\mathrm{t}(\texttt{expr1}) = \mathrm{t}(\texttt{expr2}) = \mathrm{Set}(\mathrm{T})$ for some class T,
$I[\![\texttt{expr1 = expr2}]\!](\sigma,\beta)$ (Def. 5)
$\Leftrightarrow I[\![\texttt{expr1}]\!](\sigma,\beta) = I[\![\texttt{expr2}]\!](\sigma,\beta)$ (set axioms)
$\Leftrightarrow \forall v \in \sigma_{Class}(\mathrm{T}).v \in I[\![\texttt{expr1}]\!](\sigma,\beta) \text{ iff } v \in I[\![\texttt{expr2}]\!](\sigma,\beta)$ (Ind. hyp.)
$\Leftrightarrow \forall \boxed{\mathrm{v:T}} \in c_{Class}(\sigma(T)).p \oplus \mathrm{id}_v \models tr_S(\texttt{expr1}, \boxed{\mathrm{v:T}}) \text{ iff}$
$\quad p \oplus \mathrm{id}_v \models tr_S(\texttt{expr2}, \boxed{\mathrm{v:T}})$ (Def. $\models$)
$\Leftrightarrow p \models \forall(\boxed{\mathrm{v:T}}, tr_S(\texttt{expr1}, \boxed{\mathrm{v:T}}) \text{ iff } tr_S(\texttt{expr2}, \boxed{\mathrm{v:T}}))$ (Def. 16.7b)
$\Leftrightarrow p \models tr_E(\texttt{expr1 = expr2})$

(8) See the corresponding proof in the extended version of [8].

(9) Let $\mathrm{T} = \mathrm{t}(\texttt{expr1}), \mathrm{T}' = \mathrm{t}(\texttt{expr2})$ and $att(\texttt{v}:\mathrm{T}, \texttt{att}) = \sigma_{Att}(\texttt{att})(I[\![\texttt{v}:\mathrm{T}]\!](\sigma, \beta))$,
Let $p_v = p \oplus \mathrm{id}_v$ and $p_{v'} = p \oplus \mathrm{id}_{v'}$.

$I[\![\texttt{ex1.a1 op ex2.a2}]\!](\sigma, \beta)$ (Def. 5)

$\Leftrightarrow att(\texttt{ex1}, \texttt{a1})\ \texttt{op}\ att(\texttt{ex2}, \texttt{a2})$ (Def. 5)

$\Leftrightarrow \exists v, v'.v = I[\![\texttt{ex1}]\!](\sigma, \beta) \wedge v' = I[\![\texttt{ex2}]\!](\sigma, \beta)$
$\quad \wedge att(\texttt{v}, \texttt{a1})\ \texttt{op}\ att(\texttt{v'}, \texttt{a2})$ (Ind. hyp.)

$\Leftrightarrow \exists(\boxed{\text{v:T}}, \boxed{\text{v':T'}}.p_v \models (tr_N(\texttt{ex1}, \boxed{\text{v:T}}) \wedge \exists(\begin{array}{|c|}\hline \text{v:T} \\ \hline \text{a1} = \text{x} \\ \hline \end{array})))$
$\quad \wedge p_{v'} \models tr_N(\texttt{ex2}, \boxed{\text{v':T'}}) \wedge \exists(\begin{array}{|c|}\hline \text{v':T'} \\ \hline \text{a2} = \text{x} \\ \hline \end{array})$ (Equiv. 2)

$\Leftrightarrow \exists(\boxed{\text{v=v':T}}.p_v \models tr_N(\texttt{ex1}, \begin{array}{|c|}\hline \text{v=v':T} \\ \hline \text{a1 op x} \\ \hline \end{array}) \wedge p_v \models tr_N(\texttt{ex2}, \begin{array}{|c|}\hline \text{v=v':T} \\ \hline \text{a2} = \text{x} \\ \hline \end{array}))$

$\vee \exists(\boxed{\text{v:T}}\ \boxed{\text{v':T'}}, p_v \models tr_N(\texttt{ex1}, \begin{array}{|c|}\hline \text{v:T} \\ \hline \text{a1 op x} \\ \hline \end{array}) \wedge p_{v'} \models tr_N(\texttt{ex2}, \begin{array}{|c|}\hline \text{v':T'} \\ \hline \text{a2} = \text{x} \\ \hline \end{array}))$ (Def. $\models$)

$\Leftrightarrow p \models \exists(\boxed{\text{v:T}}, tr_N(\texttt{ex1}, \begin{array}{|c|}\hline \text{v:T} \\ \hline \text{a1 op x} \\ \hline \end{array}) \wedge tr_N(\texttt{ex2}, \begin{array}{|c|}\hline \text{v:T} \\ \hline \text{a2} = \text{x} \\ \hline \end{array}))$
$\quad \vee \exists(\boxed{\text{v:T}}\ \boxed{\text{v':T'}}, tr_N(\texttt{ex1}, \begin{array}{|c|}\hline \text{v:T} \\ \hline \text{a1 op x} \\ \hline \end{array}) \wedge tr_N(\texttt{ex2}, \begin{array}{|c|}\hline \text{v':T'} \\ \hline \text{a2} = \text{x} \\ \hline \end{array}))$ (Def. 16.9)

$\Leftrightarrow p \models tr_E(\texttt{ex1.a1 op ex2.a2})$

(10a) Let $\mathrm{t}(\texttt{expr}) = \mathrm{T}'$ and $\mathrm{T} \in clan(\mathrm{T}')$.

$I[\![\texttt{expr.oclIsTypeOf(T)}]\!](\sigma, \beta)$ (Def. 5)

$\Leftrightarrow I[\![\texttt{expr}]\!](\sigma, \beta) \in (I(\mathrm{T}) - \bigcup_{\mathrm{T}'' \leq_M \mathrm{T}}^{\mathrm{T}'' \neq \mathrm{T}} I(\mathrm{T}''))$ (set axioms)

$\Leftrightarrow \exists v = I[\![\texttt{expr}]\!](\sigma, \beta).v \in I(\mathrm{T}) \wedge \bigwedge_{\mathrm{T}'' \leq_M \mathrm{T}}^{\mathrm{T}'' \neq \mathrm{T}} .v \notin I(\mathrm{T}'')$ (Def. 3, 2)

$\Leftrightarrow \exists v = I[\![\texttt{expr}]\!](\sigma, \beta).v \in \sigma_{Class}^{\preceq}(\mathrm{T}) \wedge \bigwedge_{\mathrm{T}'' \leq_M \mathrm{T}}^{\mathrm{T}'' \neq \mathrm{T}} .v \notin \sigma_{Class}^{\preceq}(\mathrm{T}'')$ (Ind. hyp.)

$\Leftrightarrow \exists(\boxed{\text{v:T'}}.\exists(\boxed{\text{v:T'}} \to \boxed{\text{v:T}}) \wedge \bigwedge_{\mathrm{T}'' \leq_M \mathrm{T}}^{\mathrm{T}'' \neq \mathrm{T}} .\neg\exists(\boxed{\text{v:T'}} \to \boxed{\text{v:T''}}))$ (Def. $\models$)

$\Leftrightarrow p \models \exists(\boxed{\text{v:T'}} \to \boxed{\text{v:T}}, \bigwedge_{\mathrm{T}'' \in clan(\mathrm{T})}^{\mathrm{T}'' \neq \mathrm{T}} \neg\exists(\boxed{\text{v:T}} \to \boxed{\text{v:T''}})$
$\quad \wedge tr_N(\texttt{expr}, \boxed{\text{v:T'}}))$ (Def. 16.10)

$\Leftrightarrow p \models tr_E(\texttt{expr.oclIsTypeOf(T)})$

(10b) The proof is analogous to the one for `ocsIsTypeOf` (without the $\bigcup$-part):
Let $\mathrm{t}(\texttt{expr}) = \mathrm{T}'$.

$I[\![\texttt{expr.oclIsKindOf(T)}]\!](\sigma, \beta)$ (Def. 5)

$\Leftrightarrow I[\![\texttt{expr}]\!](\sigma, \beta) \in I(\mathrm{T})$ (set axioms)

$\Leftrightarrow \exists v = I[\![\texttt{expr}]\!](\sigma, \beta).v \in I(\mathrm{T})$ (Def. 3, 2)

$\Leftrightarrow \exists v = I[\![\texttt{expr}]\!](\sigma, \beta).v \in \sigma_{Class}(\mathrm{T})$ (Ind. hyp.)

$\Leftrightarrow \exists(\boxed{\text{v:T'}}, \exists(\boxed{\text{v:T'}} \to \boxed{\text{v:T}}))$ (Def. $\models$)

$\Leftrightarrow \exists(\boxed{\text{v:T'}} \to \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v:T'}}))$ (Def. 16.10)

$\Leftrightarrow tr_E(\texttt{expr.oclIsKindOf(T)})$

(11) Let $\mathrm{t}(\texttt{expr}) = \mathrm{T}'$.

$$v = I[\![\texttt{expr.oclAsType(T)}]\!](\sigma,\beta) \qquad \text{(Def. 5)}$$
$$\Leftrightarrow v = I[\![\texttt{expr}]\!](\sigma,\beta) \wedge I[\![\texttt{expr}]\!](\sigma,\beta) \in I(\mathrm{T}) \qquad \text{(Def. 3, 2)}$$
$$\Leftrightarrow v = I[\![\texttt{expr}]\!](\sigma,\beta) \wedge v \in \sigma_{Class}(\mathrm{T}) \qquad \text{(Ind. hyp.)}$$
$$\Leftrightarrow \exists \boxed{\text{v:T'}} \in c_{Class}(\sigma(\mathrm{T})).\wedge p \oplus \mathrm{id}_v \models tr_N(\texttt{expr}, \boxed{\text{v:T'}}) \wedge \exists(\boxed{\text{v:T'}} \rightarrow \boxed{\text{v:T}}) \qquad \text{(Def. } \models\text{)}$$
$$\Leftrightarrow p \models \exists(\boxed{\text{v:T'}} \rightarrow \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v:T'}})) \qquad \text{(Def 16.11)}$$
$$\Leftrightarrow p \models tr_N(\texttt{expr.oclAsType(T)}, \boxed{\text{v:T'}})$$

(12a) Let $\mathrm{t}(v) = \mathrm{t}(v') = \mathrm{T}$.

$$v' = I[\![\texttt{v}]\!](\sigma,\beta) \qquad \text{(Def. 5)}$$
$$\Leftrightarrow \exists v' \in \sigma(\mathrm{T}).\beta(v) = v' \qquad (c_{Class})$$
$$\Leftrightarrow \exists \boxed{\text{v=v':T}} \in c_{Class}(\sigma(\mathrm{T})) \qquad \text{(Def. } \models\text{)}$$
$$\Leftrightarrow p \models \exists(\boxed{\text{v=v':T}}) \qquad \text{(Def. } tr_N\text{)}$$
$$\Leftrightarrow p \models tr_N(\texttt{v}, \boxed{\text{v':T}})$$

(12b) First, assume $\mathrm{T} \notin clan(\mathrm{T}')$ and let $\mathrm{t}(\texttt{expr}) = \mathrm{T}', \mathrm{t}(\texttt{expr.role}) = \mathrm{T}$.

$$v = I[\![\texttt{expr.role}]\!](\sigma,\beta) \qquad \text{(Def. 5)}$$
$$\Leftrightarrow (I[\![\texttt{expr}]\!](\sigma,\beta), v) \in \sigma_{Assoc}(\texttt{role}) \qquad (c_{Assoc})$$
$$\Leftrightarrow \exists v' = I[\![\texttt{expr}]\!](\sigma,\beta) \wedge \mathrm{t}(v') = \mathrm{T}' \wedge \boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}} \in c_{Assoc}(\sigma_{Assoc}(\texttt{role})) \qquad \text{(Ind. hyp.)}$$
$$\Leftrightarrow \exists(\boxed{\text{v':T'}} \in c_{Class}(\sigma(\mathrm{T})).p \oplus \mathrm{id}_{v'} \models tr_N(\texttt{expr}, \boxed{\text{v':T'}}) \wedge p \oplus \mathrm{id}_{v'} \oplus \mathrm{id}_v \models \boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}) \qquad \text{(Def. } \models\text{)}$$
$$\Leftrightarrow p \models \exists(\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v':T'}})) \qquad \text{(Def. 16.12)}$$
$$\Leftrightarrow p \models tr_N(\texttt{expr.role}, \boxed{\text{v':T}})$$

Now assume $\mathrm{T} \in clan(\mathrm{T}')$ and let $\mathrm{t}(\texttt{expr}) = \mathrm{T}', \mathrm{t}(\texttt{expr.role}) = \mathrm{T}$.

$$v = I[\![\texttt{expr.role}]\!](\sigma,\beta) \qquad \text{(Def. 5)}$$
$$\Leftrightarrow (I[\![\texttt{expr}]\!](\sigma,\beta), v) \in \sigma_{Assoc}(\texttt{role}) \qquad (c_{Assoc})$$
$$\Leftrightarrow \exists v' = I[\![\texttt{expr}]\!](\sigma,\beta).\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}} \in c_{Assoc}(\sigma_{Assoc}(\texttt{role})) \vee \boxed{\text{v=v':T'}} \circlearrowleft^{\text{role}} \qquad \text{(Ind. hyp.)}$$
$$\Leftrightarrow \exists(\boxed{\text{v':T'}}.p \oplus \mathrm{id}_{v'} \models tr_N(\texttt{expr}, \boxed{\text{v':T'}}) \wedge (p \oplus \mathrm{id}_{v'} \oplus \mathrm{id}_v \models \boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}} \vee \boxed{\text{v=v':T'}} \circlearrowleft^{\text{role}})) \qquad \text{(Def. } \models\text{)}$$
$$\Leftrightarrow p \models \exists(\boxed{\text{v':T'}} \xrightarrow{\text{role}} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v':T'}})) \vee \exists(\boxed{\text{v:T}} \circlearrowleft^{\text{role}}, tr_N(\texttt{expr}, \boxed{\text{v:T}})) \qquad \text{(Def. 16.12)}$$
$$\Leftrightarrow p \models tr_N(\texttt{expr.role}, \boxed{\text{v':T}})$$

The proof of the tr_S cases is analogous to the tr_N cases.

(13) Let $\mathrm{t}(\texttt{expr1}) = \mathrm{Set}(\mathrm{T})$.

$$v \in I[\![\texttt{expr1->select(v:T| expr2)}]\!](\sigma,\beta) \qquad \text{(Def. 5)}$$
$$\Leftrightarrow v \in \{v \mid v \in I[\![\texttt{expr1}]\!](\sigma,\beta)\} \wedge I[\![\texttt{expr2}]\!](\sigma,\beta)\} \qquad \text{(set axioms)}$$
$$\Leftrightarrow \exists v \in \sigma(\mathrm{T}).v \in I[\![\texttt{expr1}]\!](\sigma,\beta) \wedge I[\![\texttt{expr2}]\!](\sigma,\beta) \qquad \text{(Ind. hyp.)}$$
$$\Leftrightarrow p \oplus \mathrm{id}_v \models tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge p \oplus \mathrm{id}_v \models tr_E(\texttt{expr2}) \qquad \text{(Def. 16.13)}$$
$$\Leftrightarrow p \models tr_S(\texttt{expr1->select(v:T| expr2)}, \boxed{\text{v:T}})$$

The proof for `reject` is analogous.

(14) Let t($\texttt{expr1}$) = Set(T).

$v \in I[\![\texttt{expr1->collect(v:T| expr2)}]\!](\sigma, \beta)$ (Def. 5)

$\Leftrightarrow v \in \{I[\![\texttt{expr2}]\!](\sigma, \beta\{\texttt{v}/v'\}) | v' \in I[\![\texttt{expr1}]\!](\sigma, \beta)\}$ (set axioms)

$\Leftrightarrow \exists v' \in I[\![\texttt{expr1}]\!](\sigma, \beta).v \in I[\![\texttt{expr}]\!](\sigma, \beta\{\texttt{v}/v'\})$ (Ind. hyp.)

$\Leftrightarrow \exists(\boxed{\text{v:T}}, \boxed{\text{v':T'}}.p \oplus \mathrm{id}_{v'} \models tr_S(\texttt{expr1}, \boxed{\text{v':T'}})$

$\quad \wedge p \oplus \mathrm{id}_v \models tr_S(\texttt{expr2}, \boxed{\text{v:T}}))$ (Def. $\models$)

$\Leftrightarrow \exists(\boxed{\text{v:T}}, p \models \exists(\boxed{\text{v':T'}}, tr_S(\texttt{expr1}, \boxed{\text{v':T'}})$

$\quad \wedge p \oplus \mathrm{id}_v \oplus \mathrm{id}_{v'} \models tr_S(\texttt{expr2}, \boxed{\text{v:T}})))$ (Def. $\models$)

$\Leftrightarrow p \models \exists(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_S(\texttt{expr2}, \boxed{\text{v':T'}}))$ (Def. 16.14)

$\Leftrightarrow p \models tr_S(\texttt{expr1->collect(v:T| expr2)}, \boxed{\text{v':T'}})$

The proof for `expr2` yielding an object is analogous.

(15) Let t($\texttt{expr1}$) = Set(T).

$v \in I[\![\texttt{expr1->union(expr2)}]\!](\sigma, \beta)$ (Def. 5)

$\Leftrightarrow v \in \{v' \mid v' \in I[\![\texttt{expr1}]\!](\sigma, \beta)\} \cup \{v' \mid v' \in I[\![\texttt{expr2}]\!](\sigma, \beta)\}$ (set axioms)

$\Leftrightarrow v \in I[\![\texttt{expr1}]\!](\sigma, \beta) \vee v \in I[\![\texttt{expr2}]\!](\sigma, \beta)$ (Ind. hyp.)

$\Leftrightarrow p \oplus \mathrm{id}_v \models tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \vee p \oplus \mathrm{id}_v \models tr_S(\texttt{expr2}, \boxed{\text{v:T}})$ (Def. $\models$)

$\Leftrightarrow p \models tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \vee tr_S(\texttt{expr2}, \boxed{\text{v:T}})$ (Def. 16.15)

$\Leftrightarrow p \models tr_S(\texttt{expr1->union(expr2)}, \boxed{\text{v:T}})$

The proofs for `intersect`, `-` and `symmetricDifference` are analogous.

(16)

$v \in I[\![\texttt{T.allInstances}]\!](\sigma, \beta) = \sigma_{Class}(T)$ (Def. 5)

$\Leftrightarrow v \in \sigma_{Class}(\mathrm{T})$ ($corr_{Class}$)

$\Leftrightarrow \mathrm{t}(v) = \mathrm{T}$ (Def. $\models$)

$\Leftrightarrow p \models \exists(\boxed{\text{v:T}})$ (Def. tr_S)

$\Leftrightarrow p \models tr_S(\mathrm{T}\texttt{.allInstances()}, \boxed{\text{v:T'}})$

(17) Let t($\texttt{expr1}$) = ... = t($\texttt{exprN}$) = T.

$v \in I[\![\texttt{Set\{expr1,...,exprN\}}]\!](\sigma, \beta)$ (Def. 5)

$\Leftrightarrow v \in \{I[\![\texttt{expr1}]\!](\sigma, \beta), \ldots, I[\![\texttt{exprN}]\!](\sigma, \beta)\}$ (set axioms)

$\Leftrightarrow v = I[\![\texttt{expr1}]\!](\sigma, \beta) \vee \cdots \vee v = I[\![\texttt{exprN}]\!](\sigma, \beta)$ (Ind. hyp.)

$\Leftrightarrow p \oplus \mathrm{id}_v \models tr_N(\texttt{expr1}, \boxed{\text{v:T}}) \vee \cdots \vee p \oplus \mathrm{id}_v \models tr_N(\texttt{exprN}, \boxed{\text{v:T}})$ (Def. $\models$)

$\Leftrightarrow p \models tr_N(\texttt{expr1}, \boxed{\text{v:T}}) \vee \cdots \vee tr_N(\texttt{exprN}, \boxed{\text{v:T}})$ (Def. tr_S)

$\Leftrightarrow p \models tr_S(\texttt{Set\{expr1, ..., exprN\}}, \boxed{\text{v:T}})$

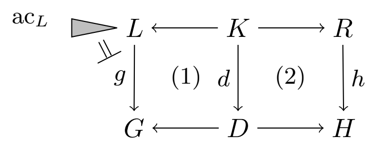
This completes the induction proof. We obtain Theorem 1 because, for OCL expression inv = `context C inv: expr` and morphism $p\colon \emptyset \to G$, $G \models tr_I(inv)$ iff $p \models \forall(\boxed{\text{self:C}}, tr_E(\texttt{expr}))$. □

5 From Essential OCL Invariants to Application Conditions

After having translated Essential OCL invariants to graph constraints, we connect this new result with the existing theory on graph constraints [9,27]. A main result shows how nested graph constraints can be translated to right, and thereafter, to left application conditions of transformation rules. In the following, we illustrate at an example how a Essential OCL invariant is translated to a left application condition.

We recall the definition of graph transformation with injective rules, left application conditions, and injective matches.

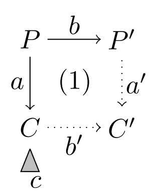
Definition 18 (rules and transformations). A *rule* $\varrho = \langle p, \mathrm{ac}_L \rangle$ consists of a *plain rule* $p = \langle L \leftarrow K \rightarrow R \rangle$ with injective morphisms $K \rightarrow L$ and $K \rightarrow R$, and an application condition ac_L over L. A *direct transformation* from a graph G to a graph H via the rule ϱ consists of two pushouts (1) and (2) as below where morphism g is injective and $g \models \mathrm{ac}_L$. We write $G \Rightarrow_{\varrho,g,h} H$ or $G \Rightarrow_{\varrho,g} H$ if there exists such a direct transformation.



The first result says that conditions can be shifted over injective morphisms.

Lemma 1 (shift of conditions over injective morphisms [24] [38]). There is a Shift′ construction such that, for each condition c over P and for each injective morphism $b\colon P \rightarrow P'$, Shift′ transforms c via b into a condition $\mathrm{Shift}'(b, c)$ over P' such that, for each injective morphism $n\colon P' \rightarrow H$, $n \circ b \models c \iff n \models \mathrm{Shift}'(b, c)$.

Construction. The Shift′ construction is inductively defined as follows:



$\mathrm{Shift}'(b, \mathit{true}_P) = \mathit{true}_{P'}$.
$\mathrm{Shift}'(b, \exists(a, c)) = \bigvee_{(a',b') \in \mathcal{F}'} \exists(a', \mathrm{Shift}'(b', c))$ where
$\mathcal{F}' = \{(a', b') \mid (a', b') \text{ jointly surjective, } a', b' \text{ inj., (1) commutes}\}$
$\mathrm{Shift}'(b, \neg c) = \neg \mathrm{Shift}'(b, c)$, $\mathrm{Shift}'(b, \wedge_{i \in J} c_i) = \wedge_{i \in J} \mathrm{Shift}'(b, c_i)$.

In contrast to the Shift in [24], in the construction Shift′, both morphisms a' and b' have to be injective.

Proof. By inspection of the proof of Lemma 2 in [24]. The Only-if case follows, by $\mathcal{M}$ is closed under decomposition, $q, n, m \in \mathcal{M}$ implies $a', b' \in \mathcal{M}$. Thus, $(a', b') \in \mathcal{F}'$. The If case follows because $\mathcal{F}' \subseteq \mathcal{F}$. □

[38] Lemma 1 is an injective version of Lemma 2 in [24]: In [24], arbitrary conditions with arbitrary matching ($n\colon P' \rightarrow H$ arbitrary) are shifted over arbitrary morphisms. In Lemma 1, injective conditions with injective matching ($n\colon P' \rightarrow H$ injective) are shifted over injective morphisms. It hold in every category with $\mathcal{E}'$-$\mathcal{M}$ pair factorizations (see e.g. [15]), in particular in the category ATGI with inclusions.

Example 13. Consider the graph constraint in Example 7 corresponding to the OCL constraint 3 saying that “there is no isolated place”. Shift′ of this constraint over the morphism $\emptyset \to \boxed{\text{:PN}}\boxed{\text{:Pl}}\boxed{\text{:Tr}}$ yields the right application condition

$$
\begin{aligned}
&\forall(\boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}} \to \boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}}\boxed{\text{p':Pl}},\\
&\exists(\boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}}\boxed{\text{p':Pl}}\xrightarrow{\text{preArc}}\boxed{\text{:TPArc}}) \vee \exists(\boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}}\boxed{\text{p':Pl}}\xrightarrow{\text{postArc}}\boxed{\text{:PTArc}})))\\
&\wedge \forall(\boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}} \to \boxed{\text{:PN}}\boxed{\text{p=p':Pl}}\boxed{\text{:Tr}},\\
&\exists(\boxed{\text{:PN}}\boxed{\text{p=p':Pl}}\xrightarrow{\text{preArc}}\boxed{\text{:TPArc}}\boxed{\text{:Tr}}) \vee \exists(\boxed{\text{:PN}}\boxed{\text{p=p':Pl}}\xrightarrow{\text{postArc}}\boxed{\text{:PTArc}}\boxed{\text{:Tr}}))\\
\equiv&\exists(\boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}} \to \boxed{\text{:PN}}\boxed{\text{p:Pl}}\xrightarrow{\text{preArc}}\boxed{\text{:TPArc}}\boxed{\text{:Tr}})\\
&\vee \exists(\boxed{\text{:PN}}\boxed{\text{p:Pl}}\boxed{\text{:Tr}} \to \boxed{\text{:PN}}\boxed{\text{p:Pl}}\xrightarrow{\text{postArc}}\boxed{\text{:PTArc}}\boxed{\text{:Tr}})
\end{aligned}
$$

stating that “the `Place` object is connected to a `PTArc` or `TPArc`”. The Shift′ construction proceeds as shown below. Quantors are written next to the morphism arrows. The original constraint can be seen on the left side, and the new condition is in the middle and right part.

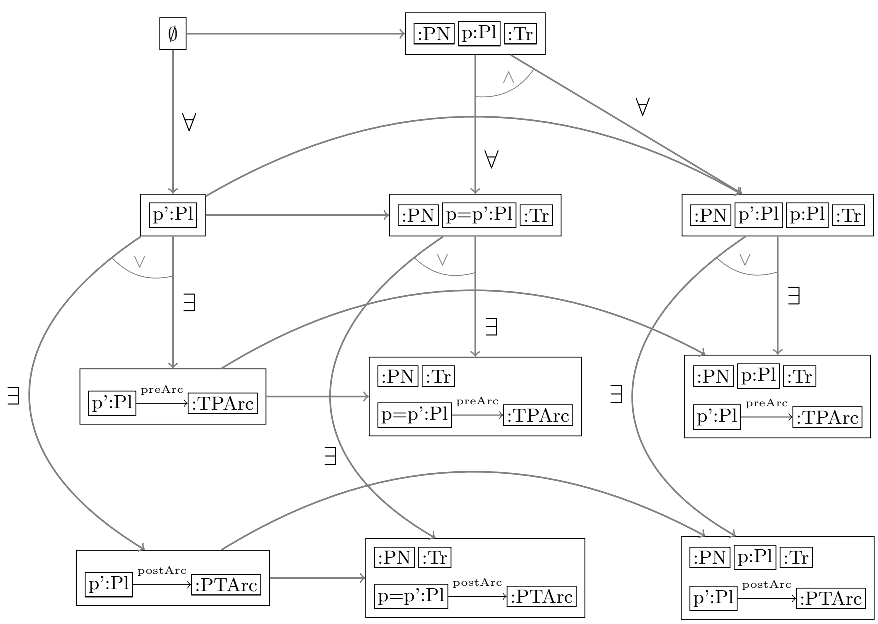
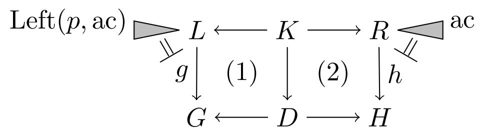


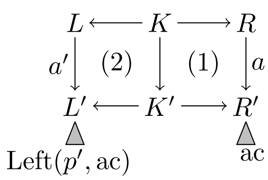
Fig. 4. Illustration of the Shift′-construction

The second result is that conditions can be shifted over rules.

Lemma 2 (shift of conditions over rules [9]). There is a construction Left such that, for each rule $p = \langle L \leftarrow K \to R\rangle$ and each condition ac over R, Left transforms ac via p into an condition $\text{Left}(p, \text{ac})$ over L such that, for each direct transformation $G \Rightarrow_{p,g,h} H$, we have $g \models \text{Left}(p, \text{ac}) \iff h \models \text{ac}$.



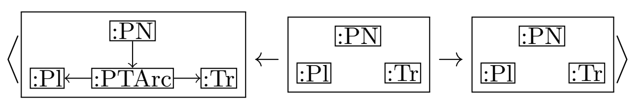
Construction. The construction Left is inductively defined as follows:



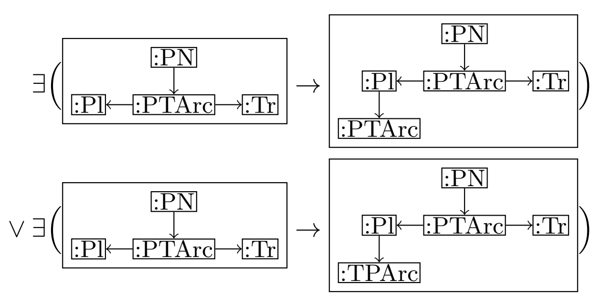
$\text{Left}(p, \mathit{true}_P) = \mathit{true}_{P'}$.
$\text{Left}(p, \exists(a, \text{ac})) = \exists(a', \text{Left}(p', \text{ac}))$
if $R' \Rightarrow_{p^{-1},a,a'} L'$ is a direct transformation by p^{-1} [39]
and $p' = \langle L' \leftarrow K' \rightarrow R' \rangle$ is the rule derived of $L' \Rightarrow_{p,a',a} R'$ and false, otherwise.
$\text{Left}(p, \neg\text{ac}) = \neg\text{Left}(p, \text{ac})$ and
$\text{Left}(p, \wedge_{i \in J} \text{ac}_i) = \wedge_{i \in J} \text{Left}(p, \text{ac}_i)$.

Proof. Immediate consequence of Theorem 6 in [9] using Fact 1. □

Example 14. Consider the rule



that removes a `PTArc` from the model. To save space, we leave out the role names here. Consider now the graph constraint from Example 7 saying "there is no isolated place" and the corresponding right application condition from Example 13 above saying "the `Place` object is connected to a `PTArc` or `TPArc`". The Left construction yields the left application condition



stating that "there must be a second `PTArc` or a `TPArc` connected to the `Place` object". The complete construction is shown below.

[39] For a rule $p = \langle L \leftarrow K \rightarrow R \rangle$, $p^{-1} = \langle R \leftarrow K \rightarrow L \rangle$ denotes the *inverse* rule of p.

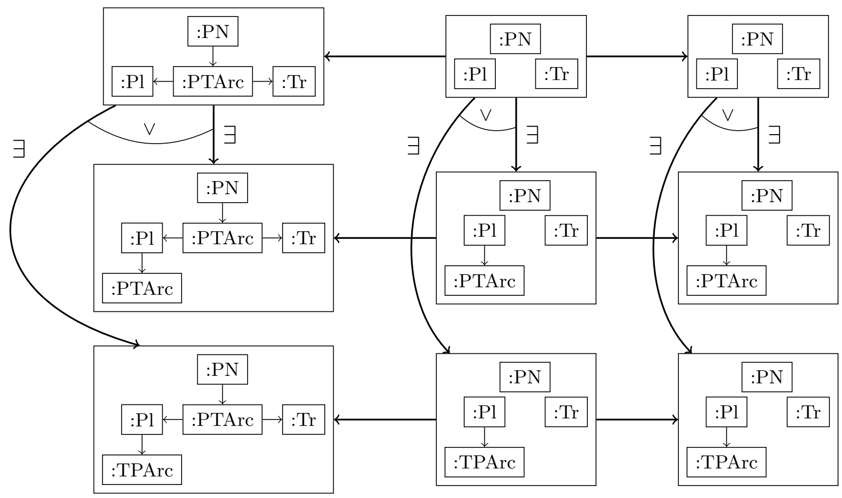


Fig. 5. Illustration of the Left-construction

6 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 [27] 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 [28] 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.

7 Conclusion

The contributions of this paper are the following:

(1) Introduction of a compact notion of graph conditions: lax conditions.

(2) Translation of Essential OCL invariants to nested graph constraints
(3) Correctness of the translation.

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 [29]. 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 [30], 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
9. Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. Math. Structures in Comp. Sci. **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. OMG: Meta Object Facility. `http://www.omg.org/spec/MOF/`

12. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin (2002)
13. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E., ed.: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). Volume 4609 of LNCS., Springer (2007) 600–624
14. Cabot, J., Gogolla, M.: Object constraint language (ocl): A definitive guide. In: Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012. Volume 7320 of LNCS. (2012) 58–90
15. 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
16. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science. Springer (1985)
17. Loeckx, J., Ehrich, H.D., Wolf, M.: Specification of abstract data types. Wiley (1996)
18. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. (2006)
19. Orejas, F.: Symbolic Graphs for Attributed Graph Constraints. J. Symb. Comput. **46**(3) (2011) 294–315
20. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In: FASE. Volume 2984 of LNCS. (2004)
21. Rensink, A.: Representing first-order logic by graphs. In: Graph Transformations (ICGT'04). Volume 3256 of LNCS. (2004) 319–335
22. 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
23. Poskitt, C.M., Plump, D.: Hoare-Style Verification of Graph Programs. Fundamenta Informaticae **118(1-2)** (2012) 135–175
24. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-Adhesive Transformation Systems with Nested Application Conditions. Part 1: Parallelism, Concurrency and Amalgamation. Math. Structures in Comp. Sci. **24**(4) (2014)
25. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Graph-Grammars and Their Application to Computer Science and Biology. Volume 73 of LNCS. (1979) 1–69
26. Ehrig, H., Kreowski, H.J.: Pushout-properties: An analysis of gluing constructions for graphs. Mathematische Nachrichten **91** (1979) 135–149
27. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, Universität Oldenburg (2009)
28. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Graph Transformation (ICGT 2014). Volume 8571 of LNCS. (2014) 17–32
29. 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
30. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In: MoDELS 2010, Oslo, Norway. Proceedings. Volume 6394 of LNCS., Springer (2010) 121–135