From Core OCL Invariants to Nested Graph Constraints: Long Version *

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

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

Abstract. Meta-modeling including the use of the Object Constraint Language (OCL) forms a well-established approach to design domain-specific modeling languages. This approach is purely declarative in the sense that instance construction is not needed and not considered. In contrast, graph grammars allow the stepwise construction of instances by the application of transformation rules. In this paper, we consider meta-models with Core OCL invariants and translate them to nested graph constraints for typed attributed graphs. Models and meta-models are translated to instance and type graphs. We show that a model satisfies a Core OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint. The aim of this work is to establish a first formal relation between meta-modeling and the theory of graph transformation including constraints to come up with an integrated approach for defining modeling languages in an optimal way in the future.

Keywords: Meta modeling, OCL, graph constraints, application conditions

1 Introduction

The trend towards model-based and model-driven software development causes a need of new, mostly domain-specific modeling languages with well-designed tool support. Therefore we need methods and techniques to define modeling languages and their tooling precisely and also intuitively. A comprehensive language definition needs the declarative as well as the constructive paradigm to specify language properties, to construct and recognize language instances as well as to modify them. Nowadays, modeling languages are typically defined by meta-models following purely the declarative approach. In this approach, language properties are specified by the Object Constraint Language (OCL) [1].

In contrast, graph grammars have shown to be suitable and natural to specify visual languages in a constructive way, by using graph transformation [2].

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

Recently, nested graph constraints [3] have been developed to include also the declarative element into graph grammars. To ensure that a graph grammar fulfills a set of graph constraints, they can be translated to application conditions of graph rules such that all graphs fulfilling the constraints in the beginning keep on fulfilling them after applying graph rules being extended by translated application conditions.

While typed attributed graphs form an adequate formalization of instance models that are typed over a meta-model [4], the relation of OCL constraints to nested graph constraints has not been considered yet. We are interested in investigating this relation, since the translation of graph constraints to application conditions for rules opens up a way to combine declarative and constructive elements in a formal approach. By translating OCL to nested graph constraints, such an integration of declarative and constructive elements becomes possible also in the meta-modeling approach. It shall open up a way to translate OCL constraints to application conditions of model transformation rules making applications as e.g. auto-completion of model editing operations to consistent models possible.

As a basis, models and meta-models (without OCL constraints) are translated to instance and type graphs. This translation includes straitening of meta-models w.r.t. language definition. They mainly comprise the restriction to binary associations and the omission of operations. In the context of meta-modeling, i.e., language definition, we are interested in OCL to specify well-formedness rules for meta-model elements. In this paper, we investigate the relation of meta-models including OCL constraints and nested graph constraints for typed attributed graphs. It turns out that Core OCL invariants [5], i.e. Boolean expressions over navigations based on the type system, can be well translated to nested graph constraints. The aim of this work is to establish a first formal relation between meta-modeling and the theory of graph transformation to come up with an integrated approach for defining modeling languages in an optimal way in the future.

This paper is structured as follows: The next section presents OCL in a nutshell focusing on Core OCL invariants. Section 3 shows typed attributed graphs and graph morphisms as well as nested graph conditions. Section 4 presents our main contribution of this paper, the translation of Core OCL invariants to nested graph constraints. Section 5 discusses how Core OCL invariants can be translated to equivalent application conditions of graph rules. Section 6 compares to related work and concludes the paper.

2 Core OCL Invariants

In this section, we recall Core OCL constraints presenting a small example first and formally defining their syntax and semantics thereafter, according to the work by Richters [6] that went into the OCL specification by the OMG [1]. For

illustration purposes, we use the following meta-model for simple Petri nets to recall OCL.

Example 1. A Petri net (*PetriNet*) is composed of places (*Place*) or transitions (*Transition*) which are linked together by arcs (*ArcTP* for linking exactly one transition to one place; *ArcPT* for linking exactly one place to one transition). Places and transitions can have an arbitrary number of incoming (*preArc*) and outgoing (*postArc*) arcs. Finally, Petri net markings are defined by the *token* attribute of places. However, this meta-model allows to build invalid models. For example, one can model a transition having no incoming arc, i.e., the transition can never be fired. Therefore, we complement the meta-model with invariants formulated in OCL.

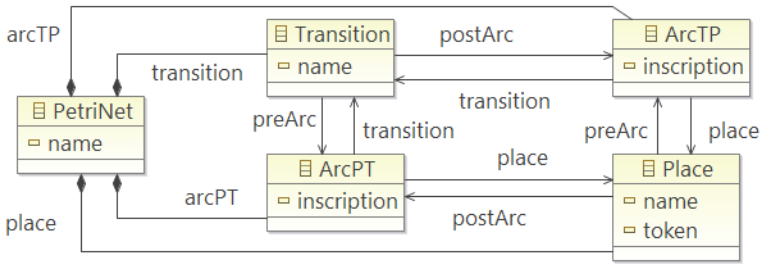


Fig. 1. Meta-model for simple Petri nets

1. A transition has incoming arcs.
   `context Transition inv: self.preArc -> notEmpty()`
2. The number of tokens on a place is not negative.
   `context Place inv: self.token >= 0`
3. The name of a transition is not empty.
   `context Transition inv: self.name <> ' '`
4. The Petri net is not empty.
   `context Petrinet inv: self.transition -> notEmpty() or self.place -> notEmpty()`
5. 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())`
6. There is no isolated place.
   `context Place inv: self.preArc -> notEmpty() or self.postArc -> notEmpty()`

7. Each two places of a Petri net have different names.
   `context Petrinet inv: self.place -> forAll(p1:Place | self.place -> forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))` or alternatively
   `context Petrinet inv: self.place -> forAll(p1:Place,p2:Place | p1 <> p2 implies p1.name <> p2.name)`
8. Each transition of a Petri net is connected with a place.
   `context Petrinet inv: self.transition -> forAll(t:Transition | t.preArc.place -> notEmpty() or t.postArc.place -> notEmpty())` or alternatively
   `context Petrinet inv: self.transition -> forAll(t:Transition | t.preArc -> exists(apt:ArcPT | apt.place -> notEmpty()) or t.postArc -> exists(atp:ArcTP | atp.place -> notEmpty()))`
9. There is at least one place in a Petri net with a number of tokens greater than 0.
   `context Petrinet inv: self.place -> exists(p:Place| p.token>0)`

2.1 OCL language description

In the following, we give a more detailed description of the OCL language. We first give an overview on the OCL type system. Then, 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.

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`, `ArcTP`, and `ArcPT`. For custom type instances, OCL provides basic operations like equality (`=`) and inequality (`<>`) as used in invariant 7 of Example 1 (`p1 <> p2`). *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 `Place::token::Integer` (see Figure 1). Again, basic operations depending on the concrete type are provided. For instance, the invariants of Example 1 use relational operators on Integer (`self.token >= 0` in invariant 2), (in)equality operators on String (`self.name <> ' '` in invariant 3), and logical operations on Boolean (logical `or` in invariant 4). 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 single-valued return type (i.e., custom or predefined type) or in a multi-valued type, more precisely in a set. For example, in invariant 7 of Example 1 the navigation `p1.name` results in a single value of predefined type `String` whereas in invariant 1 the navigation `self.preArc` [3] yields a set of type `ArcPT`. If in the latter navigation no such arc exists, the result is an *empty* set whereas in the case of multiplicity `0..1` the absence of an appropriate value yields to `null` being the only value of bottom type `OclVoid`. Further navigation from a multi-valued result through a second dot yields a bag-valued result. For example, navigating `somePetriNet.arcTP.place` would return a bag since different arcs could point at the same place which consequently has to be returned multiple times.

Logic in OCL We mentioned above that `OclVoid` (1) is a subtype of any custom and predefined type, i.e. also of predefined type `Boolean`, and (2) 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`. We use the latter one in invariant 7 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` [7]. Invariant 7 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 invariant 9.

OCL collection type operations In this section, we give a rough overview on some selected 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.place->select(token=0)` filters those places from the set of all places within the Petri net carrying no

[3] In this invariant, we use key word `self` to represent an instance of type `Transition`.

token whereas `somePetriNet.place->any(token=0)` non-deterministically returns one such place. *Extraction operations* extract some information from the given collection except of 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.place->collect(name)` returns a bag (!) of `String` values. 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 4 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.

2.2 Core OCL

In this section, we present Core OCL, a (first) restricted version of OCL. We use it for a first translation step in the remainder of this paper. In Core OCL, we concentrate on invariants only being the most commonly used kinds of OCL constraints [8]. The kind of OCL constraints we consider belong to `Core OCL` (the OCL type system and the language concepts that realize model navigation) and partially to `Assertion OCL` (invariants), taken from the modularization of OCL as proposed in [9]. They are *set-based* (we use sets only as collection type) and *operation-reduced*. Besides usual operations in basic data types, this means that we concentrate on selected Boolean-typed set operations only (*isEmpty*, *notEmpty*, *exists*, and *forAll*) and user-defined operations are not allowed. The following paragraphs present the abstract syntax and semantics of Core OCL being extracted from [10] and [6] and considerably simplified.

Abstract syntax of Core OCL Figure 2 shows the Core OCL meta-model whose meta-classes are embedded in the corresponding UML meta-model. Selected UML meta-classes have a gray-colored background. As illustrated in the left part of Figure 2, the UML 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` being typed by the constrained element of the invariant. The concrete specification is given by a subclass of `OclExpression`. Such a subclass is either a `VariableExpression` (to refer to a variable), a `LiteralExp` (to refer to a basic type literal, e.g. String *foo*),an `OperationCallExp` (to refer to an operation of a `PrimitiveType` like the addition of integers, or to a set type operation like *isEmpty*), a `PropertyCallExp` (to enable navigation to attributes or association ends, both represented as instances of meta-type `Property`), an `IfExpression` (to provide conditional expressions) or finally an `IteratorExp`

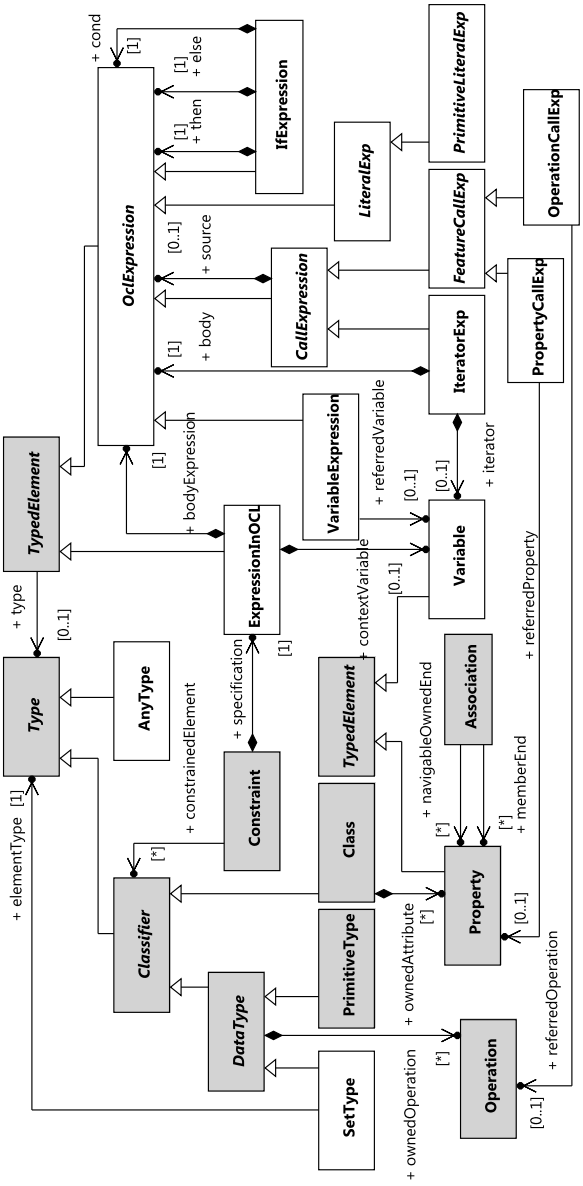


Fig. 2. Meta-model of Core OCL

representing a looping execution on each element of a given set (used in *exists* and *forAll*).

Semantics of Core OCL We describe the semantics of Core OCL based on the formal definitions included in the OCL specification [10], Annex A based on the doctoral thesis by Richters [6]. We prefer this formalization (in contrast to the UML-based specification in [10], Section 10), since it is more suitable for proving the semantic preservation of our translation later on in this paper. Due to space limitations, we present the main definitions and concepts only. For more elaborated specifications we refer to the documents mentioned above. As a first preliminary step, we define an *object model* as follows.

For Core OCL, we straiten the kind of object models being allowed: attributes have primitive types only, there are no operations defined, associations are binary, roles are the default ones indicating source and target, and multiplicities are not set, i.e. range between 0 and *. (It is obvious, however, that multiplicities can be expressed by Core OCL invariants.)

Definition 1 (Core Object Model). Let $DSIG = (S, OP)$ be a data signature with $S = \{Integer, Real, Boolean, String\}$ and a family of corresponding operation symbols OP. A *core object model* over $DSIG$ is a structure $M = (CLASS, ATT, ASSOC, associates, r_{src}, r_{tgt}, \prec)$ where $CLASS$ is a finite set of classes, $ATT = \{ATT_c\}_{c \in CLASS}$ is a family of attributes $att : c \to S$ of class c, $ASSOC$ is a set of associations, $associates$ is a function that maps each association to a pair of participating classes with $associates : ASSOC \to (CLASS \times CLASS)$, r_{src} and r_{tgt} are functions that map each association to a source respectively target role name with $r_{src}, r_{tgt} : ASSOC \to String$ and $r_{src}(assoc) = c_1$ and $r_{tgt}(assoc) = c_2$ 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 define a *system state* of a core 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 a core 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 the primitive data type t where $ATT_c^{\prec}$ is the set of all owned and inherited attribute symbols of a class c [4],

[4] $ATT_c^{\prec} := \bigcup_{c \prec c'} ATT_{c'}$.

– for each $assoc \in ASSOC$ with $associates(assoc) = (c_1, c_2)$, $\sigma_{ASSOC}(assoc) \subset \sigma^{\prec}_{CLASS}(c_1) \times \sigma^{\prec}_{CLASS}(c_2)$ where $\sigma^{\prec}_{CLASS}(c)$ is the set of all objects with type or super type c [5].

The set $States(M)$ consists of all system states $\sigma(M)$ of M .

Definition 3 (Restricted Data Signature). A *restricted data signature* over a restricted object model M is a structure $\Sigma_M = (T_M, \leq_M, \Omega_M)$ where T_M is a set of types, $\leq_M$ is a type hierarchy over T_M, and Ω_M is a set of operations on T_M. The semantics of Σ_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 $I(t)$, $I(\leq_M)$ implies for all types $t, t_0 \in T_M$ that $I(t) \subset I(t_0)$ if $t \leq_M t_0$, and $I(\Omega_M)$ assigns each operation $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_M$ a total function $I(\omega) = I(t_1) \times \cdots \times I(t_n) \to I(t)$.

In a restricted data signature, T_M consists of all basic types (S in $DSIG$), all object types (for each $c \in CLASS$ there is an object type $t_c \in T_M$), and the collection type `Set(t)` for an arbitrary $t \in T_M$ [6]. Example type interpretations are $I(Real) = \mathbb{R}$, $I(t_c) = \biguplus\{oid(c') \mid c' \in CLASS \wedge c' \prec c\}$, and $I(Set(t)) = F(I(t))$ with $F(I(t))$ denoting the set of all finite subsets of $I(t)$.

Ω_M consists of an exhaustive set of predefined operations on primitive data types (like the addition of integers), operations on object types ($allInstances_{t_c}$ for obtaining all objects of type t_c, operations $a : t_c \to t$ to access type attributes, and operations $c' : t_c \to t_{c'}$ with $assoc \in ASSOC$ and $associates(assoc) = (c, c')$ to access navigable association ends of a given type), operations on sets ($isEmpty$, $notEmpty$, and the constructor operation $mkSet_t$ for creating a set with elements of type t), and operations equality and inequality for all types in T_M. Example interpretations of operations are: $I(+_{Integer})(i, j) = i + j$, $I(\mathtt{42}) = 42$, $I(allInstances_{t_c}) = \sigma_{CLASS}(c)$, $I(att : t_c \to t) = \sigma_{ATT}(att)(\underline{c})$ with $\underline{c} \in \sigma_{CLASS}(c)$, $I(c' : t_c \to t_{c'}) = \{\underline{c'} \mid (\underline{c}, \underline{c'}) \in \sigma_{ASSOC}(assoc)\}$, and $I(notEmpty(S)) = (S \neq \varnothing)$.

For specifying expressions for Core OCL we use a signature over an object model M as described 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 $\beta : Var_t \to I(t)$ which map variable names to values.

Definition 4 (Core OCL Expressions). Let $Var = \{Var_t\}_{t \in T_M}$ be a family of variable sets indexed by types $t \in T_M$. The family of *Core OCL expressions* over Σ_M is given by $Expr = \{Expr_t\}_{t \in T_M}$ of sets of expressions. An expression in $Expr$ is a

– *VariableExpression*: $v \in Expr_t$ for all variables $v \in Var_t$,

[5] $\sigma^{\prec}_{CLASS}(c) := \bigcup_{c' \prec c} \sigma_{CLASS}(c')$

[6] The set of types reflects the type hierarchy in the left part of Figure 2.

- *OperationExpressions*: $e := \omega(e_1, \cdots, e_n) \in Expr_t$ for each operation symbol $\omega : t_1 \times \cdots \times t_n \to t \in \Omega$ and for all $e_i \in Expr_{t_i} (1 \leq i \leq n)$.
- *IfExpressions*: For all $e_1, e_2, e_3 \in Expr_{Boolean} : e :=$ `if` e_1 `then` e_2 `else` $e_3 \in Expr_{Boolean}$.
- *IteratorExpressions*: For all $s \in Expr_{Set(c)}, v \in Var_c$, and $b \in Expr_{Boolean}$: $e := s \to exists(v \mid b) \in Expr_{Boolean}$ and $e := s \to forAll(v \mid b) \in Expr_{Boolean}$.

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 semantics of an R1-OCL expression $e \in Expr_t$ is a function $I[\![e]\!] : Env \to I(t)$. Both, syntax and semantics, are defined inductively as follows.

- *VariableExpression*: $v \in Expr_t$ for each variable $v \in Var_t$ and $free(v) := \{v\}$. [7] Moreover, $I[\![v]\!](\tau) = \beta(v)$ for each $\tau = (\sigma, \beta) \in Env$.
- *OperationExpression*: $e := \omega(e_1, \cdots, e_n) \in Expr_t$ for each operation symbol $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_M$ and for all $e_i \in Expr_{t_i} (1 \leq i \leq n)$ and $free(e) := free(e_1) \cup \cdots \cup free(e_n)$. [8] Moreover, $I[\![\omega(e_1, \cdots, e_n)]\!](\tau) = I(\omega)(\tau)(I[\![e_1]\!](\tau), \cdots, I[\![e_n]\!](\tau))$ for each $\tau \in Env$. Table 1 shows an overview on the syntax and semantics of concrete operation expressions in R1-OCL [9].
- *IfExpression*: For each $e_1, e_2, e_3 \in Expr_{Boolean} : e :=$ `if` e_1 `then` e_2 `else` $e_3 \in Expr_{Boolean}$ with $free(e) := (free(e_1) \cup free(e_2) \cup free(e_3))$. [10] 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 $\tau \in Env$. [11]
- *IteratorExpression*: For each $s \in Expr_{Set(t)}, v \in Var_t$, and $b \in Expr_{Boolean}$: $e := s \to exists(v \mid b) \in Expr_{Boolean}$ and $e := s \to forAll(v \mid b) \in Expr_{Boolean}$. It is: $free(e) := (free(s) \cup free(b)) \setminus \{v\}$. Moreover,

$$I[\![s \to exists(v|b)]\!](\tau) = \begin{cases} false & \text{if } I[\![s]\!](\tau) = \emptyset \\ \bigvee\limits_{1 \leq i \leq n} I[\![b]\!](\sigma, \beta\{v \setminus x_i\}) & \text{if } I[\![s]\!](\tau) = \{x_1, ..., x_n\} \end{cases}$$

$$I[\![s \to forAll(v|b)]\!](\tau) = \begin{cases} true & \text{if } I[\![s]\!](\tau) = \emptyset \\ \bigwedge\limits_{1 \leq i \leq n} I[\![b]\!](\sigma, \beta\{v \setminus x_i\}) & \text{if } I[\![s]\!](\tau) = \{x_1, ..., x_n\} \end{cases}$$

[7] This means, that a `VariableExpression` (see Figure 2) refers to a variable, being either a context variable or an iterator variable.

[8] Operations in Ω_M include: predefined operations on data types (`OperationCallExp`), class attribute operations, navigable association end operations (both `PropertyCallExp`), and constants (`LiteralExp`).

[9] Selected operations for primitive types only.

[10] Refers to an `IfExpression` in Figure 2.

[11] Alternatively, we can define the semantics of a conditional expression by using the equivalent logical expression: $I[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!](\tau) = ((I[\![e_1]\!](\tau) \wedge I[\![e_2]\!]) \vee (\neg I[\![e_1]\!] \wedge I[\![e_3]\!]))$.

Table 1. Syntax and semantics of concrete operation expressions in Core OCL

	Operation $\omega \in \Omega_M$	**Syntax** $e \in Expr$	**Semantics** $I[\![e]\!](\tau)$ with $\tau = (\sigma, \beta) \in Env$
All Types	$=: t \times t \to Boolean$	e_1 `=` $e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_t$	$I[\![e_1]\!](\tau) = I[\![e_2]\!](\tau)$
	$\neq: t \times t \to Boolean$	e_1 `<>` $e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_t$	$I[\![e_1]\!](\tau) \neq I[\![e_2]\!](\tau)$
Primitive Types	$+ : Int \times Int \to Int$	e_1 `+` $e_2 \in Expr_{Integer}$ with $e_1, e_2 \in Expr_{Integer}$	$I[\![e_1]\!](\tau) + I[\![e_2]\!](\tau)$
	$and : Boolean \times Boolean \to Boolean$	e_1 `and` $e_2 \in Expr_{Boolean}$ with $e_1, e_2 \in Expr_{Boolean}$	$I[\![e_1]\!](\tau) \wedge I[\![e_2]\!](\tau)$
	$\omega :\to String$	`'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 $e_c \in Expr_{t_c}$	$\sigma_{ATT}(att)(I[\![e_c]\!](\tau))$
	$c' : t_c \to t'_c$ with $associates(assoc) = (c, c')$	$e_c.c' \in Expr_{Set(t'_c)}$ with $e_c \in Expr_{t_c}$	$\{\underline{c'} \mid (I[\![e_c]\!](\tau), \underline{c'}) \in \sigma_{ASSOC}(assoc)\}$
Set Types	$isEmpty : Set(t) \to Boolean$	$s \to isEmpty() \in Expr_{Boolean}$ with $s \in Expr_{Set(t)}$	$I[\![s]\!](\tau) = \varnothing$
	$notEmpty : Set(t) \to Boolean$	$s \to notEmpty() \in Expr_{Boolean}$ with $s \in Expr_{Set(t)}$	$I[\![s]\!](\tau) \neq \varnothing$

for each $\tau \in Env$. [12] [13]

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

Definition 5 (Core OCL Invariant). A *Core OCL invariant* is a Boolean Core 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 Core OCL invariants over M.

Remark 1. The following properties hold for Core OCL invariants:

1. An invariant `context v:C inv : <expr>` is equivalent to `C.allInstances` $\rightarrow$ `forAll(v|<expr>)`. So, the overall semantics of an invariant is equal to the semantics of the equivalent Core OCL expression.
2. Navigation expressions to collections are not contained in other navigation expressions, e.g., `somePetriNet.arcTP.place->notEmpty()` is replaced by `somePetriNet.arcTP->exists(a:ArcTP | a.place->notEmpty())`.
3. Iterator expressions are completed, i.e. the iterator variable is explicitly declared. Moreover, a variable declaration is always complete, i.e. consists of a variable name and a type name.
4. If `nav op nav` occurs for the same navigation expression nav and $op(nav, nav) = true$ then `nav op nav` can be replaced by `true`.
5. Note that `v1 = v2.r` and `v1 <> v2.r` are not possible since the result of `v1` is an object and `v2.r` yields a set of objects.

Remark 2. The following items summarize the characteristics of Core OCL being used in a first translation to graph constraints in the remainder of this paper:

1. Core OCL is exclusively used to formulate invariants.
2. Core OCL is type-restricted and set-based.
   (a) Each class attribute has a simple data type.
   (b) Associations between classes are binary.
   (c) Association end roles are default ones indicating source and target.
   (d) Association end multiplicities are not set, i.e. range between 0 and *.
   (e) As collection type, Core OCL uses sets only.
   (f) Conditionals allow Boolean expressions for their alternatives only.
3. Core OCL is operation-reduced.
   (a) Classes do not have operations.

[12] Note, that in [10] and [6] the semantics of iterator expressions is defined in a more common but slightly different way. The definition presented here is quite obvious. However, the equivalence of both definitions has to be shown.

[13] Alternatively, we can define the semantics of the *forAll* iteration by using the equivalent logical expression: $I[\![s \rightarrow forAll(v \mid b)]\!](\tau) = I[\![s \rightarrow \neg exists(v \mid \neg b)]\!](\tau)$.

(b) Core OCL uses selected Boolean-typed set operations only (*isEmpty*, *notEmpty*, *exists*, and *forAll*).
(c) User-defined operations respectively queries are not allowed.

Although being restricted in various directions, Core OCL covers a relevant part of OCL. First of all, it is meant to be used for formulating invariants over meta-models which results in restricting the kind of object models and the kind of constraints. Furthermore, it identifies that part of OCL being "first order" assuming that the semantics of OCL formulas is defined over finite models only. In addition, its semantic is restricted to the usual two-valued semantics which makes sense since we consider standard operations only.

3 Nested graph conditions

In the following, we recall the formal definition of typed, attributed graphs with node type inheritance as presented in [11]. They form the basis to define attributed graph conditions.

3.1 Graphs

Attributed graphs as defined here allow to attribute nodes only while the original version [11] supports also the attribution of edges.

Definition 6 (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. $f_V \circ src^1_G = src^2_G \circ f_E$.

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

We assume that the reader is familiar with the basics of algebraic specification. In [11], 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. [12,13]. The definition of attributed graphs generalizes largely the one in [14] by allowing variables and a set of formulas that constrain the possible values of these variables. The definition is closely related to symbolic graphs [15].

Definition 7 (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[14] with free variables in X. A set $\{F_1, \ldots, F_n\}$ of formulas can 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 $f(x)$.

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

Remark 3. 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 [14] 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 [14].

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 14) 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**.

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

Proof.
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 [14], 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. □

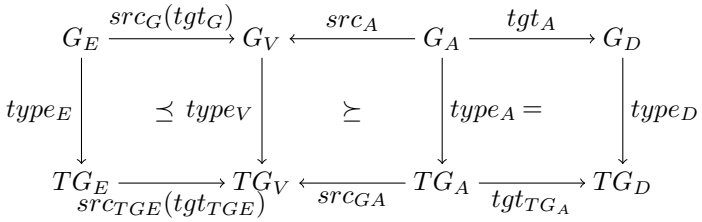
Definition 8 (Typed attributed graph over $ATGI$). An *attributed type graph with inheritance* $ATGI = (TG, Z, I)$ consists of an A-graph, a final $DSIG$-algebra Z, and a simple[15] inheritance graph I with $I_V = TG_V$. For each node $v \in I_V$, the *inheritance clan* is defined by $clan_I(v) = \{v' \in I_V \mid \exists\ path\ v' \overset{*}{\to} v\ in\ I\}$[16]. If I is discrete[17], $ATGI$ 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* $type : AG \to ATGI$.

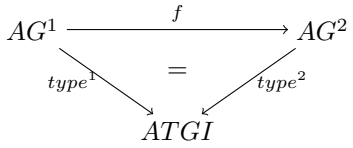
A *clan morphism* $type$ consists of typing functions $type_V\colon G_V \to TG_V$, $type_D\colon G_D \to TG_D$ for nodes, $type_E\colon G_E \to TG_E$, $type_A\colon G_A \to TG_A$ for edges, and the unique final $DSIG$-homomorphism $type_{DSIG}\colon D \to Z$ such that:

- $type_V \circ src_{GE} \preceq clan_I \circ src_{TGE} \circ type_E$[18]
- $type_V \circ tgt_{GE} \preceq clan_I \circ tgt_{TGE} \circ type_E$
- $type_V \circ src_{GA} \preceq clan_I \circ src_{TGA} \circ type_A$
- $type_D \circ tgt_{GA} = tgt_{TGA} \circ type_A$
- $type_{\mathrm{DSIG},s} = type_{D|D_s}$ for all $s \in S$.

[15] A graph is *simple* if it has neither multiple edges nor loops.
[16] $v' \overset{*}{\to} v$ in I stands for a directed path in I from v' to v of length ≥ 0.
[17] A graph is *discrete* if the edge set is empty.
[18] 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 $clan_I(B) = \{clan(v) \mid v \in B\}$.



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 2 (properties of typed attributed graphs).

1. $ATGI$-graphs and $ATGI$-morphisms form the category $\mathbf{AGraphs_{ATGI}}$.
2. The category has pushouts.
3. 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 $\mathbf{AGraphs_{ATGI}}$.

Proof. 1. is straightforward. Statements 2. and 3. follow directly from [14], Lemma 13.13. □

3.2 Nested graph conditions on typed attributed graphs

Nested graph conditions [3] 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 conditions over $ATGI$-graphs, closely related to attributed graph constraints [15] and E-conditions [16].

Definition 9 (nested graph conditions). A *(nested) graph condition* on typed attributed graphs, short *$ATGI$-condition*, over a graph P is of the form *true*, $\exists(a, c)$, or $\exists(P \sqsupseteq C, c)$[19] where $a\colon P \to C$ is an injective morphism and c is an $ATGI$-condition over C. Boolean formulas over $ATGI$-conditions over P yield $ATGI$-conditions over P, that is $\neg c$ and $\bigwedge_{i \in I} c_i$ are $ATGI$-conditions over P. Conditions over $\emptyset$ are also called *constraints*. In the context of rules (see Section 5), conditions are also called *application conditions*.

[19] Conditions of the form $\exists(P \sqsupseteq C, c)$ are syntactic sugar, i.e. they can be expressed in terms of the other constructs. See Lemma 1.

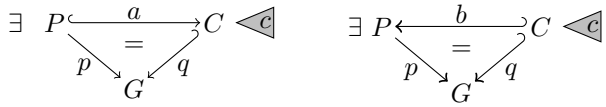
Notation. Graph conditions may be written in a more compact form: $\exists a$ abbreviates $\exists(a, \mathit{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, $\vee_{i\in I} c_i$ abbreviates $\neg \wedge_{i\in I} \neg c_i$, and $c \Rightarrow c'$ abbreviates $\neg c \vee c'$. For an injective morphism $a\colon P \hookrightarrow C$ in a condition, we just depict the codomain C, if the domain P can be unambiguously inferred, i.e. if it is known over which graph a condition is.

Example 2 (OCL constraints as graph constraints). OCL constraint `context Place inv: self.token >= 0` in Example 1 is represented as an attributed graph constraint in full and in abbreviated form. The last graph in the condition is decorated by the formula $x \geq 0$, with the notation as in [16]. The attributing DSIG-algebra is the quotient term algebra $T_{\mathrm{DSIG}_{\equiv}}(X)$ where $\equiv$ is the congruence relation on $T_{DSIG}(X)$ induced by $\geq (x, 0)$.

$$\neg\exists\left(\emptyset \to \boxed{\text{self:Place}}, \neg\exists\left(\boxed{\text{self:Place}} \to \boxed{\begin{array}{c}\text{self:Place}\\ \hline \text{token = x}\end{array}} \mid x \geq 0\right)\right)$$

or, in short form: $\forall\left(\boxed{\text{self:Place}}, \exists\ \boxed{\begin{array}{c}\text{self:Place}\\ \hline \text{token} \geq 0\end{array}}\right)$

Definition 10 (Semantics of nested graph conditions). Let $p\colon P \to G$ be a morphism. *Satisfiability* of a condition over P is inductively defined as follows: Every morphism satisfies *true*. Morphism p satisfies $\exists(P \to C, c)$ if there exists an injective morphism $q\colon C \hookrightarrow G$ such that the left diagram below commutes and q satisfies c. Morphism p satisfies $\exists(P \sqsupseteq C, c)$ if there exist injective morphisms $b\colon C \hookrightarrow P$ and $q\colon C \hookrightarrow G$ such that $q = p \circ b$ and q satisfies c (see right diagram below). Morphism 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 $\emptyset$) by a graph is defined as follows: A graph G satisfies a constraint c, short $G \models c$, if the morphism $p\colon \emptyset \hookrightarrow G$ satisfies c.

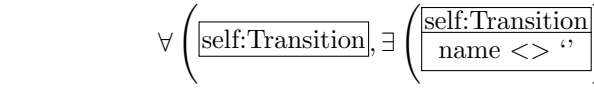
Example 3 (OCL-induced graph constraints). Consider the basic OCL constraints 1-9 of Example 1. They can be represented as graph constrains as follows.

1. A transition has incoming arcs.
   `context Transition inv: self.preArc -> notEmpty()`

$$\forall\left(\boxed{\text{self:Transition}}, \exists\left(\boxed{\text{self:Transition}} \xrightarrow{\text{preArc}} \boxed{\text{:ArcPT}}\right)\right)$$

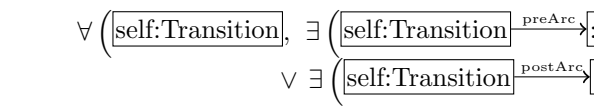
2. The number of tokens on a place is not negative.
   `context Place inv: self.token >= 0`

3. The name of a transition is not empty.
   `context Transition inv: self.name <> ' '`

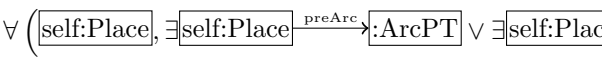


4. The Petri net is not empty.
   `context Petrinet inv: self.transition -> notEmpty() or self.place -> notEmpty()`

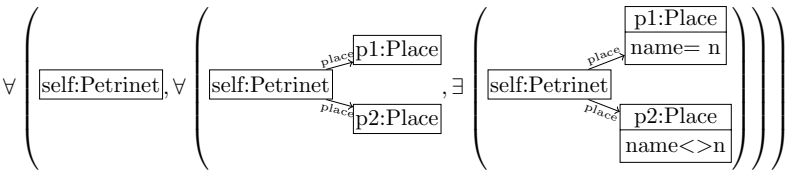
5. There is no isolated transition.
   `context Transition inv: self.preArc -> notEmpty() or self.postArc -> notEmpty()`



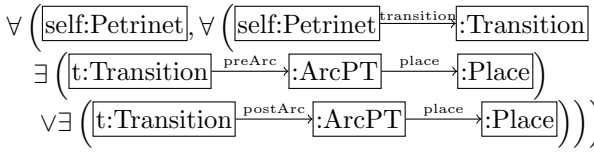
6. There is no isolated place.
   `context Place inv: self.preArc -> notEmpty() or self.postArc -> notEmpty()`



7. Each two places of a Petri net have different names.
   `context Petrinet inv: self.place -> forAll(p1:Place | self.place -> forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))`



8. Each transition of a Petri net is connected with a place.
   `context Petrinet inv: self.transition -> forAll(t:Transition | t.preArc.place -> notEmpty() or t.postArc.place -> notEmpty())`

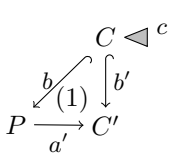


9. There is at least one place in a Petri net with a number of tokens greater than 0.
   `context Petrinet inv: self.place -> exists(p:Place| p.token>0)`

In Definition 9, conditions of the form $\exists(P \sqsupseteq C, c)$ are syntactic sugar: They can be expressed using only a disjunction of conditions of the form $\exists(P \hookrightarrow C, c)$. As Example 3 shows, this containment operator is very useful and makes the condition more succinct.

Lemma 1 (Nested+ $\sqsupseteq\equiv$ Nested). There is a Shift$_{\sqsupseteq}$-construction such that, for each $ATGI$-condition c over P and for each $ATGI$-morphism $b\colon C \hookrightarrow P$, Shift$_{\sqsupseteq}$ transforms c via b into an $ATGT$-condition Shift$_{\sqsupseteq}(b, c)$ without containment operator such that, for each $ATGI$-morphism $n\colon P \to H$, $n \circ b \models c \iff n \models \text{Shift}_{\sqsupseteq}(b, c)$.

Construction. For $ATGI$-morphisms b and $ATGI$-conditions c', let



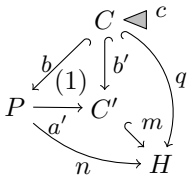
$$\begin{aligned}&\text{Shift}_{\sqsupseteq}(b, c') = \textstyle\bigvee_{b' \in \mathcal{F}} \text{Shift}_{\sqsupseteq}(b'\colon C \hookrightarrow C', c)\\ &\text{if } \mathcal{F} = \{(a', b') \mid (a', b') \text{ jointly surjective, } b' \text{ inj., (1) commutes}\}\\ &\text{is non-empty and } c' = \exists(P \sqsupseteq C, c),\\ &\text{Shift}_{\sqsupseteq}(b, c') = \text{Shift}(b, c') \text{ otherwise.}\end{aligned}$$

Proof. By induction over the structure of the $ATGI$-condition c'.
Base Case: $c' = \mathit{true}$. See the proof for Shift in [17].
Hypothesis: For an injective $ATGI$-morphism $b\colon C \hookrightarrow P$ and an arbitrary $ATGI$-morphism $n\colon P \hookrightarrow H$, $n \circ b \models c \iff n \models \text{Shift}_{\sqsupseteq}(b, c)$.
Induction Step: For $ATGI$-conditions c' of the form $\exists(a, c)$, $c_1 \wedge c_2$, $\neg c$, see the proof for Shift in [17]. For $ATGI$-conditions c' of the form $\exists(P \sqsupseteq C, c)$, we proceed as follows:

Only if. Let $n \circ b \models c'$. Then there exists injective $ATGI$-morphisms $b\colon C \hookrightarrow P$ and $q\colon C \hookrightarrow G$ such that $q = n \circ b \models c$. By $\mathcal{E}'$-$\mathcal{M}$ pair factorization[20], there exist an object C' and morphisms $a'\colon P \to C'$, $b'\colon C \to C'$, and $m\colon C' \hookrightarrow H$ such that (a', b') is jointly surjective and m is injective such that $m \circ a' = n$ and $m \circ b' = q$. By commutativity, $n \circ b = (m \circ a') \circ b = m \circ b'$ and $q = m \circ b' \models c$. By the hypothesis, $m \circ b' \models c \Leftrightarrow m \models \text{Shift}_{\sqsupseteq}(b', c)$. By (a', b') jointly surjective, m injective, and (1) commutative, $b' \in \mathcal{F}$, and $m \models \text{Shift}_{\sqsupseteq}(b', c)$, we obtain

[20] See e.g. [11]

$n = m \circ a' \models \text{Shift}_{\sqsupseteq}(b, c')$.



If. Let $n \models \text{Shift}_{\sqsupseteq}(b, c')$. Then there is some $(a', b') \in \mathcal{F}$ such that (a', b') is jointly surjective, b' is injective, $a' \circ b = b'$, and $n \models \text{Shift}_{\sqsubseteq}(b', c)$. By definition of satisfiability, there is some injective $ATGI$-morphism $m\colon C' \hookrightarrow H$ such that $m \circ a' = n$ and $m \models \text{Shift}_{\sqsubseteq}(b', c)$. By inductive hypothesis, $m \models \text{Shift}_{\sqsubseteq}(b', c) \Leftrightarrow m \circ b' \models c$. Then there are injective $ATGI$-morphism $b\colon C \hookrightarrow P$ and $q\colon C \hookrightarrow H$ such that $q = n \circ b$ and $n \circ b \models \exists(P \sqsupseteq C, c)$. This completes the inductive proof. □

4 Translation of Meta-models with Core OCL Invariants

To translate Core OCL invariants, we first show how to translate the type information of meta-models, i.e. core object models, to attributed type graphs with inheritance [18] are considered. Thereafter, system states are translated to typed attributed graphs. Having these ingredients available, our main contribution, the translation of Core OCL invariants is presented, together with two example translations. Finally, completeness and correctness of the translation are shown.

4.1 Type and state correspondences

To define the translation of Core OCL invariants to graph constraints, we translate a given object model to its corresponding type graph.

Definition 11 (Type Correspondence). Let $DSIG = (S, OP)$ be a data signature and Z the final DSIG-algebra. Given a core object model $M = (CLASS, ATT, ASSOC, associates, \prec)$ over $DSIG$, it corresponds to an attributed type graph with inheritance $ATGI = ((TG, Z), Inh)$ with type graph $TG = (TG_V, TG_D, TG_E, TG_A, src_G, tgt_G, src_A, tgt_A)$ and inheritance relation Inh if there is a *correspondence relation* $corr_{type} = (corr_{CLASS}, corr_{ATT}, corr_{ASSOC})$ with bijective mappings

- $corr_{CLASS} : CLASS \to TG_V$ with $\forall c1, c2 \in CLASS$:
  $c_1 \prec c_2 \iff (corr_{CLASS}(c_1), corr_{CLASS}(c_2)) \in Inh$,
- $corr_{ATT} : ATT \to TG_A$ with
  $src_A(corr_{ATT}(att)) = corr_{CLASS}(c)$ for $c \in CLASS$ and
  $tgt_A(corr_{ATT}(att)) = x$ if $att : c \to s \in ATT_c$ and $\{x\} = Z_s$ with $s \in S$,

– $corr_{ASSOC} : ASSOC \to TG_E$ with $src_G \circ corr_{ASSOC} = corr_{CLASS} \circ pr_1$ and $tgt_G \circ corr_{ASSOC} = corr_{CLASS} \circ pr_2$ with $associates(a) = \langle c1, c2 \rangle$, $pr_i(a) = c_i$ for $i = 1, 2$, $c1, c2 \in Class$ and $a \in ASSOC$.

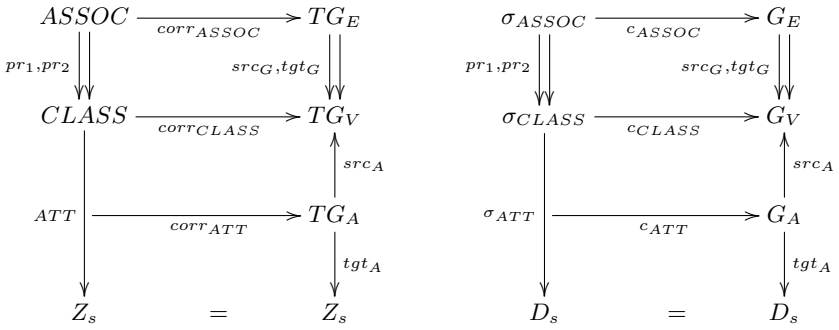


Fig. 3. Type and system state correspondences

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

Definition 12 (State Correspondence). Let M be a core object model and $ATGI$ an attributed type graph with inheritance, both defined over data signature $DSIG = (S, OP)$. We assume that $I(s) = D_s$ for all sorts $s \in S$. Furthermore, let $corr_{type}(M) = ATGI$ be a type correspondence.

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* $corr_{state} = (c_{CLASS}, c_{ATT}, c_{ASSOC}) : States(M) \to Graph_{ATGI}$ 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 $l = (o1, o2) \in \sigma_{ASSOC}(assoc)$ and $pr_i(l) = o_i$ for $i = 1, 2$.
Furthermore, $type_{G_E} \circ c_{ASSOC}(\sigma_{ASSOC}) = corr_{ASSOC}(ASSOC)$.

4.2 Translation of Core OCL invariants

To get an initial understanding on how Core OCL invariants shall be translated to graph conditions, we take a pattern-based approach. The principle idea is that navigation expressions are translated to graphs and graph morphisms while the usual Boolean operations correspond to each other directly. The subset-operator of graph conditions is useful to correspond to iterating variables in iterator expressions. In Figure 4, basic OCL patterns and their corresponding graph constraint patterns are depicted. In these patterns, "Class", "v", "v1","v2", "b" , "c", and "r" are variables for model elements. Non-terminal `<op>` can be replaced by some comparator such as $=, <>, <$. Non-terminal `<expr>` may be replaced by any Core OCL expression.

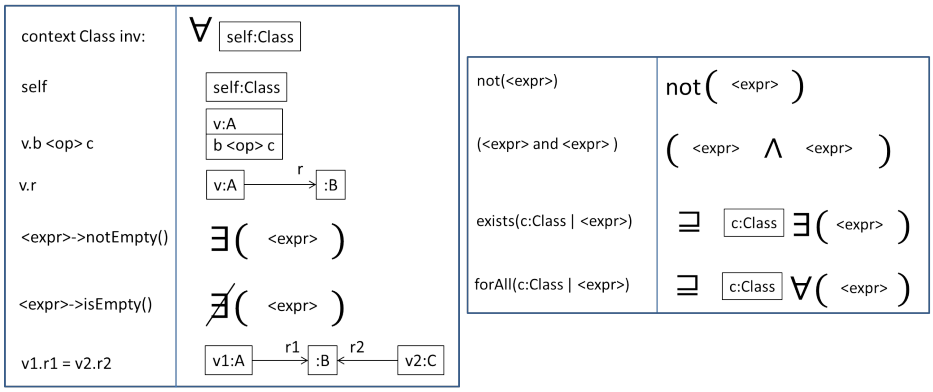


Fig. 4. Translation of basic OCL patterns to graph constraint patterns

In the following, we define the translation of Core OCL invariants as outlined in the beginning of this section. The translation is basically structured along the definition of Core OCL expressions given in Def. 4. However, operation expressions, If expressions and iterator expressions are distinguished along their result type yielding navigations (with $Set(t)$, $t \in CLASS$ as result type) and (Boolean) expressions. In the following definition, rules 1 and 2 translate the header of a CoreOCL invariant, rules 4 - 12 translate Boolean expressions, rules 13 - 15 translate basic object comparisons, rules 16 - 17 translate attribute value comparisons, and rules 18 - 19 translate basic navigation expressions and variables.

Definition 13 (Constraint translation). Let $M = (CLASS, ATT, ASSOC, associates, \prec)$ be a core object model with $ATGI = corr_{type}(M)$ being the corresponding attributed type graph with inheritance and $t : Expr \to T$ a typing function which returns the type of an OCL expression (for T see Section 2). Let furthermore Invariant_M be the set of Core OCL invariants over M as defined in

Def. 5 and GraphConstraint$_{ATGI}$ be the set of all graph constraints as defined in Definition 9. Then, the *translation functions*

- invariant translation: tr_I: Invariant$_M$ $\rightarrow$ GraphConstraint$_{ATGI}$
- expression translation tr_E: $Expr_{Boolean}$ $\rightarrow$ GraphConstraint$_{ATGI}$
- navigation translation tr_N: $Expr_c$ $\rightarrow$ Graph$_{ATGI}$ with $c \in CLASS$
- variable translation tr_V: Var_c $\rightarrow$ Graph$_{ATGI}$ with $c \in CLASS$

are defined as follows:

1. tr_I('context' C 'inv:' expr) = $\forall$ (self:C, tr_E(expr))
2. tr_I('context' var ':' C 'inv:' expr) = $\forall$ ($tr_V(var)$, tr_E(expr))
3. tr_E (expr) = tr_E(setOpCallExpr) | tr_E(basicExpr)
   | tr_E(boolExpr) | tr_E(iteratorExpr)
4. tr_E(boolExpr) = *true* if boolExpr ::= '*true*'
5. tr_E(boolExpr) = ($\neg$ tr_E(expr)) if boolExpr ::= '(' 'not' expr ')'
6. tr_E(boolExpr) = (tr_E(expr1) op_g tr_E(expr2)) with $op_g \in \{\wedge, \vee\}$
   if boolExpr ::= '(' expr1 op_b expr2 ')' with $op_b \in$ { 'and', 'or'}
7. tr_E(boolExpr) = ($\neg$ tr_E(expr1) $\vee$ tr_E(expr2))
   if boolExpr ::= '(' expr1 'implies' expr2 ')'
8. tr_E(boolExpr) = ((tr_E(cond) $\wedge$ tr_E(expr1)) $\vee$ ($\neg$ tr_E(cond) $\wedge$ tr_E(expr2)))
   if boolExpr ::= '(' 'if' cond 'then' expr1 'else' expr2')'
9. tr_E(setOpCallExpr) = $\neg\exists$ (tr_N(navExpr))
   if setOpCallExpr ::= navExpr '$\rightarrow$ isEmpty()'
10. tr_E(setOpCallExpr) = $\exists$ (tr_N(navExpr))
    if setOpCallExpr ::= navExpr '$\rightarrow$ notEmpty()'
11. tr_E(iteratorExpr) = $\exists$ (tr_N(navExpr) $\sqsupseteq$ tr_V(var), tr_E(expr))
    if iteratorExpr ::= navExpr '$\rightarrow$' exists (' var '|' expr ')'
12. tr_E(iteratorExpr) = $\forall$ (tr_N(navExpr) $\sqsupseteq$ tr_V(var), tr_E(expr))
    if iteratorExpr ::= navExpr '$\rightarrow$' forAll (' var '|' expr ')'
13. (a) tr_E(basicExpr) = $\exists$ (v:t(v) v2:t(v2) $\rightarrow$ v,v2:t(v))[21]
    if basicExpr ::= v '=' v2
    (b) tr_E(basicExpr) = $\exists$ (v:t(v) v2:t(v2))
    if basicExpr ::= v '<>' v2
14. (a) tr_E(basicExpr) = $\exists$ (tr_V(v ':' t(v)) $\xrightarrow[as2]{as}$:t(r))
    if basicExpr ::= v '.' r '=' v '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$

[21] Note that this a "non-injective" condition in the sense of [3], i.e. a condition with non-injective morphism. By [17], for each non-injective condition c and each morphism $p = m \circ e$ with e surjective and m injective, there is an injective condition Shift(e, c) in the sense of Definition 9 such that we have $p \models c \Leftrightarrow m \models$ Shift(e, c). Whenever a non-injective morphism occurs in a condition, we have to replace the whole condition by Shift(e, c).

(b) tr_E(basicExpr) = $\exists$ (tr_V(v ':' t(v)) $\xrightarrow{as}$:t(r) $\xleftarrow{as2}$ tr_V(v2 ':' t(v2)))
if basicExpr ::= v '.' r '=' v2 '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$

15. (a) tr_E(basicExpr) = $\exists$ (:t(r) $\xleftarrow{as}$ tr_V(v) $\xrightarrow{as2}$:t(r2))
if basicExpr ::= v '.' r '<>' v '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$

(b) tr_E(basicExpr) = $\exists$ (tr_V(v) $\xrightarrow{as}$:t(r) $\quad tr_V$(v2) $\xrightarrow{as2}$:t(r2))
if basicExpr ::= v '.' r '<>' v2 '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$

16. tr_E(basicExpr) = $\exists$ ($\frac{v:t(v)}{attr\ op\ x}$)[22] if basicExpr ::= v '.' attr op x and x is a constant or a variable

17. (a) tr_E(basicExpr) = $\exists$ ($\frac{v:t(v)}{\begin{array}{c}attr\ =\ x\\ attr2\ op\ x\end{array}}$)
if basicExpr ::= v '.' attr op v '.' attr2, attr $\neq$ attr2, and x is a new variable with t(x) = t(attr) = t(attr2).

(b) tr_E(basicExpr) = $\exists$ ($\frac{v:t(v)}{attr = x}$ $\frac{v2:t(v2)}{attr2\ op\ x}$)
if basicExpr ::= v '.' attr op v2 '.' attr2, v $\neq$ v2, and x is a new variable with t(x) = t(attr).

18. tr_N(navExpr) = tr_V(v) $\xrightarrow{as}$:t(r)
if navExpr ::= v '.' r and r is a role of as

19. tr_V(var) = v:t(v)
for var::= v ':' t with t = t(v) or var::= v

where $expr, expr1, expr2, boolExpr, setOpCallExpr, basicExpr, iteratorExpr \in Expr_{Boolean}$, $var \in Var_c$, $navExpr, navExpr1, navExpr2 \in Expr_c$, $c \in CLASS$, $v, v2 \in Var_t$, $t \in T_M$, $as, as2 \in ASSOC$, $r = r_{tgt}(as)$, $r2 = r_{tgt}(as2)$, $op \in \{<, >, \leq, \geq, =, <>\}$, $attr \in ATT_c$ and w.l.o.g., $corr_{CLASS}(c) = c$ for $c = t(r)$, $c = t(r2)$, $c = t(v)$ or $c = t(attr)$, $corr_{ASSOC}(as) = as$, $corr_{ASSOC}(as2) = as2$, and $corr_{ATT}(attr) = attr$.

In the following, we show two example translations using OCL invariants of Example 1. Small numbers behind equality signs denote the rules being used.

Example 4 (Translation of OCL constraint 1).

tr_I('context Transition inv: self.preArc→notEmpty()') $=^1$
$\forall$ (self:Transition tr_E('self.preArc→notEmpty()')) $=^{10}$
$\forall$ (self:Transition, $\exists$ (tr_N('self.preArc'))) $=^{18}$
$\forall$ (self:Transition, $\exists$ (tr_V('self:Transition') $\xrightarrow{preArc}$:ArcPT)) $=^{19}$
$\forall$ (self:Transition, $\exists$ (self:Transition $\xrightarrow{preArc}$:ArcPT))

[22] Compare the short notation of attribute conditions in Example 2.

Example 5 (Translation of OCL constraint 5.2). tr_C('context Petrinet inv: self.transition→forAll(t: Transition | t.preArc→notEmpty() or t.postArc → notEmpty()')) $=^1$
∀ (:Petrinet , tr_E('self.transition→forAll(t: Transition | t.preArc→notEmpty() or t.postArc→notEmpty()')) $=^{12}$
∀ (:Petrinet , ∀ (tr_E('self.transition') ⊒ tr_V('t:Transition'),
tr_E('t.preArc→notEmpty() or t.postArc→notEmpty()'))) $=^{18,6}$
∀ (:Petrinet , ∀ (tr_V('self:Petrinet') $\overset{transition}{\longrightarrow}$:Transition ⊒ t:Transition ,
tr_E('t.preArc→notEmpty()') ∨ tr_E('t.postArc→notEmpty()'))) $=^{19,10,10}$
∀ (:Petrinet , ∀ (self:Petrinet $\overset{transition}{\longrightarrow}$:Transition ⊒ t:Transition ,
(∃ (tr_N('t.preArc')) ∨ ∃ (tr_N('t.postArc'))))) $=^{18,18}$
∀ (:Petrinet , ∀ (self:Petrinet $\overset{transition}{\longrightarrow}$:Transition ⊒ t:Transition ,
∃ (tr_V('t:Transition') $\overset{preArc}{\longrightarrow}$:Place) ∨ ∃ (tr_V('t:Transition') $\overset{postArc}{\longrightarrow}$:Place)))
$=^{19}$ ∀ (:Petrinet , ∀ (self:Petrinet $\overset{transition}{\longrightarrow}$:Transition ⊒ t:Transition ,
∃ (t:Transition $\overset{preArc}{\longrightarrow}$:ArcPT) ∨ ∃ (t:Transition $\overset{postArc}{\longrightarrow}$:ArcPT)))

Example 6 (Translation of OCL constraint 7.1).
tr_I('context Petrinet inv: self.place→forAll(p1:Place | self.place →
forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))') $=^1$
∀ (self:Petrinet , tr_E('self.place→forAll(p1:Place | self.place →
forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))')) $=^{12}$
∀ (self:Petrinet , ∀ (tr_E('self.place') ⊒ tr_V('p1:Place'),
tr_E('self.place→forAll(p2:Place | p1<>p2 implies p1.name<>p2.name)'))) $=^{18,12}$
∀ (self:Petrinet , ∀ (tr_V('self') $\overset{place}{\longrightarrow}$:Place ⊒ p1:Place ,
∀ (tr_E('self.place') ⊒ tr_V('p2:Place'),
tr_E('p1 <> p2 implies p1.name <> p2.name')))) $=^{19,18,6}$
∀ (self:Petrinet , ∀ (self:Petrinet $\overset{place}{\longrightarrow}$:Place ⊒ p1:Place ,
∀ (tr_V('self') $\overset{place}{\longrightarrow}$:Place ⊒ p2:Place ,
¬ tr_E('p1 <> p2') ∨ tr_E('p1.name <> p2.name')))) $=^{19}$
∀ (self:Petrinet , ∀ (self:Petrinet $\overset{place}{\longrightarrow}$:Place ⊒ p1:Place ,
∀ (self:Petrinet $\overset{place}{\longrightarrow}$:Place ⊒ p2:Place ,
(¬ tr_E('p1 <> p2') ∨ tr_E('p1.name <> p2.name'))))) $=^{10,13.b,17.b}$
∀ (self:Petrinet , ∀ (self:Petrinet $\overset{place}{\longrightarrow}$:Place ⊒ p1:Place ,
∀ (self:Petrinet $\overset{place}{\longrightarrow}$:Place ⊒ p2:Place ,
¬ ∃ (p1:Place p2:Place) ∨ ∃ ($\frac{p1:Place}{name = x}$ $\frac{p2:Place}{name <> x}$))))
The resulting graph constraint can to be optimized by joining all ∀-formulas:
∀ (self:Petrinet , p1:Place $\overset{place}{\longleftarrow}$ self:Petrinet $\overset{place}{\longrightarrow}$ p2:Place ,

$\neg\ \exists\ (\boxed{\text{p1:Place}} \xleftarrow{place} \boxed{\text{self:Petrinet}} \xrightarrow{place} \boxed{\text{p2:Place}})$
$\vee\ \exists\ (\boxed{\frac{p1:Place}{name = x}} \xleftarrow{place} \boxed{\text{self:Petrinet}} \xrightarrow{place} \boxed{\frac{p2:Place}{name <> x}}))$

Example 7 (Translation of constraint 9). $tr_C($ 'context Petrinet inv: self.place→exists(p:Place | p.token > 0)') $=^1$
$\forall\ (\boxed{\text{:Petrinet}},\ tr_E(\text{'self.place→exists(p:Place | p.token > 0)'})) =^{11}$
$\forall\ (\boxed{\text{:Petrinet}},\ \exists\ (tr_E(\text{'self.place'}) \sqsupseteq tr_V(\text{p:Place}),\ tr_E(\text{'p.token > 0'}))) =^{18,19,16}$
$\forall\ (\boxed{\text{:Petrinet}},\ \exists\ (tr_V(\text{'self:Petrinet'}) \xrightarrow{place} \boxed{\text{:Place}} \sqsupseteq \boxed{\text{p:Place}},\ \exists\ (\boxed{\frac{p:Place}{token > 0}}))) =^{19}$
$\forall\ (\boxed{\text{:Petrinet}},\ \exists\ (\boxed{\text{self:Petrinet}} \xrightarrow{place} \boxed{\text{:Place}} \sqsupseteq \boxed{\text{p:Place}},\ \exists\ (\boxed{\frac{p:Place}{token > 0}})))$

4.3 Correctness and completeness

To be sure that the translation of Core OCL invariants is well-defined, we show its correctness and completeness. Moreover, we want to ensure that each translation terminates.

Proposition 1 (Termination). The invariant translation tr_I as defined in Definition 13 terminates.

Proof. All OCL constraints in OclConstraint are finite. Each translation rule in Definition 13 translates at least one token (a terminal or a non-terminal symbol). In the following we show that each rule does not increase the number of tokens that have to be translated (i.e. are arguments of translation functions). Rule 1, for example, has 4 tokens to translate and keeps one to be translated. Rule 9 has 7 tokens to translate and keeps two to be translated. Hence, in each translation step the total number of tokens is not increased and at least one token is translated. It follows that all constraints of OclConstraint are fully translated after finite many steps. □

Theorem 1 (Completeness of translation). Given a core object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$, all Core OCL invariants over M are translated to some graph constraint over $ATGI$.

Proof. We have to show that all R-OCL constraints can be translated to some graph constraints. The proof is performed by induction on the structure of R-OCL constraints. First, we start with R-OCL constraints as defined in Def. 5 and continue to show the completeness of translation for all R-OCL expressions defined in Def. 4.

1. `context C inv : <expr>` is translated by rule 13.1.
2. `context v:C inv : <expr>` is translated by rule 13.2.

3. Variable expression `v` is translated by rule 13.19.
4. Operation expressions `op(`t_1`,...,`t_n`)` can be manifold:
   (a) Navigation expression `v1.r1` is translated by rule 13.18.
   (b) Boolean expressions over attribute expressions of type Integer, Real, Boolean, or String:
      i. t_1 `= v.attr` and (t_2 `= cons` or t_2 `= v`) is translated by rule 13.16.
      ii. t_1 `= v.attr` and t_2 `= v.attr2` is translated by rule 13.17.a.
      iii. t_1 `= v.attr` and t_2 `= v2.attr2` is translated by rule 13.17.b.
      iv. t_1 `= v.attr` and t_2 `= v.attr` is treated by case 13.17.c.
   (c) Boolean expression over expressions of $Expr_t$ with $t \in CLASS$:
      i. `v1 = v2` is translated by rule 13.13.a.
      ii. `v1 = v2.r2` and `v1.r1 = v2` are not allowed (see Remark 1).
      iii. `v1.r1 = v2.r2` is translated by rule 13.14.a for `v1 = v2` and 13.14.b otherwise.
      iv. `v1 <> v2` is translated by rule 13.13.b.
      v. `v1 <> v2.r2` and `v1.r1 <> v2` are not allowed (see Remark 1).
      vi. `v1.r1 <> v2.r2` is translated by rule 13.15.a for `v1 = v2` and 13.15.b otherwise.
   (d) Boolean expressions over Boolean expressions t_1, t_2, and *cond*:
      i. *true* is translated by rule 13.4.
      ii. `not` t_1 is translated by rule 13.5.
      iii. t_1 `and` t_2 is translated by rule 13.6.
      iv. t_1 `or` t_2 is translated by rule 13.6.
      v. t_1 `implies` t_2 is translated by rule 13.7.
      vi. `if` *cond* `then` t_1 `else` t_2 is translated by rule 13.8.
      vii. $t_1 \rightarrow$ `isEmpty()` is translated by rule 13.9.
      viii. $t_1 \rightarrow$ `notEmpty()` is translated by rule 13.10.
      ix. $t_1 \rightarrow$ `exists(` *var* `|` *expr*`)` is translated by rule 13.11.
      x. $t_1 \rightarrow$ `forAll(` *var* `|` *expr*`)` is translated by rule 13.12.

□

To show that the translation of Core 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 2 (Correct Translation of Core OCL invariants). Given an object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$, the following statement holds for all Core OCL invariants $inv \in \text{Invariant}_M$: For all environments $env = (\sigma, \beta) \in Env$

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

Proof.

1. con = 'context' v:C 'inv:' $expr$: We assume
   $I[\![expr]\!](env) = true \iff G = corr_{state}(\sigma) \models tr_E(expr)$
   $I[\![con]\!](env) = I[\![\text{'context } 'v' :' C' \text{ inv:' expr}]\!](env) =$
   $I[\![\text{C.allInstances()} \ ' \rightarrow \text{forAll}'(expr)]\!](env) = true$. This is equivalent to
   $G = corr_{state}(\sigma) \models \forall(\boxed{\text{v:C}}, tr_E(expr)) \iff G \models tr_C(con)$
2. con = 'context' C 'inv:' $expr$: Analogous with v = ' self '.

□

Lemma 2 (Translation of Variables). Given a restricted object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$, the following statement holds for all variables $v \in$ Variable: For all environments $env = (\sigma, \beta) \in Env$ and their corresponding graphs $G = corr_{state}(\sigma)$,

$$I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v)) \iff \exists \text{ inj. mapping } f_V : \{tr_V(v)\} \rightarrow G_V$$

with $t(v) \in CLASS$ and $corr_{CLASS}(t(v)) = type_V^G(f_V(tr_V(v)))$ with $type^G : G \rightarrow TG$.

Proof. $I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v))$. Due to tr_V, this is equivalent to the existence of a node $\boxed{\text{v:t(v)}}$ in the corresponding graph constraint and to the existence of a node mapping that maps $\boxed{\text{v:t(v)}}$ to the node set G_V of G such that $corr_{CLASS}(t(v)) = type_V^G(f_V(tr_V(v)))$. The node mapping exists due to $\beta(v) \in \sigma_{CLASS}(t(v))$. □

Lemma 3 (Translation of Navigation Expressions). Given a restricted object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$, the following statement holds for all navigation expressions $expr \in$ PropertyCallExp: For all environments $env = (\sigma, \beta) \in Env$ and their corresponding graphs $G = corr_{state}(\sigma)$,

$$I[\![expr]\!](env) = O \subseteq \sigma_{CLASS}(t(expr)) \iff \exists \text{ inj. morphisms } g_i : tr_N(expr) \rightarrow G$$

with $t(expr) \in CLASS$ and $1 \leq i \leq |O|$.

Proof. By induction on the set of translation rules for tr_N:

1. $expr = v'.'r$: We assume
   r to be a role of association as, i.e. $roles(as) = (r, r2)$ (w.l.o.g.) and
   $I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v)) \iff \exists$ inj. mapping $f_V : tr_V(v) \rightarrow G_V$.
   Furthermore, let $tr_N(v'.'r) = N$ be an attributed graph with
   $N_V = \{tr_V(v), n2\}, N_D = N_A = \emptyset, N_E = \{e\}, src_{NE}(e) = tr_V(v), trg_{NE}(e) = n2$, and $src_{NA} = trg_{NA} = \emptyset$. Furthermore, let $type : N \rightarrow ATGI$ be defined by: $type_V^N(tr_V(v)) = corr_{CLASS}(c1)$, $type_V^N(n2) = corr_{CLASS}(c2)$, and $type_E^N(e) = corr_{ASSOC}(as)$.

The equivalence is shown as follows: $I[\![expr]\!](env) = I[\![v'.'r]\!](env) = I(r_{(as,r')} : r' \to Set(r))(I[\![v]\!](env)) = O$ with $l_i = (\beta(v), o_i) \in \sigma_{ASSOC}(as)$ for all $o_i \in O$.
This is equivalent to the existence of $|O|$ inj. typed attributed morphisms $g_i : tr_N(expr) \to G$ with $1 \leq i \leq |O|$ defined by:
$g_{iV}(tr_V(v)) = f_V(tr_V(v))$, $g_{iV}(n2) = c_{CLASS}(o_i)$, and $g_{iE}(e) = l_i$.
Furthermore, we have $type^G(g_i(N)) = type^N(N)$ by
$type^G_V(g_{iV}(tr_V(v))) = type^G_V(f_V(tr_V(v))) = corr_{CLASS}(t(v)) = type^N_V(tr_V(v))$,
$type^G_V(g_{iV}(n2)) = type^G_V(c_{CLASS}(o_i)) = corr_{CLASS}(c2) = type^N_V(n2)$,
and $type^G_E(g_{iE}(e)) = type^G_E(c_{ASSOC}(l_i)) = corr_{ASSOC}(as) = type^N_E(e)$.

2. $expr = v'.'attr$: We assume $attr$ to be an attribute on $t(v)$ and
$I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v)) \iff \exists$ inj. mapping $f_V : tr_V(v) \to G_V$.
Furthermore, let $tr_N(v'.'attr) = N$ be an attributed graph with
$N_V = \{tr_V(v)\}$, $N_D = T_{DSIG}(\{x\})$ being the term algebra with variable x, $N_A = \{attr_N\}$, $N_E = \emptyset$, $src_{NE} = trg_{NE} = \emptyset$, $src_{NA}(attr_N) = tr_V(v)$, and $trg_{NA}(attr_N) = x$. Furthermore, let $type : N \to ATGI$ be defined by:
$type^N_V(tr_V(v)) = corr_{CLASS}(c1)$ and $type^N_A(attr_N) = corr_{ATT}(attr)$.
The equivalence is shown as follows:
$I[\![expr]\!](env) = I[\![v'.'attr]\!](env) =$
$I(attr : t(v) \to t(attr))(I[\![v]\!](env))(env) = \sigma_{ATT}(attr)(\beta(v)) = \beta(x) = d$
This is equivalent to the existence of an injective typed attributed morphism $g : tr_N(expr) \to G$ defined by: $g_V(tr_V(v)) = f_V(tr_V(v))$, and $g_A(attr_N) = attr_G$ with $src_{GA}(attr_G) = f_V(tr_V(v))$ assuming that $attr_G \in G_A$ exists.
Furthermore, we have $type_G(g_i(N)) = type_N(N)$ by
$type^G_V(g_V(tr_V(v))) = type^G_V(f_V(tr_V(v))) = corr_{CLASS}(t(v)) =$
$type^N_V(tr_V(v))$ and
$type^G_A(g_A(attr_N)) = type^G_A(c_{ATT}(\sigma_{ATT}(att)(f_V(tr_V(v))))) = corr_{ATT}(attr) =$
$type^N_A(attr_N)$.

□

Lemma 4 (Translation of Boolean Expressions). Given a restricted object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$, the following statement holds for all Boolean expressions $expr \in Expr_{Boolean}$: For all environments $env = (\sigma, \beta) \in Env$

$$I[\![expr]\!](env) = true \iff G = corr_{state}(\sigma) \models tr_E(expr).$$

Proof. By induction on the set of translation rules for tr_E starting from Rule 13.4. Hence, the number in the following proof corresponds with the rule number in Definition 13 decreased by 3:

1. $expr = true$: We assume that $I[\![true]\!](env) = true$. Since every graph satisfies $true$, we have $G \models true$. If vice versa, $G \models true$, we have to evaluate $I[\![true]\!](env)$ which is always equal to $true$, independently of the environment env.

2. $expr$ ='not' $expr2$: We assume
   $I[\![expr2]\!](env) = true \iff G = corr_{state}(\sigma) \models tr_E(expr2)$.

   $I[\![expr]\!](env) = I[\![$ 'not' $expr2]\!](env) = \neg(I[\![expr2]\!](env)) = true \iff$
   $I[\![expr2]\!](env) = false$. This is equivalent with $G \not\models tr_E(expr2) \iff$
   $G \models \neg(tr_E(expr2)) \iff G \models tr_E(expr)$.
3. (a) $expr$ = '(' $expr1$ 'or' $expr2$ ')': We assume
   $I[\![expr1]\!](env) = true \iff G = corr_{state}(\sigma) \models tr_E(expr1)$ and
   $I[\![expr2]\!](env) = true \iff G = corr_{state}(\sigma) \models tr_E(expr2)$.

   $I[\![expr]\!](env) = I[\![$'(' expr1 'or' expr2 ')'$]\!](env) =$
   $(I[\![expr1]\!](env)) \vee (I[\![expr2]\!](env)) = true \iff$
   $I[\![expr1]\!](env) = true$ or $I[\![expr2]\!](env) = true$. This is equivalent with
   $G \models tr_E(expr1) \vee G \models tr_E(expr2) \iff$
   $G \models (tr_E(expr1) \vee tr_E(expr2)) \iff G \models tr_E(expr)$.
   (b) $expr$ = '(' $expr1$ 'and' $expr2$ ')' = 'not (not' $expr1$ 'or not' $expr2$ ')'
   $G \models tr_E(expr)$ since cases 2 and 3 can be combined.
4. $expr$ = '(' expr1 'implies' expr2 ')' = '(not' expr1 'or' expr2 ')'
   $G \models tr_E(expr)$ since cases 2 and 3 can be combined.
5. $expr$ = '(' 'if' cond 'then' expr1 'else' expr2')' = '((' cond 'and' expr1 ') or
   (not' cond 'and' expr2 '))'
   $G \models tr_E(expr)$ since cases 2, 3, and 4 can be combined.
6. $expr = navExpr$ '→isEmpty()': We assume
   $I[\![navExpr]\!](env) = O \subseteq \sigma_{CLASS}(t(navExpr)) \iff$
   $\exists f : tr_N(navExpr) \to G$ with $t(expr) \in CLASS$.

   $I[\![expr]\!](env) = I[\![navExpr\ ' \to$ isEmpty()$']\!](env) =$
   $(I[\![navExpr]\!](env) = \emptyset) = (O = \emptyset)$. This is equivalent with
   $\neg\exists f : tr_N(navExpr) \to G \iff G \models \neg\exists(tr_N(navExpr)) \iff$
   $G \models tr_E(expr)$.
7. $expr = navExpr$ '→notEmpty()': We assume
   $I[\![navExpr]\!](env) = O \subseteq \sigma_{CLASS}(t(navExpr)) \iff$
   $\exists f : tr_N(navExpr) \to G$ with $t(navExpr) \in CLASS$.

   $I[\![expr]\!](env) = I[\![navExpr\ ' \to$ notEmpty()$']\!](env) =$
   $(I[\![navExpr]\!](env) \neq \emptyset) = (O \neq \emptyset)$. This is equivalent with
   $\exists f : tr_N(navExpr) \to G \iff G \models \exists(tr_N(navExpr)) \iff G \models$
   $tr_E(expr)$.
8. $expr = navExpr$ '→exists (' v '|' $expr2$ ')': We assume
   $I[\![navExpr]\!](env) = O \subseteq \sigma_{CLASS}(t(navExpr)) \iff$
   $\exists f : tr_N(navExpr) \to G$ and
   $I[\![expr2]\!](\sigma, \beta\{v|o\}) = true$ for $o \in O \iff$
   $G = corr_{state}(\sigma) \models \exists(G(tr_V(v)), tr_E(expr2))$.[23]

[23] If x is a vertex, $G(x)$ is the smallest graph having x as vertex.

$I[\![expr]\!](env) = I[\![navExpr' \rightarrow exists('v' \mid' expr2')']\!](env) = true \iff$
$\bigvee_{x \in [\![navExpr]\!](env)} I[\![expr2]\!](\sigma, \beta\{v|x\}) = true \iff$
$\bigvee_{v \in tr_N(navExpr)} Shift_{\sqsupseteq}(G(tr_V(v)) \rightarrow tr_N(navExpr), tr_E(expr)) \iff$
$G \models \exists(tr_N(navExpr) \sqsupseteq tr_V(v), \exists(tr_E(expr))) \iff$
$G \models tr_E(expr)$.

9. $expr = navExpr$ '→forAll (' v '|' $expr2$ ')': Analogously to previous case.
10. (a) $expr = v1' =' v2$: We assume
$I[\![v1]\!](env) = \beta(v1) \in \sigma_{CLASS}(t(v1)) \iff \exists f1_V : \{tr_V(v1)\} \rightarrow G_V$
and
$I[\![v2]\!](env) = \beta(v2) \in \sigma_{CLASS}(t(v2)) \iff \exists f2_V : \{tr_V(v2)\} \rightarrow G_V$
with $t(v1) = t(v2)$.

$I[\![expr]\!](env) = I[\![v1' =' v2]\!](env) = I[\![v1]\!](env) = I[\![v2]\!](env) =$
$\beta(v1) = \beta(v2) = true \iff$
$\exists$ mappings $f1_V : \{tr_V(v1)\} \rightarrow G_V$ and $f2_V : \{tr_V(v2)\} \rightarrow G_V$ such that $f1_V = f2_V \iff$
$\exists$ mappings $f1_V : \{\boxed{\text{v1:t(v1)}}\} \rightarrow G_V$ and $f2_V : \{\boxed{\text{v2:t(v2)}}\} \rightarrow G_V$ such that $f1_V = f2_V \iff$
$\exists$ a mapping $f_V : \{\boxed{\text{v1:t(v1)}}, \boxed{\text{v2:t(v2)}}\} \rightarrow G$ with $f_V(v1) = f_V(v2)$
$\iff$
$G \models \exists(\boxed{\text{v1:t(v1)}}\ \boxed{\text{v2:t(v2)}} \rightarrow \boxed{\text{v1,v2:t(v1)}}) \iff G \models tr_E(expr)$.

(b) $expr = v1' <>' v2$: We assume
$I[\![v1]\!](env) = \beta(v1) \in \sigma_{CLASS}(t(v1)) \iff \exists f1_V : \{tr_V(v1)\} \rightarrow G_V$
and
$I[\![v2]\!](env) = \beta(v2) \in \sigma_{CLASS}(t(v2)) \iff \exists f2_V : \{tr_V(v2)\} \rightarrow G_V$
with $t(v1) = t(v2)$.

$I[\![expr]\!](env) = I[\![v1' <>' v2]\!](env) = I[\![v1]\!](env) <> I[\![v2]\!](env) =$
$\beta(v1) \neq \beta(v2) = true \iff$
$\exists$ mappings $f1_V : \{tr_V(v1)\} \rightarrow G_V$ and $f2_V : \{tr_V(v2)\} \rightarrow G_V$ such that $f1_V \neq f2_V \iff$
$\exists$ mappings $f1_V : \{\boxed{\text{v1:t(v1)}}\} \rightarrow G_V$ and $f2_V : \{\boxed{\text{v2:t(v2)}}\} \rightarrow G_V$ such that $f1_V \neq f2_V \iff$
$\exists$ an injective mapping $f_V : \{\boxed{\text{v1:t(v1)}}, \boxed{\text{v2:t(v2)}}\} \rightarrow G \iff$
$G \models \exists(\boxed{\text{v1:t(v1)}}\ \boxed{\text{v2:t(v2)}}) \iff G \models tr_E(expr)$.

11. We prove the two rules in Definition 13.11 in inverse order.
(a) $expr = (v1.r1' =' v2.r2)$: We assume
$I[\![v1]\!](env) = \beta(v1) \in \sigma_{CLASS}(t(v1)) \iff \exists f1_V : \{tr_V(v1)\} \rightarrow G_V$
and
$I[\![v2]\!](env) = \beta(v2) \in \sigma_{CLASS}(t(v2)) \iff \exists f2_V : \{tr_V(v2)\} \rightarrow G_V$.

$I[\![expr]\!](env) = I[\![v1.r1 = v2.r2]\!](env) = (I[\![v1.r1]\!](env) = I[\![v2.r2]\!](env)) =$
$(I(r_{(as1,r1')} : r1' \rightarrow Set(r1))(I[\![v1]\!](env)) = O =$
$I(r_{(as2,r2')} : r2' \rightarrow Set(r2))(I[\![v2]\!](env))) = true$ with

$(\beta(v1), o) \in \sigma_{ASSOC}(a1) \iff (\beta(v2), o) \in \sigma_{ASSOC}(a2)$ for all $o \in O$.
This is equivalent to
$\exists$ morphisms $g1_i : tr_N(v1.r1) \to G$ and $g2_i : tr_N(v2.r2) \to G$
with $1 \leq i \leq |O|$ such that $g1_i(o) = g2_i(o)$ $\forall o \in O \iff$
$\exists$ an injective morphism
$f : \{tr_V(v1\ ':'\ t(v1)) \xrightarrow{a1} \boxed{:\text{t(r1)}} \xleftarrow{a2} tr_V(v2\ ':'t(v2))\} \to G \iff$
$G \models \exists\ (tr_V(v1\ ':'\ t(v1))) \xrightarrow{a1} \boxed{:\text{t(r1)}} \xleftarrow{a2} tr_V(v2\ ':'\ t(v2)) \iff$
$G \models tr_E(expr)$.

(b) $expr = (v.r1' =' v.r2)$:
Analogous to the previous sub-case with $v = v1 = v2$.

12. Analogous to the previous case (with sub-cases) replacing "=" by "<>".

(a) $expr = (v1.r1 <> v2.r2)$: Especially, $\exists$ morphisms $g1_i : tr_N(v1.r1) \to G$ and $g2_i : tr_N(v2.r2) \to G$ with $1 \leq i \leq |O|$ such that $g1_i(o) <> g2_i(o)$ for some $o \in O \iff \exists$ an injective morphism $f : \{tr_V(v1\ ':'\ t(v1)) \xrightarrow{a1} \boxed{:\text{t(r1)}}\ tr_V(v2\ ':'t(v2)) \xrightarrow{a2} \boxed{:\text{t(r2)}}\} \to G$.

(b) $expr = (v.r1 <> v.r2)$:
Analogous to the previous sub-case with $v = v1 = v2$.

13. $expr = v\ '.'\ attr\ op\ x$: We assume
$I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v)) \iff \exists$ inj. $f_V : \{tr_V(v)\} \to G_V$

$I[\![expr]\!](env) = I[\![v'.'attr\ op\ x]\!](env) =$
$I(op)(env)(I[\![v'.'attr]\!](env), I(x)(env)) =$
$I(op)(env)(I(attr : t(v) \to t(attr))(I[\![v]\!](env)), x)(env) =$
Here, we have two sub-cases:
(a) x is a constant: $I(op)(env)(\sigma_{ATT}(attr)(\beta(v)), x) = true$
(b) x is a variable: $I(op)(env)(\sigma_{ATT}(attr)(\beta(v)), \beta(x)) = true$
This is equivalent to
$\exists$ injective morphism $g : tr_N(v'.'attr) \to G$ and there is an attribute condition $op(y, x) = true$ that is satisfied by graph G with
$tr_N(v'.'attr) = \boxed{\frac{v:t(v)}{attr = y}} \iff G \models \exists\ (\boxed{\frac{v:t(v)}{attr\ op\ x}}) \iff G \models tr_E(expr)$.

14. (a) $expr ::= v\ '.'\ attr\ op\ v\ '.'\ attr2$: We assume attr $\neq$ attr2 and
$I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v)) \iff \exists$ inj. $f_V : \{tr_V(v)\} \to G_V$.

$I[\![expr]\!](env) = I[\![v'.'attr\ op\ v'.'attr2]\!](env) =$
$I(op)(env)(I[\![v'.'attr]\!](env), I[\![v'.'attr2]\!](env)) =$
$I(op)(env)(I(attr : t(v) \to t(attr))(I[\![v]\!](env))(env),$
$\qquad I(attr2 : t(v) \to t(attr2))(I[\![v]\!](env))(env)) =$
$I(op)(env)(\sigma_{ATT}(attr)(\beta(v)), \sigma_{ATT}(attr2)(\beta(v))) = true$

Due to the assumptions, this is equivalent to
$\exists$ injective $g : tr_N(v'.'attr) \to G$ and $h : tr_N(v'.'attr2) \to G$ with
$g(tr_N(v'.'attr)) \neq h(tr_N(v'.'attr2))$ and there is an attribute condition
$op(y, x) = true$ that is satisfied by graph G

with $tr_N(v'.'attr) = \boxed{\frac{v:t(v)}{attr = x}}$ and $tr_N(v'.'attr2) = \boxed{\frac{v:t(v)}{attr2 = y}} \iff$
$G \models \exists \left(\boxed{\begin{array}{c} v:t(v) \\ \hline attr \ = \ x \\ attr2 \ op \ x \end{array}} \right) \iff G \models tr_E(expr).$

(b) $expr ::= v$ '.' $attr$ op $v2$ '.' $attr2$: We assume
$I[\![v]\!](env) = \beta(v) \in \sigma_{CLASS}(t(v)) \iff \exists$ inj. $f1_V : \{tr_V(v)\} \to G_V$
$I[\![v2]\!](env) = \beta(v2) \in \sigma_{CLASS}(t(v2)) \iff$
$\exists$ inj. $f2_V : \{tr_V(v2)\} \to G_V$
with $v \neq v2$ and $f1_V(tr_V(v)) \neq f2_V(tr_V(v2))$.

$I[\![expr]\!](env) = I[\![v'.'attr \ op \ v2'.'attr2]\!](env) =$
$I(op)(env)(I[\![v'.'attr]\!](env), I[\![v2'.'attr2]\!](env)) =$
$I(op)(env)(I(attr : t(v) \to t(attr))(I[\![v]\!](env))(env),$
$I(attr2 : t(v2) \to t(attr2))(I[\![v2]\!](env))(env)) =$
$I(op)(env)(\sigma_{ATT}(attr)(\beta(v)), \sigma_{ATT}(attr2)(\beta(v2))) = true$

Due to the assumptions, this is equivalent to
$\exists$ injective morphisms $g : tr_N(v) \to G$ and $h : tr_N(v2) \to G$
with $g(tr_V(v)) \neq h(tr_V(v2))$ and there is an attribute condition
$op(y, x) = true$ that is satisfied by graph G. Moreover,
$tr_N(v'.'attr) = \boxed{\frac{v:t(v)}{attr = x}}$ and $tr_N(v2'.'attr2) = \boxed{\frac{v:t(v2)}{attr2 = y}} \iff$
$G \models \exists \left(\boxed{\frac{v:t(v)}{attr \ op \ x}} \ \boxed{\frac{v:t(v2)}{attr2 \ op \ y}} \right) \iff G \models tr_E(expr).$

□

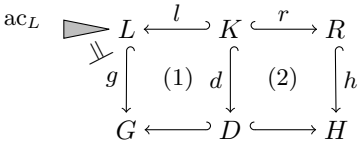
5 From Core OCL invariants to Application Conditions

After having translated Core OCL invariants to graph constraints, we connect this new result with the existing theory on graph constraints [3,19]. 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 Core OCL invariant is translated to a left application condition.

We recall the definition of graph transformation with injective rule morphisms $l\colon K \hookrightarrow L$ and $r\colon K \hookrightarrow R$, a left application condition ac_L, and injective matches $g\colon L \hookrightarrow G$.

Definition 14 (rules and transformations). A *rule* $\varrho = \langle p, \mathrm{ac}_L \rangle$ consists of a *plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ with injective morphisms $l\colon K \hookrightarrow L$ and $r\colon K \hookrightarrow R$, 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 $G \Rightarrow_{\varrho,g} H$ if

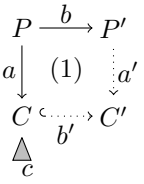
there exists such a direct transformation.



The first result says that $ATGI$-conditions can be shifted over $ATGI$-morphisms.

Lemma 5 (shift of $ATGI$-conditions over $ATGI$-morphisms). There is a Shift-construction such that, for each $ATGI$-condition c over P and for each $ATGI$-morphism $b\colon P \to P'$, Shift transforms c via b into an $ATGI$-condition $\mathrm{Shift}(b,c)$ over P' such that, for each $ATGI$-morphism $n\colon P' \to H$, $n \circ b \models c \iff n \models \mathrm{Shift}(b,c)$.

Construction ([17]). The Shift-construction is inductively defined as follows:[24]

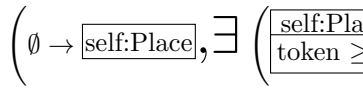


$\mathrm{Shift}(b, true) = true$.
$\mathrm{Shift}(b, \exists(a,c)) = \bigvee_{(a',b')\in\mathcal{F}} \exists(a', \mathrm{Shift}(b',c))$
if the set $\mathcal{F}$ of jointly surjective pairs (a',b') such that b' is injective and (1) commutes is non-empty and false, otherwise.
$\mathrm{Shift}(b, \neg c) = \neg\mathrm{Shift}(b,c)$ and $\mathrm{Shift}(b, \wedge_{i\in J} c_i) = \wedge_{i\in J}\mathrm{Shift}(b,c_i)$.

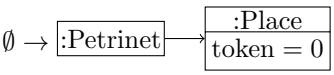
Conditions of the form $\exists(P \sqsupseteq C, c)$ can be expressed in terms of the other constructs (see Lemma 1). Therefore, we don't have to define $\mathrm{Shift}(b, \exists(P \sqsupseteq C, c))$.

Proof. Immediate consequence of Lemma 2 in [17] using the fact that the category in consideration has $\mathcal{E}'$-$\mathcal{M}$ pair factorizations. □

Example 8. We shift the graph constraint in Example 3 corresponding to the OCL constraint 2

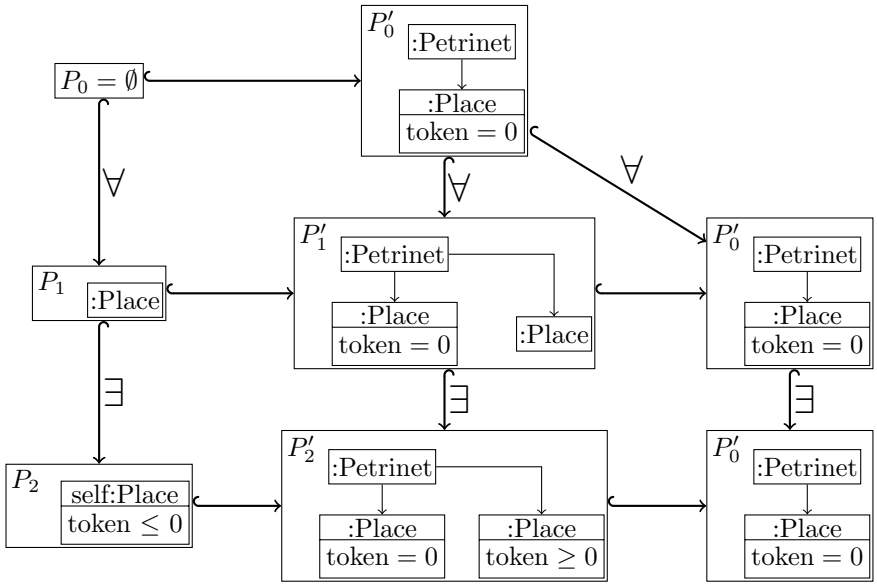


saying that "The number of tokens on a place is not negative" over the morphism

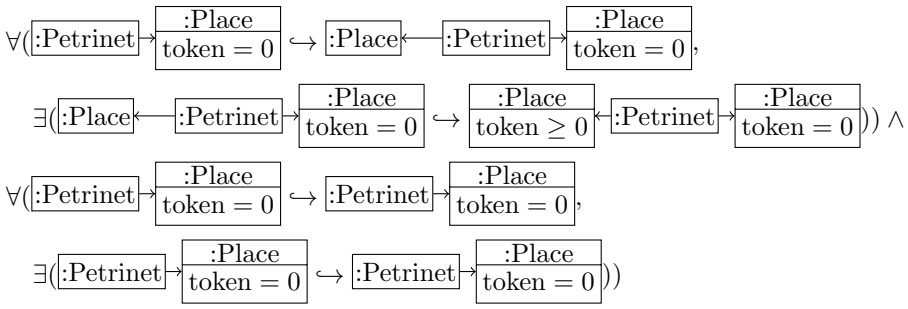


The construction proceeds as shown below. Quantors are written next to the morphism arrows. The original constraint can be seen on the left side, and the new condition is in the middle and right part.

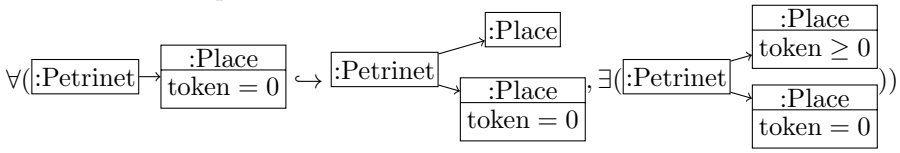
[24] A pair of morphisms $P' \to C' \leftarrow C$ is *jointly surjective* if each item in C' has a preimage in P' or C.



The Shift construction yields the condition



which can be simplified to

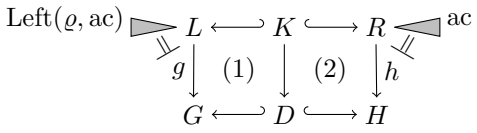


stating that "The Place object has zero token count and any other Place object has a non-negative token count."

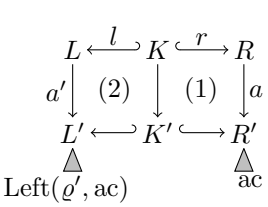
The second result is that $ATGI$-conditions can be shifted over rules.

Lemma 6 (shift of $ATGI$-conditions over $ATGI$-rules [3]). There is a construction Left such that, for each $ATGI$-rule $\varrho = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and each $ATGI$-condition ac over R, Left transforms ac via ϱ into an $ATGI$-condition

$\mathrm{Left}(\varrho, \mathrm{ac})$ over L such that, for each direct transformation $G \Rightarrow_{\varrho,m,m^*} H$, we have $g \models \mathrm{Left}(\varrho, \mathrm{ac}) \iff h \models \mathrm{ac}$.



Construction. The construction Left is inductively defined as follows:



$\mathrm{Left}(\varrho, \mathit{true}) = \mathit{true}$.
$\mathrm{Left}(\varrho, \exists(a, \mathrm{ac})) = \exists(a', \mathrm{Left}(\varrho', \mathrm{ac}))$ if $R' \Rightarrow_{\varrho^{-1},a,a'} L'$ is a direct transformation and $\varrho' = \langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ is the *derived* rule derived of $L' \Rightarrow_{\varrho,a',a} R'$ and false, otherwise.
$\mathrm{Left}(\varrho, \neg \mathrm{ac}) = \neg \mathrm{Left}(\varrho, \mathrm{ac})$ and
$\mathrm{Left}(\varrho, \wedge_{i \in J} \mathrm{ac}_i) = \wedge_{i \in J} \mathrm{Left}(\varrho, \mathrm{ac}_i)$.

By the argumentation above, we don't have to define $\mathrm{Left}(\varrho, \exists(P \sqsupseteq C, c))$.

Proof. Immediate consequence of Theorem 6 in [3] using Fact 1. □

We present a simple rule to create places in a Petri net. The graph constraint from Example 2 shall be translated to a left application condition. The conditions are given in abbreviated form (i.e. whenever it is unambiguous, only the codomain of a morphism is shown); node mappings are obvious from their relative position. Edge labels are omitted for brevity.

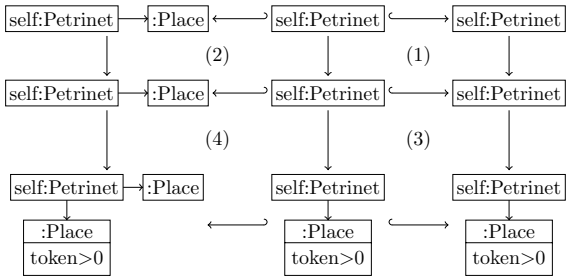
Example 9. Consider the rule $\langle$[:Petrinet] → [:Place] ↩ [:Petrinet] ↪ [:Petrinet]$\rangle$ and the right application condition

$$\forall \left(\boxed{\text{self:Petrinet}}, \exists \left(\boxed{\text{self:Petrinet}} \rightarrow \boxed{\begin{array}{c}\text{:Place}\\ \text{token>0}\end{array}} \right) \right) \equiv \exists \left(\boxed{\text{self:Petrinet}} \rightarrow \boxed{\begin{array}{c}\text{:Place}\\ \text{token>0}\end{array}} \right)$$

Shifting this over the rule with Left, we obtain the left application condition

$$\forall \left(\boxed{\text{self:Petrinet}} \rightarrow \boxed{\text{:Place}}, \exists \left(\boxed{\begin{array}{c}\text{:Place}\\ \text{token>0}\end{array}} \leftarrow \boxed{\text{self:Petrinet}} \rightarrow \boxed{\text{:Place}} \right) \right).$$

saying that "there is a Place object with a connection from the Petri net and at least one token". The detailed construction is given below.



6 Related Work and Conclusion

In the literature, there are several significant approaches to define a formal semantics for OCL. The motivations for a formal OCL semantics are manifold and include defining a clear semantics, generating model instances, and performing formal verification of UML/OCL models. All main approaches are logic-oriented, in contrast to ours being the first one that relates OCL to a graph-based approach. In the following, we sketch logic-oriented approaches using the Key prover, the Alloy project, and Constraint Logic Programming, respectively.

In [20], Beckert et al. present a translation of UML class diagrams with OCL constraints into first-order logic; the goal is logical reasoning about UML models. The translation has been implemented as a part of the KeY system, but can also be used stand-alone. Formal methods such as Alloy [21] can be used for instance generation: 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. In [22], UML models are automatically transformed to corresponding Alloy representations. Alloy models can then be analyzed automatically, with the help of the Alloy Analyzer. A recent work translating OCL to relational logic is presented in [23] covering more features than UML2Alloy. The USE tool [6,24] can be used for generating snapshots that conform to the model or for checking the conformity of a specific instance. In [25], Cabot et al. present UMLtoCSP, a tool that is able to automatically check correctness properties of UML class model with OCL constraints based on Constraint Logic Programming.

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. Hence, the reasoning is performed on the level of model elements. Translating OCL invariants to graph constraints allows to keep graph structures as units of abstraction while checking for satisfiability. Pennemann has shown in [19] 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. Furthermore, a translation of OCL to graph constraints yields a new visualization of OCL which can help understanding. And finally, our translation offers a way to translate Core OCL invariants to application conditions of transformation rules. This is a new form to apply an OCL translation which might lead to number of new applications including test model generation as well as auto-completion of model editing operations. The backward translation from graph conditions to OCL might also be interesting to come up with model transformation rules restricted by OCL. In future work, we plan to extend this work towards the whole range of OCL invariants being translated to more powerful graph conditions.

Acknowledgement. We are grateful to Christoph Peuser and 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. 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
3. Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. Mathematical Structures in Computer Science **19** (2009) 245–296
4. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Software and System Modeling **11**(2) (2012) 227–250
5. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a Widespread Use of OCL. In: Workshop on Tool Support for OCL and Related Formalisms, Technical Report LGL-REPORT-2005-001, EPFL (2005) 68–82
6. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin (2002)
7. 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
8. Hennicker, R., Hussmann, H., Bidoit, M.: On the Precise Meaning of OCL Constraints. In Clark, T., Warmer, J., eds.: Object Modeling with the OCL. Volume 2263 of LNCS. Springer Berlin Heidelberg (2002) 69–84
9. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a Widespread Use of OCL. In: Workshop on Tool Support for OCL and Related Formalisms, Technical Report LGL-REPORT-2005-001. (2005) 68–82
10. OMG: Object Constraint Language. http://www.omg.org/spec/OCL/2.2/
11. 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
12. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science. Springer (1985)
13. Loeckx, J., Ehrich, H.D., Wolf, M.: Specification of abstract data types. Wiley (1996)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. Springer (2006)
15. Orejas, F.: Symbolic Graphs for Attributed Graph Constraints. J. Symb. Comput. **46**(3) (2011) 294–315
16. Poskitt, C.M., Plump, D.: Hoare-Style Verification of Graph Programs. Fundamenta Informaticae **118(1-2)** (2012) 135–175
17. 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. Mathematical Structures in Computer Science **24** (2014)
18. 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)
19. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, Universität Oldenburg (2009)

20. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into First-order Predicate Logic. In: VERIFY, Workshop at Federated Logic Conferences (FLoC). (2002)
21. Jackson, D.: Alloy Analyzer website (2012) `http://alloy.mit.edu/`.
22. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling **9**(1) (2010) 69–86
23. 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
24. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. SoSyM **4**(4) (2009) 386–398
25. 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