Translating Essential OCL Invariants to Nested Graph Constraints for Generating Instances of Meta-models: Long Version

Hendrik Radke\textsuperscript{b,}\textsuperscript{*}, Thorsten Arendt\textsuperscript{a}, Jan Steffen Becker\textsuperscript{b}, Annegret Habel\textsuperscript{b}, Gabriele Taentzer\textsuperscript{c}

\textsuperscript{a}GFFT Innovationsförderung GmbH, Bad Vibel, Germany
\textsuperscript{b}Universität Oldenburg, Germany
\textsuperscript{c}Philipps-Universität Marburg, Germany

Abstract

Domain-specific modeling languages (DSMLs) are usually defined by meta-modeling using the Object Constraint Language (OCL) for specifying invariants. This approach is purely declarative in the sense that instance construction is not supported. In contrast, grammar-based language definition incorporates the stepwise construction of instances by applying production rules. Since the underlying structure of models are generally graphs, graph grammars are well suited to define modeling languages. Establishing a formal relation between meta-modeling and graph grammars opens up the possibility to integrate techniques of both fields. This integration can be advantageously used for optimizing DSML definition. We follow an approach where a meta-model is translated to a type graph with a set of nested graph constraints. While previous meta-model translations neglected OCL constraints, we focus on the translation of Essential OCL invariants to nested graph constraints in this paper. We show that a model satisfies an Essential OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint. In addition, nested graph constraints can be translated to application conditions of graph transformation rules. Composing both translations, an instance-generating graph grammar can be equipped with application conditions such that it generates instances of the original meta-model only.

Keywords: Meta modeling, Essential OCL, graph constraints, instance generation

Contents

1 Introduction 2

2 Informal introduction to the translation of OCL invariants 3

2.1 Running example 4

2.2 Type and instance graphs 5

2.3 Translating OCL invariants to nested graph constraints 5

2.4 Instance-creating transformation rules 7

2.5 Translating graph constraints to application conditions of grammar rules 8

2.6 Generating meta-model instances 9

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

\*Corresponding Author

Email addresses: radke@informatik.uni-oldenburg.de (Hendrik Radke), thorsten.arendt@gfft-ev.de (Thorsten Arendt), jan.steffen.becker@informatik.uni-oldenburg.de (Jan Steffen Becker), habel@informatik.uni-oldenburg.de (Annegret Habel), taentzer@informatik.uni-marburg.de (Gabriele Taentzer)

Preprint submitted to Elsevier February 12, 2018
1. Introduction

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

Given a meta-model, instance generation has been considered by several approaches in the literature. Most of them are logic-oriented as, e.g., [2] [3]. They translate class models with OCL constraints into logical facts and formulas. Logic approaches such as Alloy [4] can be used for instance generation, as done, e.g., in [7]: 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 possible instances. All these approaches have in common that they translate class models with OCL constraints into logical facts and formulas, not exploring the graph properties of underlying model structures.

Alternatively, graph grammars have shown to be suitable and natural to specify (domain-specific) visual languages in a constructive way [5]. Similarly to string grammars, all language instances are generated from
a start graph by applying rules consecutively. This instance generation process is optimized w.r.t. graph properties.

In this article, we translate OCL constraints to graph patterns or graph constraints and therefore bridge between both approaches. To formally treat models and meta-models (without OCL constraints) they are translated to instance and type graphs. Hence, we follow the graph-based approach keeping the graph structure of models as units of abstraction where graph axioms are satisfied by default. In [6], we started to formally translate OCL constraints to nested graph constraints [7]. In [8], this translation is extended to set operations such as select, collect, union and size. Meanwhile, Bergmann [9] has implemented a translator of OCL constraints to graph patterns. The focus of that work, however, is not a formal translation but an efficient implementation of constraint checking.

Our main intention for using such an OCL translation is to generate instances of meta-models. Following the ideas by Ehrig et al. in [10], a meta-model can be translated into a graph grammar defining an equivalent language. Taking such a graph grammar, graph constraints which result from the OCL translation can be further translated to application conditions of transformation rules [7]. We illustrate this approach on the generation of Petri nets.

Since graph-based approaches rely on (type and object) graphs, they support flat object sets as the only form of OCL collections to be translated to. This is sufficient for language definition where a specific order of objects as well as the distinction of duplicates are not crucial (see also [3]). Moreover, our OCL translation is formulated for a slightly simpler form of meta-models specified by EMOF [11], i.e., we focus on Essential OCL being closer to supporting technologies such as the Eclipse Modeling Framework. Furthermore, our considerations are restricted to a first-order, two-valued logic, as for graph constraints, i.e., the translation is straightened to the corresponding OCL features. Our translation comprises all language features of Essential OCL used in existing specifications, according to an empirical study conducted in [12] which identifies the set of OCL language constructs actually used in practice. Since the focus of OCL usage is DSML definition, our translation mainly concentrates on OCL invariants.

The main contribution of this paper is the following: The translation of Essential OCL invariants to nested graph constraints is shown to be correct, i.e., a model satisfies an Essential OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint. Moreover, we show that our translation is complete for the chosen set of OCL invariants. The overall aim of this work is to establish a formal relation between meta-modeling and the theory of graph transformation. It shall form a basis for integrating works in both fields in an advantageous way.

This paper is structured as follows: The next section gives an informal introduction to our approach. It introduces the running example and shows the translation of OCL to graph constraints and further to application conditions using this example. Moreover, it presents an instance-generating grammar for this example. Section 3 recalls Essential OCL. Section 4 recalls nested graph conditions. It also introduces a compact notion of graph conditions. Section 5 presents the translation of Essential OCL invariants to nested graph constraints, more precisely to compact conditions. The correctness and completeness of this translation is shown in Section 6. The translation of graph constraints to application conditions of rules is sketched in Section 7. Section 8 contains a comparison to related work and Section 9 concludes the paper.

2. Informal introduction to the translation of OCL invariants

This section gives an informal overview of our approach: We show how a given meta-model is translated to an instance-generating graph grammar. While the meta-model structure (including types) is defined by the type graph and instance-generating grammar rules, multiplicities and OCL invariants are translated to graph constraints being further translated to application conditions (abbreviated by “ac” in Figure 1) of grammar rules. This augmented graph grammar can be used to generate instance graphs which conform to the given meta-model. See also Figure 1.

We illustrate our approach with Petri nets as modeling language. Note that the presentation in this section is completely informal. All formal definitions and theorems are presented later in Sections 3 to 7.
2.1. Running example

A Petri net (PetriNet) \[13\] is composed of several places (Place) and transitions (Transition). Petri nets, places and transitions are all NamedElements. Arcs between places and transitions are explicit. PTArc and TPArc are respectively representing place-to-transition arcs and transition-to-place ones. An arc is annotated with a weight. A place can have an arbitrary number of incoming (preArc) and outgoing (postArc) arcs. In order to model dynamic aspects, places need to be marked with tokens (Token). The corresponding meta-model is shown in Figure 2.

Despite of multiplicities, this meta-model allows to build inappropriate instances, e.g., place and transition names may be empty or equal. Therefore, the meta-model has to be complemented with invariants being formulated in OCL. In the following, the main invariants are formulated.

1. Two different Petri nets have distinct names.
   
   context PetriNet inv: PetriNet.allInstances() ->
   
   forall(p: PetriNet | p <> self implies p.name <> self.name)

2. The name of a place is not empty.

   context Place inv: self.name <> ''

3. The name of a transition is not empty\[1\]

   context Transition inv: self.name <> ''

\[1\]OCL constraints 2 and 3 can also be combined to one OCL constraint in the context of superclass NamedElement, i.e., the name of a named element is not empty.
4. Two different places of a Petri net have distinct names.
   context PetriNet inv:
   self.place -> forall(p1,p2:Place | p1 <> p2 implies p1.name <> p2.name))

5. Two different transitions of a Petri net have distinct names.
   context PetriNet inv:
   self.transition -> forall(t1,t2:Transition | t1 <> t2 implies t1.name <> t2.name))

6. The weight of an arc is strictly positive.
   context Arc inv: self.weight >= 1

7. There is at most one arc between a pair of place and transition.
   context PetriNet inv:
   TPArc.allInstances() ->
   forall(a1,a2:TPArc | a1 <> a2 implies (a1.src <> a2.src or a1.dst <> a2.dst))) and
   PTArc.allInstances() ->
   forall(a1,a2:PTArc | a1 <> a2 implies (a1.src <> a2.src or a1.dst <> a2.dst)))

8. There is at least one place in a Petri net having at least one token.
   context PetriNet inv: self.place -> exists(p:Place | p.token -> notEmpty())

An example Petri net is given in Figure 3. It specifies a simple producer/consumer system with a 1-buffer. This means that the producer can produce exactly one piece to be stored in the buffer. Thereafter, the consumer has to become active before the producer creates the next piece. A transition can fire if all places connected to this transition by a PTArc each have a token. In the example Petri net, produce is the only transition that can fire. After firing, there is again a token on place readyP and a token on place buffer. Hence, transition consume can fire thereafter. Actually this Petri net can continuously fire, i.e, transitions produce and consume can fire alternately.

![Figure 3: An example Petri net showing a simple producer/consumer system with a 1-buffer.](image)

2.2. Type and instance graphs

When translating a meta-model, we focus on its structure first and neglect all kinds of constraints, even multiplicities. This structural part of a meta-model is translated to a type graph in a straightforward way: The meta-model structure is just considered as a graph. From now on, we abbreviate some of the type names due to space limitations in subsequent figures: A “PetriNet” is abbreviated to “PN”, “Place” to “Pl”, “Transition” to “Tr”, and “Token” to “Tk”. Furthermore, we use bidirectional edges to clearly indicate that edges without arrows (occurring in the meta-model in Figure 2) are actually bidirectional ones. In Figure 4 we show a snippet of the abstract syntax of the example Petri net in Figure 3 as instance of the meta-model in Figure 2.

2.3. Translating OCL invariants to nested graph constraints

In the next step, we translate multiplicities and all the OCL constraints of a meta-model to nested graph constraints and hence, complete the translation of meta-models.
Simple graph constraints are concerned with the existence and non-existence of graph patterns. Graph patterns are drawn in a standard way: A node is depicted by a rectangle carrying its name followed by its type. An edge is drawn by an arrow pointing from the source to the target node and the edge label is placed next to the arrow. For instance, the upper bound of a multiplicity can be expressed by the non-existence of a pattern that shows a situation where the upper bound is exceeded by 1. If a PTArc must not have more than one source place ("Place" is abbreviated to "Pl"), a pattern consisting of a PTArc with two source edges must not exist:

$$\neg \exists (\text{Pl} \leftarrow \text{PTArc} \leftarrow \text{Pl})$$

A graph fulfills this graph constraint if none of its PTArcs has more than one source edge, i.e., there is no occurrence of this pattern in the graph.

Graph constraints may be nested, i.e., graph constraints may occur in graph constraints. Each graph in a graph constraint is equipped with a quantifier, namely the existential quantifier $$\exists$$, the universal quantifier $$\forall$$ or their negations. Each nesting increases the expressiveness of constraints. Inclusions are given by the names of the nodes: Two occurrences of $$v$$ in different graphs of the graph constraint, e.g. $$\exists (\text{v} \exists (\text{v}, c))$$ or $$\exists (\text{v} \exists (\text{v}, c))$$, mean that they are in inclusion relation. A simple example of a nested graph constraint is used to formalize lower bounds of multiplicities. For example, a PTArc must have at least one source place:

$$\forall (\exists \text{PTArc} \exists \exists (\text{Pl}))$$

A graph fulfills this graph constraint if all its PTArcs have at least one source edge. Furthermore, nested graph constraints may be connected by propositional logic. As rules, graph constraints may contain variables for attribute values. They are implicitly declared.

The translation of OCL invariants to nested graph constraints is possible if they stick to flat object sets as the only form of OCL collections. Furthermore, their semantics is restricted to first-order, two-valued logic, as the one for nested graph constraints. These restrictions do not harm if OCL is used for language definition.

In the following, we show the graph constraints to which the OCL invariants in Section 2.1 are translated. The way in which OCL invariants are translated to graph constraints is presented in Section 5.
1. Two different Petri nets have distinct names.

\[ \neg \exists (1:PN \text{name}=n \land 2:PN \text{name}=n) \]

2. There is neither a place nor a transition with an empty name.

\[ \neg (\exists 1:Pl \text{name}=\emptyset \lor \exists 1:Tr \text{name}=\emptyset) \]

3. Two different places or transitions of a Petri net have distinct names.

\[ \forall (p_1:Pl \text{place} \land p_2:Pl \text{place}, \exists (p_1:Pl \text{name}=n \land p_2:Pl \text{name} \neq n)) \land \forall (t_1:Tr \text{trans} \land t_2:Tr \text{trans}, \exists (t_1:Tr \text{name}=n \land t_2:Tr \text{name} \neq n)) \]

4. The weight of an arc is positive.

\[ \forall (1:Arc \exists \text{weight} \geq 1) \]

5. Double arcs from a transition to a place, or vice versa, are forbidden.

\[ \forall (1:TPArc \exists \text{src} = 1:Tr \land 2:TPArc \exists \text{src} = 2:Tr) \lor \exists (1:TPArc \exists \text{src} = 1:Pl \land 2:TPArc \exists \text{src} = 2:Pl) \land \forall (1:PTArc \exists \text{src} = 1:Pl \land 2:PTArc \exists \text{src} = 1:Tr) \lor \exists (1:PTArc \exists \text{src} = 1:Tr \land 2:PTArc \exists \text{src} = 1:Pl) \]

6. Every Petri net has at least one token.

\[ \forall (1:PN \exists 1:Pl \text{token}) \]

2.4. Instance-creating transformation rules

Since the main motivation for the OCL translation is to get a rule-based approach for generating meta-model instances, we consider generation rules next. In the following, we first present the rules and then translate the graph constraints shown above to left application conditions (short ACs) of these transformation rules restricting their applicability such that consistency is preserved, also w.r.t. OCL constraints. We just show the resulting application conditions. The proper translation is illustrated in Section 7.

We consider a rule set for the generation of Petri nets. All rules conform to the meta-model in Figure 2. Together with the empty instance as start graph, they form a graph grammar for the generation of Petri nets. A generated graph may represent more than one Petri net. The main idea of graph grammars and graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph change. In particular, graph grammars can be used to define graph languages. A graph belongs to the graph grammar-defined language if it can be derived from its start graph by arbitrarily many rule applications.

The core of a graph transformation rule is a pair of typed graphs, called left-hand side (LHS) and right-hand side (RHS). Roughly spoken, a rule is applied by finding a match of the LHS in the source graph and by replacing its image by a copy of the RHS leading to the target graph of the graph transformation. This means in particular that attribute values occurring in the LHS are compared with those in the source graph while attributes are set to values occurring in the RHS. In rules, attribute values may also be variables that are instantiated during rule matching and used for attribute computations during rule application. Abstract types as presented in [15] may be used in rules to formulate them on an appropriate abstraction level. All elements that are newly created, however, need to have concrete types. For further controlling the application of a rule, left ACs can be formulated that have to be fulfilled before applying the rule. For the moment, we do not consider ACs and come back to them below.
1. Creating a new Petri net with at least one place and transition:
\[
\text{createPetriNet(}tn: \text{String}, n: \text{String}, pn: \text{String})
\]
\[
\emptyset \Rightarrow \begin{array}{c}
\text{:Tr} \\
\text{name=tn}
\end{array} \\
\begin{array}{c}
\text{:PN} \\
\text{name=n}
\end{array} \\
\begin{array}{c}
\text{:Pl} \\
\text{name=pn}
\end{array}
\]

2. Inserting a new place in a given Petri net: \text{insertPlace(p:PN, n:String)}
\[
p:PN \Rightarrow p:PN \begin{array}{c}
\text{:Pl} \\
\text{name=n}
\end{array}
\]

3. Inserting a transition in a given Petri net: \text{insertTransition(p:PN, n:String)}
\[
p:PN \Rightarrow p:PN \begin{array}{c}
\text{:Tr} \\
\text{name=n}
\end{array}
\]

4. Connecting a transition to a place: \text{insertTPArc( t:Tr, w:int, p:Pl)}
\[
\begin{array}{c}
t:Tr \\
\text{weight=w}
\end{array} \\
\begin{array}{c}
\text{:TPArc} \\
\text{src=p:Pl} \\
\text{dst=n:PN}
\end{array} \\
\begin{array}{c}
\text{:Tr} \\
\text{weight=w}
\end{array} \\
\begin{array}{c}
\text{:TPArc} \\
\text{src=n:PN} \\
\text{dst=p:Pl}
\end{array}
\]

5. Connecting a place to a transition: \text{insertPTArc(p:Pl, w:int, t:Tr)}
\[
\begin{array}{c}
t:Tr \\
\text{weight=w}
\end{array} \\
\begin{array}{c}
\text{:PTArc} \\
\text{src=p:Pl} \\
\text{dst=n:PN}
\end{array} \\
\begin{array}{c}
\text{:Tr} \\
\text{weight=w}
\end{array} \\
\begin{array}{c}
\text{:PTArc} \\
\text{src=n:PN} \\
\text{dst=p:Pl}
\end{array}
\]

6. Inserting a token in a given place: \text{insertToken(p:Pl)}
\[
p:Pl \Rightarrow p:Pl \text{:Token}
\]

By consecutive rule applications from the start graph, a specific instance can be created. Let’s start with the start graph which is empty: Creating a Petri net with name ‘ProducerConsumer’ together with place ‘readyP’ and transition ‘produce’, inserting two places with names ‘buffer’ and ‘empty’, connecting them by two TPArcs and two PTArcs (all with weight 1) and inserting a token in place ‘readyP’ and another one in place ‘empty’ by corresponding rule applications would yield the abstract syntax graph in Figure 4.

2.5. Translating graph constraints to application conditions of grammar rules

After having translated OCL invariants to graph constraints, those constraints may be further translated to left application conditions (ACs) of transformation rules. The basic idea behind this additional step is the following: Given a graph grammar or graph transformation system, its rules may violate the graph constraints resulting from the OCL translation. In this case, corresponding rules are augmented by additional left ACs restricting their application. It has been shown in [7] that all possible applications of augmented rules ensure all constraints.

The set of left ACs for all rules being presented in Section 2.4 is discussed below. The corresponding translation process is presented in Section 7.

1. Creating a new Petri net: \text{createPetriNet}: This rule gets additional left ACs checking that all the names are not empty. Furthermore, the name of the new Petri net has to be compared with the names of existing nets in advance to ensure that all Petri nets have unique names. The resulting left AC is:
\[
\neg \exists \begin{array}{c}
\text{:PN} \\
\text{name=n}
\end{array} \land n \neq ' ' \land tn \neq ' ' \land pn \neq ' '
\]

8
2. Rules insertPlace and insertTransition: These rules get additional left ACs checking that the names of the new nodes are not empty or already used by existing places/transitions. The resulting left ACs are:

$$\neg \exists (p:PN_{place \ name = n} \land n \neq \text{''}) \quad \neg \exists (p:PN_{name = n} \land n \neq \text{''})$$

3. Rules insertTPArc, insertPTArc: These rules get additional left ACs checking that the weight w of a new arc is at least 1 and that there are not already TPArc or PTArc between the same pairs of place and transition. A node may carry a set of names. In the case of name set \{u,v\}, we write u,v (or u = v) inside the node rectangle. The resulting left ACs are:

$$w \geq 1 \land \neg \exists (1,t:Tr_{seatArc} \land (u,v))$$

4. Rule insertToken: This rule does not get any further left AC.

2.6. Generating meta-model instances

To generate the snippet of the example Petri net shown in Figure 4 we start with the empty start graph and apply the following rules:

createPetriNet('ProducerConsumer','readyP',produce'); insertPlace(pn,'buffer'); insertPlace(pn,'empty'); insertTPArc(tr1,1,pl2); insertPTArc(pl1,1,tr1); insertPTArc(pl3,1,tr1); insertToken(pl3)

These rules are applicable in this order at the given arguments such that all ACs are fulfilled: Applying rule createValidPetriNet to the empty graph there is no other Petri net with the same name and all newly inserted elements have proper names. Hence, the left AC is satisfied. For the following two applications of insertPlace we have to check that the names 'buffer' and 'empty' have not been used which is fulfilled. Moreover, they are non-empty. For the applications of insertTPArc and insertPTArc, we have to check that the chosen weights are at least 1 which is the case and that there are not already arcs of corresponding types between the specified places and transitions which is also the case. Rule insertToken can be applied without checking any further AC. Hence, the left ACs of all rules applied are fulfilled. The result is a valid instance graph.

3. Essential OCL Invariants

In this section, we recall the main concepts of OCL and its sub-language Essential OCL. Furthermore, we clarify which part of Essential OCL is translated by our approach and recall the formal syntax definition of OCL given in the OCL specification [1] which forms the basis our translation is building on.

3.1. OCL Language Description

The Object Constraint Language (OCL) [1] is a formal language used to describe expressions on object-oriented models being consistent to either the Meta Object Facility (MOF) [11] or the Unified Modeling Language (UML) [16]. These expressions typically specify invariant conditions that must hold for the system being modeled (e.g., see the running example in Section 2.1) or queries over objects described in a model. Further usages of OCL are the specification of initial and derived values as well as the specification of operation contracts, i.e., pre-, body-, and post-conditions of operations.
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. Predefined types are Integer, Real, String, and Boolean, called primitive data types. In meta-models, they are used as types of class attributes. 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(T₁,T₂) whose parameters T, T₁, and T₂ are applied to other types. 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.

Navigation in 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, a navigation results in either a single-valued return type (for multiplicities with upper bound set to 1) or in a multi-valued type, more precisely in a set (for multiplicities with upper bound greater than 1). If, in a multi-valued reference, there does not exist any target object, the navigation results in an empty set. Whereas, in the case of multiplicity 0..1, the absence of an appropriate value yields null representing the only value of bottom type OclVoid.

Logic used in OCL. We mentioned above that (1) OclVoid is a subtype of any custom and predefined type, i.e., it is also a subtype of the predefined type Boolean, and that (2) OclVoid consists of value null. Moreover, OclInvalid is used as an additional type to allow invalid outcomes of functions. As a consequence, OCL type Boolean comes along with a four-valued logic, i.e., Boolean={true,false,null,invalid} [1]. Besides and, or, not, xor, and implies, OCL provides a universal quantifier forAll and an existential quantifier exists, both in the sense 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 [17].

OCL collection type operations. In the following, we give a rough overview of some selected but substantial predefined collection type operations being called by the arrow-notation (for example, someSet->foo()). 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. Extraction operations extract some information from the given collection except for Boolean values. Examples of this kind of operations are size(), collect(Exp), and union(Collection(T)). Operation size() returns the number of elements within the collection while collect(Exp) is used to construct new collections (with potentially other type elements) from existing ones according to the expression Exp. Finally, OCL provides a number of operations returning Boolean values. For checking the existence of elements within a collection, isEmpty() respectively notEmpty() can be used, 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.

3.2. Essential OCL

According to the OCL specification of the OMG [1], Essential OCL is “…the minimal OCL required to work with EMOF”. Essential MOF (EMOF) is a subset of MOF [1] that allows to define simple meta-models using simple concepts. In the following, we consider a slightly restricted form of Essential OCL neglecting OCL features that cannot be translated. W.r.t. our translation of EssentialOCL to nested graph constraints, we restrict our consideration to a two-valued logic. The exact limitations of our translation are presented in Section 5.4.
Figure 5 shows the type system of the Essential OCL meta-model we consider in this paper. The meta-classes are embedded into the corresponding EMOF meta-model. As can be seen, the EMOF type system is extended by the specific collection type SetType whose meta-class has a light gray-colored background. Figure 6 shows the substantial part of the Essential OCL meta-model according to the expressions we translate to graph constraints. Meta-classes which are reused from the type system presented in Figure 5 have a white, new meta-classes have a light gray-colored background. An invariant (respectively Constraint) on a Classifier is defined by an ExpressionInOCL that owns a contextual Variable which is named self in most cases. Since OCL is a strongly typed language, the context variable is typed by the constrained element of the invariant, whereas the invariant itself has type Boolean which is a concrete implementation of PrimitiveType.

The concrete specification of an invariant is given by a subclass of OclExpression. The following subclasses of OclExpression are possible: a VariableExpression to refer to a variable; a PrimitiveLiteralExp to refer to a primitive type literal, e.g., String foo; an OperationCallExp to refer to an operation of a primitive type like the addition of integers, to an auxiliary non-recursive query operation of a class, or to a set type operation like isEmpty; a PropertyCallExp to enable navigation to class attributes (typed by a primitive type or an enumeration) or to an association end (typed by a class), both represented as instances of meta-type Property; a SetLiteralExp to refer to a set of model elements which are represented by meta-type SetItem; an EnumLiteralExp to refer to an enumeration literal; a TypeExp to provide type checking and type casting; an IfExp to provide conditional expressions; a LetExp to define and initialize a local variable for use within another expression; or finally an IteratorExp representing a looping execution on each element of a given set (used in exists and forAll, for example).

Example 1. Figure 7 shows the abstract syntax graph of an OCL invariant specifying that the weight of an arc within a Petri net instance is positive: context Arc inv: self.weight >= 1 (see Section 2). Class Arc is constrained by an instance of meta-class Constraint whose specification owns a variable named self of type Arc. The invariant is specified by calling the operation greater or equals defined on the primitive

---

2 Indeed, all meta-classes are direct or indirect sub classes of TypedElement (see Figure 6).
Figure 6: The expressions part of the Essential OCL meta-model.

type Integer. The arguments for this operation call are obtained (1) by calling the value of property weight
on the arc object referenced by the context variable, and (2) by the value of integer literal 1.

3.3. Formalizing the OCL syntax

We base our OCL translation on the OCL specification [11], Annex A which itself is based on the doctoral thesis by Richters [13]. We prefer this formalization, in contrast to the UML-based specification, since it is more suitable for proving the semantics preservation of our translation. In the following, we recall the main definitions from [13] formalizing the OCL syntax. As a first step, we define an object model representing the EMOF-based meta-model types as follows.

**Assumption 1.** Let $DSIG = (S, OP)$ be a data signature with $S = \{\text{Integer, Real, Boolean, String}\}$ and corresponding operation symbols $OP$.

**Definition 1 (Object Model).** An object model over $DSIG$ is a structure $M = (\text{CLASS, ABSCL, ENUM, ATT, OPS, ASSOC, assoc, } \iota_{\text{tgt}}, \text{mult, } \prec)$ where CLASS is a finite set of classes with a true subset of abstract classes $\text{ABSCL} \subset \text{CLASS}$, ENUM is a finite set of enumerations, ATT $= \{\text{ATT}_c\}_{c \in \text{CLASS}}$ is a family of attributes $\text{att}: c \rightarrow (S \cup \text{ENUM})$ of class $c$, OPS is a family of user-defined operation signatures $\omega: w \rightarrow t$ where $w \in (S \cup \text{ENUM} \cup \text{CLASS})^*$ and $t \in (S \cup \text{ENUM} \cup \text{CLASS})$ having no side-effects (queries), ASSOC is a set of associations, with $\text{assoc}: \text{ASSOC} \rightarrow (\text{CLASS} \times \text{CLASS})$ mapping each association to a pair of participating classes and $\iota_{\text{tgt}}: \text{ASSOC} \rightarrow \text{String}$ mapping each association to a non-empty role name, $\text{mult}: \text{ASSOC} \rightarrow \mathcal{P}(\mathbb{N}_+)^3$ is a function assigning each association a multiplicity interval and $\prec$ is a partial order on CLASS reflecting its generalization hierarchy.

**Example 2.** Regard the Petri net meta-model shown in Figure 2. The corresponding object model $M$ consists of CLASS $= \{\text{NE}, \text{PN}, \text{Pl}, \text{Tk}, \text{Arc}, \text{PTArc}, \text{TPArc}, \text{Tr}\}$, ABSCL $= \{\text{NE}, \text{Arc}\}$, ENUM $= \emptyset$, ATT $= \{\text{name} : \text{NE} \rightarrow \text{String}, \text{weight} : \text{Arc} \rightarrow \text{Integer}\}$, OPS $= \emptyset$, ASSOC $= \{\text{PN, place}\}$.

---

$^3\mathcal{P}(\mathbb{N}_+)$ denotes the power set of the natural numbers (excluding $\{0\}$ since an association with both lower bound and upper bound of $0$ makes no sense). More specifically, we consider intervals with lower and upper bound. Moreover, the upper bound may be * which means infinite.
PN\text{transition}, \text{PN}_{\text{arc}}, \text{Pl}_{\text{token}}, \text{Pl}_{\text{preArc}}, \text{Pl}_{\text{postArc}}, \text{PT}_{\text{src}}, \text{PT}_{\text{dst}}, \text{Tr}_{\text{preArc}}, \text{Tr}_{\text{postArc}}, \text{TP}_{\text{src}}, \text{TP}_{\text{dst}}$ with, e.g. for association $a = \text{PT}_{\text{src}}$, \text{assoc}(a) := \langle \text{PT}, \text{Pl} \rangle$, $r_{\text{tgt}}(a) = \text{src}$, and mult$(a) = \{1\}$, and $\subseteq \{\text{PN} \prec \text{NE}, \text{Pl} \prec \text{NE}, \text{Tr} \prec \text{NE}, \text{PT} \prec \text{Arc}, \text{TP} \prec \text{Arc}\}$.

Since the evaluation of an OCL invariant requires knowledge about the complete context of an object model at a discrete point in time, we recall the definition of a system state of an object model $M$. Informally, a system state consists of a set of objects, functions assigning attribute values to each object attribute, and a finite set of links connecting objects within the model.

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

- for each class $c \in \text{CLASS}$, $\sigma_{\text{Class}}(c)$ is a finite set of object identifiers (since abstract classes cannot be instantiated), $\sigma_{\text{Class}}(c) = \emptyset$ for each $c \in \text{ABSCL}$,
- for each attribute $\text{att} : c \rightarrow t \in \text{ATT}_{\text{c}}^\prec$, $\sigma_{\text{Att}}(\text{att}) : \sigma_{\text{Class}}(c) \rightarrow I(t)$ is an operation from class objects to some interpretation of type $t \in (S \cup \text{ENUM})$ where $\text{ATT}_{\text{c}}^\prec := \bigcup_{c' < c} \text{ATT}_{c'}^\prec$ represents the set of all owned and inherited attribute symbols of a class $c$,
- for each association $a \in \text{ASSOC}$ with $\text{assoc}(a) = (e_1, e_2)$, $\sigma_{\text{Assoc}}(a) \subset \sigma_{\text{Class}}(e_1) \times \sigma_{\text{Class}}(e_2)$ is a finite set of links connecting objects where $\sigma_{\text{Class}}(c) := \bigcup_{c' < c} \sigma_{\text{Class}}(c')$ is the set of all objects of type $c$ or a (possibly indirect) subtype of $c$. Furthermore, $\sigma_{\text{Assoc}}(a)$ must meet the multiplicity specification for $a$: $\forall a_1 \in \sigma_{\text{Class}}(c_1) : \{|l = (a_1, a_2) | l \in \sigma_{\text{Assoc}}(a)\} \in \text{mult}(a)$.

**Example 3.** The system state $\sigma(M)$ corresponding to the Petri net instance in Figure 3 consists of $\sigma_{\text{Class}} = (\sigma_{\text{Class}}(\text{PN}), \sigma_{\text{Class}}(\text{Pl}), \sigma_{\text{Class}}(\text{Tk}), \sigma_{\text{Class}}(\text{PT}), \sigma_{\text{Class}}(\text{Tr}))$ with, e.g., $\sigma_{\text{Class}}(\text{Tr}) = \{tr_1, tr_2\}$ and $\sigma_{\text{Class}}(\text{TP}) = \{tp_1, tp_2, tp_3, tp_4\}$, e.g., $\sigma_{\text{Att}}(\text{name})(tr_1) =$ \text{produce} and $\sigma_{\text{Att}}(\text{weight})(tp_1) = 1$, and, e.g., $\sigma_{\text{Assoc}}(\text{Tr}_{\text{postArc}}) = \{< tr_1, tp_1 >, < tr_1, tp_2 >, < tr_2, tp_3 >, < tr_2, tp_4 >\}$ and $\sigma_{\text{Assoc}}(\text{TP}_{\text{src}}) = \{< tp_1, tr_1 >, < tp_2, tr_1 >, < tp_3, tr_2 >, < tp_4, tr_2 >\}$.

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

**Definition 3 (Signature).** A signature over an object model $M$ is a structure $\Sigma_M = (T_M, \preceq_M, \Omega_M)$ where $T_M$ is a set of types consisting of all basic types ($S$ in $\text{DSIG}$), all object types (for each $c \in \text{CLASS}$ there is an object type $t_c \in T_M$), all enumeration types $E \in \text{ENUM}$, and the collection type $\text{Set}(t_c)$ for an arbitrary object type $t_c \in T_M$. $\preceq_M$ is a partial order on $T_M$ representing a type hierarchy over $T_M$, where $\text{Integer} \preceq_M \text{Real}$ and $t_c \preceq_M t_{c'}$ if $c < c'$. $\Omega_M$ is a set of operations on $T_M$ consisting of an exhaustive set of predefined operations on primitive data types such as comparison operations on Integer, implication on Boolean, etc., operations $\text{allInstances}_{t_c}$ for obtaining all objects of type $t_c$, operations $a : t_c \rightarrow t$ to access type attributes, operations $\omega : w \rightarrow t$ to obtain the result of a query, operations $\text{role}' : t_c \rightarrow t_{c'}$ with $a \in \text{ASSOC}$, $\text{assoc}(a) = (c, c')$, $r_{\text{tgt}}(a) = \text{role}'$, $\text{mult}(a) = M_{c'}$, and $M_{c'} \subseteq \{0, 1\}$ to access single-valued associations of type $t_c$, operations $\text{role} : t_c \rightarrow \text{Set}(t_{c'})$ with $a \in \text{ASSOC}$, $\text{assoc}(a) = (c, c')$, $r_{\text{tgt}}(a) = \text{role}$, $\text{mult}(a) = M_{c'}$, and $M_{c'} \subseteq \{0, 1\}$ to access multi-valued associations of type $t_c$, operations on sets, i.e., $\text{isEmpty}$, $\text{notEmpty}$, $\text{includes}$, $\text{includesAll}$, $\text{excludes}$, $\text{excludesAll}$, $\text{including}$, $\text{excluding}$, $\text{size}$, $\text{union}$, $\text{−}$, $\text{intersection}$, and $\text{symmetricDifference}$, the constructor $\text{mkSet}_t$ for creating a set with elements of type $t$, and operations equality $(\approx)$ and non-equality $(\neq)$ for all types $t \in T_M$.

For specifying expressions in Essential OCL we use a signature over an object model $M$ as defined above and a family of variable sets indexed by types $t \in T_M$.

---

\footnote{Since in the meta-model shown in Figure 2 associations are unnamed, we use the convention $A_b$ to label an association between classes $A$ and $B$ with $r_{\text{tgt}} = b$.}
Definition 4 (Essential OCL Expressions). Let \( \Sigma_M = (T_M, \leq_M, \Omega_M) \) be a signature over an object model \( M \). Let \( Var = \{ Var_t \}_{t \in T_M} \) be a family of variable sets indexed by types \( t \in T_M \). The family of Essential OCL expressions over \( \Sigma_M \) is given by \( Expr = \{ Expr_t \}_{t \in T_M} \) representing sets of expressions. The syntax of an Essential OCL expression is defined recursively as follows (see also Figure 6):

- **VariableExpression**: \( v \in Expr_t \) for each variable \( v \in Var_t \), referring to a context, iterator or local variable.
- **OperationExp**: \( e := \omega(e_1, \ldots, e_n) \in Expr_t \) for each operation symbol \( \omega : t_1 \times \cdots \times t_n \to t \in \Omega_M \) and for all \( e_i \in Expr_{t_i} (1 \leq i \leq n) \). This includes predefined operations on data types and user-defined non-recursive queries (OperationCallExp), class attribute operations, navigable association end operations (both PropertyCallExp), and constants (LiteralExp). Tables 1 and 2 in Section 10 give an overview of the syntax of selected concrete operation expressions in Essential OCL.
- **IfExp**: If \( e_1, e_2, e_3 \in Expr_{Boolean} \) then \( e := \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \in Expr_{Boolean} \).
- **TypeExp**: If \( e \in Expr_t, t', t'' \in T_M \), and \( t'' \leq_M t \) then \( e.oclIsTypeOf(t') \), \( e.oclIsKindOf(t') \) \( \in Expr_{Boolean} \) and \( e.oclAsType(t'') \in Expr_{t''} \).
- **IteratorExp**: If \( s \in Expr_{Set(t)}, v \in Var_t, b \in Expr_{Boolean}, e_1 \in Expr_v, \) and \( e_2 \in Expr_{Set(v)} \) then \( s->\exists(v | b) \), \( e->\text{select}(v | b) \in Expr_{Set(v)}, s->\text{collect}(v | e_1) \in Expr_{Set(v)} \). This formal definition of collect results in sets of values instead of bags, possibly yielding duplicate values. The translation approach presented in this paper thus restricts the expressiveness of collect. However, in many circumstances, not the number of duplicate values is crucial, but the collection of distinct values.
- **LetExp**: If \( v \in Var_t, e \in Expr_t, e' \in Expr_t, \) and \( t, t'' \in T_M \) then \( \text{let } v = e \text{ in } e' \in Expr_{t'} \).

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

Definition 5 (Essential OCL Invariant). Given an object model \( M \), an Essential OCL invariant is a Boolean OCL expression with a free variable \( v \in Var_C \) where \( C \) is a classifier type. The concrete syntax of an invariant is: \( \text{context } v : C \ inv : \langle expr \rangle \). The set \( \text{Invariant}_M \) denotes the set of all Essential OCL invariants over \( M \).

The semantics definition of Essential OCL expressions which is necessary to proof correctness, is recalled in Section 6.1.

4. Nested Graph Constraints

In this section, we recall the definition of typed attributed graphs with node type inheritance and introduce nested constraints over this type of graphs and compact constraints, compact representations of nested constraints, used in the translation of Essential OCL invariants.

4.1. Graphs

In this subsection, we recall the definitions of A-graphs, attributed graphs, and typed attributed graphs with node type inheritance extended by a set of variables and a set of formulas that constrain the possible values of these variables, closely related to symbolic graphs and show some properties of these graphs. This type of graphs forms the basis to define typed attributed nested graph constraints. A directed graph consists of a set of nodes and a set of edges where each edge is equipped with a source and a target node. An A-graph is a directed graph together with a set of data nodes and a set of node attribute edges where each node attribute edge is equipped with a graph node and a data node.
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: \( src_G : G_E \to G_V, tgt_G : G_E \to G_V \) for graph edges and \( src_A : G_A \to G_V, tgt_A : G_A \to G_D \) for node attribute edges. Let \( Names \) be a set of names for nodes, \( n_G : G_V \to P(Names) \) is an injective function, called name function, that assigns a set of names to each node of \( G \), e.g. \( n_G(v) = \{u, v\} \), written as \( u = v \). Given two A-graphs \( G^1 \) and \( G^2 \), an A-graph morphism \( f : G^1 \to G^2 \) is a tuple of functions \( f_V : G_V^1 \to G_V^2 \), \( f_D : G_D^1 \to G_D^2 \), \( f_E : G_E^1 \to G_E^2 \) and \( f_A : G_A^1 \to G_A^2 \) such that \( f \) commutes with all source and target functions, e.g. \( f_V \circ src_G^1 = src_G^2 \circ f_E \). An A-graph morphism \( f \) is injective if the functions \( f_V, f_D, f_E \), and \( f_A \) are injective. An injective morphism \( f : G \to H \) is an inclusion if \( n_G(x) \subseteq n_H(f(x)) \) for all items \( x \in G \).

An attributed graph is an A-graph combined with an algebra over a data signature \( DSIG \), in the sense of algebraic signatures (see e.g. [21]). In the signature, we distinguish a set of attribute value sorts. The corresponding carrier sets in the algebra are used for the attribution. Our definition generalizes largely the one in [19] by allowing variables and a set of formulas that constrain the possible values of these variables and is closely related to symbolic graphs [20].

Definition 7 (Attributed graphs). Let \( DSIG = (S, OP) \) be a data signature, \( X = \{X_i\}_{i \in S} \) be 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 over the attribute values of \( G \), and \( \Phi \) is a finite set of \( DSIG \)-formulas over attributes with free variables in \( X \). A set \( \{F_1, \ldots, F_n\} \) of formulas can be regarded as a single formula \( F_1 \land \ldots \land F_n \). An attributed graph \( AG = (G, D, \Phi) \) with an empty set of formulas is basic and is compactly denoted by \( AG = (G, D) \). Given two attributed graphs \( AG^1 \) and \( AG^2 \), an attributed graph morphism \( f : AG^1 \to AG^2 \) is a pair \( f = (f_G, f_D) \) of an attributed graph morphism \( f_G : G^1 \to G^2 \) and a \( DSIG \)-homomorphism \( f_D : D^1 \to D^2 \) such that \( f_G^2 \equiv f_D(\Phi^1) \) where \( f_D(\Phi^1) \) is the set of formulas obtained when replacing in \( \Phi^1 \) every variable \( x \) in \( G^1 \) by \( f_D(x) \). An attributed graph morphism \( f \) is injective (an inclusion) if \( f_G \) and \( f_D \) are injective (inclusions).

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

An attributed type graph with inheritance is an attributed graph with a distinguished set of abstract nodes and inheritance relations between the nodes. The inheritance clan of a node represents all its subnodes. The notion of typed graph morphism has to be extended to capture the inheritance clan. They are called clan morphisms.

Definition 8 (Typed attributed graphs). An attributed type graph with inheritance \( ATGI = (TG, Z, I) \) consists of an A-graph \( TG \), the final algebra \( Z \) over attributes with an empty set of formulas, and a simple inheritance graph \( I \) (i.e. having neither multiple edges nor loops) with \( I_V = TG_V \). For each node \( v \in I_V \), the inheritance clan of \( v \) is the set \( clan_I(v) = \{v' \in I_V \mid \exists \text{ path } v' \xrightarrow{*} v \text{ in } I\} \) where \( v' \xrightarrow{*} v \) in \( I \) stands for a directed path in \( I \) from \( v' \) to \( v \) of length \( \geq 0 \). 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 : \( AG \to ATGI \).
Given two ATGI-graphs $AG^1 = (G^1, \type^1)$ and $AG^2 = (G^2, \type^2)$, an ATGI-morphism $f: AG^1 \to AG^2$ is an attributed graph morphism such that $\type^2 \circ f \preceq \type^1$. An ATGI-morphism is called (type-)strict if $\type^2 \circ f = \type^1$.

An example of a typed attributed graph is shown in Figure 4.

Lemma 1 (Properties of typed attributed graphs). ATGI-graphs and ATGI-morphisms form the category $\mathcal{AGraphs}_{ATGI}$. The category has $\mathcal{M}$-pushouts, i.e., pushouts where at least one morphism is in $\mathcal{M}$. Moreover, it has $\mathcal{E}'$-$\mathcal{M}$ pair factorization with $\mathcal{M}$ being the class of all type-strict injective morphisms and $\mathcal{E}'$ the class of all jointly surjective pairs.

Proof. We prove the lemma in two steps. For attributed graphs and attributed morphisms, the proof follows more or less from the corresponding result in [19]. The first statement is straightforward. For the second statement, let $r: K \to R$ and $d: K \to D$ be attributed morphisms on basic attributed graphs and $\Phi_K, \Phi_R, \Phi_D$ be the corresponding sets of formulas. By [19], there are a basic attributed graph $H$ and basic attributed morphisms $r': R \to H$ and $h: D \to H$ such that (1) $r \circ h = r\circ d$. Let $\Phi_H$ 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. For $\mathcal{E}'$-$\mathcal{M}$ pair factorization, we refer to the proof in [19, Remark 5.26]: For morphisms $f_j: A_j \to C$ ($j \in \{1, 2\}$), compute the coproduct injections $i_j: A_j \to A_1 + A_2$ component-wise on the node, edge and attribute sets and $\Phi_{A_1 + A_2} = \Phi_{A_1} \cup \Phi_{A_2}$. Define $f: A_1 + A_2 \to C$ as $f(x) = f_j(x)$ for $x \in i_j(A_j)$ and compute $K, e: A_1 + A_2 \to K$ and $m: K \to C$ as epi-mono factorization $f = m \circ e$, i.e. $K = f(A_1 + A_2)$, $\Phi_K = \Phi_C$ and $m$ an injection. Finally, $(e_1, e_2) = (e \circ i_1, e \circ i_2)$. 

For ATGI-graphs and ATGI-morphisms, the statements follow the corresponding statements for attributed graphs and attributed morphisms: The first statement is straightforward. For an injective $d: K \to D$ and type-strict injective $r: K \to R$ with typing-morphisms $\type_K: K \to ATGI$, $\type_D: D \to ATGI$ and $\type_R: R \to ATGI$ compute the pushout (1) in $\mathcal{AGraphs}$ - ignoring typing - (as above) and choose the typing-morphism $\type_H: H \to ATGI$ as follows: For $y \in H$ with $x \in D$ and $h(x) = y$ let $\type_H(y) = \type_D(x)$. Let $\type_H(y) = \type_R(z)$ with $z \in R$ and $r'(z) = y$ otherwise. Morphisms $h$ and $r'$ are valid morphisms in $\mathcal{AGraphs}_{ATGI}$ since for all $x \in K$, $\type_H(h(d(x))) = \type_D(d(x)) \preceq \type_K(x) = \type_R(r(x))$ where $x \preceq y$ means that $x \in clan(y)$. Furthermore, (1) is a pushout in $\mathcal{AGraphs}_{ATGI}$ since for any pair $(f: D \to G, g: R \to G)$ of morphisms with common co-domain $G$ and $f \circ d = g \circ r$ there is a morphism $m: H \to G$ with $m \circ h = f$, $m \circ r' = g$ in $\mathcal{AGraphs}$ (since (1) is pushout in $\mathcal{AGraphs}$). Since for all $y \in H$ there exists $x \in D$ with $h(x) = y$ and $\type_H(x) = \type_D(y) \preceq \type_G(f(x)) = \type_G(m(y))$ or $z \in R$ with $r'(z) = y$ and $\type_R(z) = \type_G(y) \preceq \type_G(g(r)) = \type_G(m(x))$, $m$ is also a morphism in $\mathcal{AGraphs}_{ATGI}$. $\mathcal{E}'$-$\mathcal{M}$ pair factorization is similar: Let $f_i: A_i \to C$ ($i \in \{1, 2\}$) with typing $\type_i: A_i \to ATGI$ and $\type_C: C \to ATGI$. Compute the $\mathcal{E}'$-$\mathcal{M}$ pair factorization in $\mathcal{AGraphs}$ - ignoring typing - (as above) and choose the typing-morphism $\type_K: K \to ATGI$ such that $\type_K = \type_C \circ m$, i.e. $\type_K(x) = \type_C(m(x))$.
type\(_C(m(x))\) for \(x \in K\). Morphisms \(e_1\) and \(e_2\) respect typing since \(type_K(e_i(x)) = type_C(f_i(x)) \leq type_i(x)\) for all \(x \in A_i, \ i \in \{1,2\}\). \(\square\)

In the following, we always use typed attributed graphs and ATGI-morphisms but simply call them graphs and (graph) morphisms.

4.2. Nested graph constraints

In this section, we introduce the concepts of nested constraints over typed attributed graphs with node type inheritance, short nested graph constraints or graph constraints, and their compact representations used in the translation of Essential OCL Invariants.

Graph conditions and graph constraints \[22^7\] are nested constructs, which can be represented as trees of morphisms equipped with quantifiers and Boolean connectives. Graph conditions are implemented in, e.g., the tool environments AGG \[23\], GROOVE \[24\], GrGen \[25\], and Henshin \[26\].

Definition 9 (Nested graph conditions). A nested graph condition on typed attributed graphs, short condition, over a graph \(P\) is of the form true or \(\exists(a,c)\) where \(a: P \rightarrow C\) is an injective morphism and \(c\) is a condition over \(C\). Boolean formulas over conditions over \(P\) yield conditions over \(P\), that is, for conditions \(c, c_i\) (\(i \in I\) with \(I\) being finite) over \(P\), \(\neg\) and \(\bigwedge_{i \in I} c_i\) are conditions over \(P\). Conditions over the empty graph \(\emptyset\) are also called constraints. In the context of rules, conditions are called application conditions.

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

Examples for nested constraints and conditions can be found in Section \[2\] above. The satisfaction of a graph condition is established by the presence and absence of certain graph morphisms from the graphs within the condition to the tested graph. The presented injective satisfiability notion restricts these morphisms to be injective: no identification of nodes and edges is allowed. In this way, explicit counting such as the existence/non-existence of \(n\) nodes is easily expressible.

Definition 10 (Semantics). Satisfiability of a condition over \(P\) by an injective morphism \(p: P \rightarrow G\) is inductively defined as follows: \(p\) satisfies true. \(p\) satisfies \(\exists(P, \rightarrow C, c)\) if there exists an injective morphism \(q: C \rightarrow G\) such that \(p = q \circ a\) and \(q\) satisfies \(c\).

For Boolean formulas over conditions, the semantics is as usual: \(p\) satisfies \(\neg c\) if \(p\) does not satisfy \(c\), and \(p\) satisfies \(\bigwedge_{i \in I} c_i\) if \(p\) satisfies each \(c_i\) (\(i \in I\)). We write \(p \models c\) if \(p\) satisfies the condition \(c\) over \(P\).

For simplicity, we also consider so-called compact conditions, built from true and compact conditions of the form \(\exists(C, c)\) where \(C\) is a graph and \(c\) is a compact condition. One may translate nested conditions to compact conditions as follows: By definition, nested conditions are based on injective morphisms. Without loss of generality, they also may be based on inclusions. For each inclusion in a condition, the domain is not represented when it can unambiguously inferred, e.g. \(\forall(C_1, \exists(C_2, c)) := \forall(\emptyset \rightarrow C_1, \exists(C_1 \rightarrow C_2, c))\). Examples of compact conditions can be found in Section \[2.3\].
Definition 11 (Compact conditions). A compact (nested) condition on typed attributed graphs is of the form true or \( \exists (C, c) \) where \( C \) is a graph and \( c \) is a compact condition. Boolean formulas over compact conditions yield compact conditions. \( \exists (C) \) abbreviates \( \exists (C, \text{true}) \).

The semantics of compact conditions is defined by the semantics of nested conditions. In addition to the short notation for nested conditions we allow for compact conditions to omit parts of the graphs yielding less complex expressions. However, this makes a completion step necessary to infer the missing parts of the graphs in the inner conditions from the surrounding. To identify the parts of the graphs being matched, we tag each node in a graph with a set of names with the restriction that no two nopens in a graph have a name in common. We call a morphism \( p : P \rightarrow Q \) name preserving if, for all nodes \( v \in P_v, p_v(v) \) has all the names from \( v \).

Construction (From compact conditions to conditions). For a graph \( P \) and a compact condition \( d \), \( \text{Cond}(P,d) \) denotes the condition over \( P \), inductively defined as follows:

\[
\begin{align*}
\emptyset & \longrightarrow C \quad \equiv_C \quad \text{Cond}(P, \text{true}) = \text{true}.
\end{align*}
\]

\[
\begin{align*}
P & \xrightarrow{a} C' \equiv_C \quad \text{Cond}(P, \exists (C, c)) = \bigvee_{(a,b) \in F} \exists (P, C' \rightarrow \text{Cond}(C', c))
\end{align*}
\]

where \( F = \{(a,b) \mid \text{jointly surjective, } a, b \text{ inclusions} \} \) and \((a, b)\) is jointly surjective if, for each \( x \in C' \), there is a preimage of \( x \) in \( P \) or in \( C \).

\[
\begin{align*}
\text{Cond}(P, \neg c) &= \neg \text{Cond}(P, c) \\
\text{Cond}(P, c \land i_{cJ_G}) &= \land_{i \in J_G} \text{Cond}(P, c_i).
\end{align*}
\]

Remark 1. The Cond and the Shift construction in [27] look very similar. While Shift is based on injective morphisms, Cond is restricted on name-preserving morphisms in order to identify nodes with common names. Cond is based on empty morphisms and “completes” compact conditions \( \exists (C, c) \) with empty morphism to \( C \) with respect to an empty morphism \( b \) to \( P' \). Instead of the empty morphisms, we write the codomain of the morphisms.

Example 4. The OCL constraint

\[
\begin{align*}
\text{context Petrinet inv: self.transition->forAll(t:Transition | t.preArc->notEmpty() \lor t.postArc->notEmpty())}
\end{align*}
\]

can be expressed as the compact condition

\[
\begin{align*}
\forall (\emptyset \rightarrow \text{self.PN} \rightarrow t:Tr) \exists (\text{self.PN} \rightarrow t:Tr \xrightarrow{\text{preArc}} v1:PTArc) \lor \exists (\text{self.PN} \rightarrow t:Tr \xrightarrow{\text{postArc}} v2:TPArc) \text{true})
\end{align*}
\]

The completion over the empty graph leads to the nested condition

\[
\begin{align*}
\forall (\emptyset \rightarrow \text{self.PN} \rightarrow t:Tr) \exists (\text{self.PN} \rightarrow t:Tr \xrightarrow{\text{preArc}} v1:PTArc) \lor \exists (\text{self.PN} \rightarrow t:Tr \xrightarrow{\text{postArc}} v2:TPArc) \text{true})
\end{align*}
\]

In more detail:

\[
\begin{align*}
\text{Cond}(\emptyset, \forall (\text{self.PN} \rightarrow t:Tr) \exists (t:Tr \xrightarrow{\text{preArc}} v1:PTArc) \lor \exists (t:Tr \xrightarrow{\text{postArc}} v2:TPArc) \text{true}) \\
\equiv \text{Cond}(\emptyset, \forall (\text{self.PN} \rightarrow t:Tr) \exists (t:Tr \xrightarrow{\text{preArc}} v1:PTArc) \lor \exists (t:Tr \xrightarrow{\text{postArc}} v2:TPArc) \text{true}) \\
\equiv \neg \exists (t:Tr \xrightarrow{\text{preArc}} v1:PTArc) \lor \exists (t:Tr \xrightarrow{\text{postArc}} v2:TPArc) \text{true}) \\
\equiv \neg \exists (t:Tr \xrightarrow{\text{preArc}} v1:PTArc) \lor \exists (t:Tr \xrightarrow{\text{postArc}} v2:TPArc) \text{true}) \\
\end{align*}
\]

Note that this constraint does not occur in Section 2 since definitions of Petri nets usually allow transitions without arcs.
Satisfiability of a compact condition is defined by the satisfiability of the corresponding nested condition.

**Definition (Semantics of compact conditions).** An injective morphism $p$ satisfies the compact condition $c$, denoted $p \models c$, if $p$ satisfies the nested condition $\text{Cond}(P, c)$. Two compact conditions $c$ and $c'$ are equivalent wrt. a graph $P$, denoted $c \equiv_P c'$, if $\text{Cond}(P, c) \equiv \text{Cond}(P, c')$. We call them equivalent if they are equivalent wrt. any graph.

By definition, compact conditions and nested conditions have the same expressive power. Somewhat ambiguously, compact conditions are also called conditions.

The following equivalences can be used to simplify compact conditions.

**Lemma 2 (equivalences).** Let $C_1 \oplus_P C_2$ denote the gluing of $C_1$ and $C_2$ along $P$ and let $\mathcal{P}$ denote the set of all intersections of $C_1$ and $C_2$.

\begin{enumerate}[(E1)]
  
  \item (a) $\exists(C_1, \exists(C_2)) \equiv \forall_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2)$,  
  
  \item (b) $\exists(C_1, \exists(C_2)) \equiv \exists(C_1 + C_2)$ if $C_1$ and $C_2$ are clan-disjoint,  
  
  \quad i.e., if the clans of the types of $C_1$ and $C_2$ are disjoint  
  
  \quad and $C_1 + C_2$ denotes the disjoint union of $C_1$ and $C_2$.
\end{enumerate}

\begin{enumerate}[(E2)]
  \item (a) $\exists(C_1, \exists(C_2 \land \exists(C_3)) \equiv \exists(C_1, \forall_{P \in \mathcal{P}} \exists(C_2 \oplus_P C_3))$, if for all node names occurring in both $C_2$ and $C_3$, a node with that name already exists in $C_1$,  
  
  \item (b) $\exists(C_1) \land \exists(C_2) \equiv \exists(C_1 + C_2)$ if $C_1$ and $C_2$ are clan-disjoint and have disjoint sets of node names.
\end{enumerate}

\begin{enumerate}[(E3)]
  \item $\exists(u:v \mapsto C) \land \exists(u:v \mapsto C') \equiv \exists(u:v \mapsto \exists(C[u=v]))$ provided that either $u$ or $v$ does not exist in $C$ and $C[u=v]$ is the graph obtained from $C$ by renaming $u$ by $u = v$.
\end{enumerate}

**Proof.** The proof of the equivalences makes use of the Pullback-Pushout-Lemma in [25]: The pushout of the pullback of a pair $(b_1, b_2) \in \mathcal{F}$ leads to the pushout $C_1 \oplus_P C_2$ of $C_1$ and $C_2$ along the pullback $P$. In the following, $\mathcal{P}$ denotes the set of pairs $(a_1, a_2)$ induced by the pairs $(b_1, b_2) \in \mathcal{F}$.

\[
\begin{array}{c}
P\longrightarrow \quad C_2
\
\downarrow a_2 \quad \downarrow \quad \downarrow b_2
\
C_1 \quad \quad \quad \quad \quad \quad C
\end{array}
\]

(E1) (a) follows with the help of the definition of $\text{Cond}$:

\[
\begin{align*}
\exists(C_1, \exists(C_2)) & \equiv \text{Cond}(P_0, \exists(C_1, \exists(C_2))) \\
& \equiv \forall_{(a, b) \in \mathcal{F}} \exists(a, \text{Cond}(C_1', \exists(C_2, \text{true}))) \\
& \equiv \forall_{(a, b) \in \mathcal{F}} \exists(a, \forall_{(a', b') \in \mathcal{F}} \exists(a', \text{Cond}(C_1', \exists(C_2, \text{true})))) \\
& \equiv \forall_{(a, b) \in \mathcal{F}} \exists(a, \forall_{(a', b') \in \mathcal{F}} \exists(a', \text{true})) \\
& \equiv \forall_{(a, b) \in \mathcal{F}} \forall_{(a', b') \in \mathcal{F}} \exists(a' \circ a) \\
& \equiv \forall_{(a, b) \in \mathcal{F}} \exists(C_1', \forall_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2)) \\
& \equiv \forall_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2).
\end{align*}
\]

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

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

20
(E2) follows from the definition of Cond and $|=\cdot$

We show both directions separately.

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

For $\Leftarrow$ consider the following commutative diagram on the left. By definition of Cond, some $P \in P$, $C$, $C'$ and $C'_1$ with $C' \rightarrow G$ exist. Let $P_2$ and $P_3$ be the common part of $C'_1$ and $C_2$, $C_3$ respectively. By $E\cdot M$ pair factorization, $C'_1$ and $C'_3$ also exist and with the definition of $|=\cdot$, $p |= \exists(C_1, \exists(C_2) \land \exists(C_3))$. In the case of clan-disjointness of $C_1$ and $C_2$, $\exists(C_1) \land \exists(C_2) \equiv \exists(0, \exists(C_1 \land \exists(C_2))) \equiv \exists(0, \bigvee_{p \in P} \exists(C_1 \oplus P C_2)) \equiv \exists(0, \exists(C_1 + C_2))$ because $F$ consists of the pair $C_1 \rightarrow C_1 + C_2 \leftarrow C_2$, $P$ of the pair $C_1 \leftarrow 0 \rightarrow C_2$, and $C_1 \oplus 0 C_2 = C_1 + C_2$.

(E3) is a special case of (E2)(a) since $C[u=v] = C \oplus P [u=v]$.

Additionally, there are some equivalences on nested conditions similar to the ones in \[20]\ [30].

**Lemma 3 (Equivalences).**

(E4) $\exists(a, \exists(b, c)) \equiv \exists(b \circ a, c)$

(E5) $\exists(a, c_1) \land c_2 \equiv \exists(a, c_1 \land \text{Shift}(a, c_2))$

where $a: P \rightarrow C'$, $b: C' \rightarrow C$ are morphisms, $c, c_1, c_2$ nested conditions over $C$, $C'$, and $P$, respectively and Shift denotes the construction in Lemma \[6\] on page \[30\] similar to the Shift-construction in \[7\].

**Proof.** (E4) follows directly from the definition: For any $p: P \rightarrow G$,

\[
\begin{align*}
 p &|= \exists(a, \exists(b, c)) \\
 \iff & \text{there are some } q: C' \rightarrow G, r: C \rightarrow G \text{ with } p = q \circ a, q = r \circ b \text{ and } r |= c \tag{Def. $|=\cdot$} \\
 \iff & \text{there is some } r: C \rightarrow G \text{ with } p = r \circ (b \circ a) \text{ and } r |= c \\
 \iff & p |= \exists(b \circ a, c) \tag{Def. $|=\cdot$}
\end{align*}
\]

(E5) follows with the help of Lemma \[6\] For any $p: P \rightarrow G$,

\[
\begin{align*}
 p &|= \exists(a, c_1) \land c_2 \\
 \iff & p |= \exists(a, c_1) \text{ and } p |= c_2 \\
 \iff & \text{there is some } q: C' \rightarrow G \text{ with } p = q \circ a, q |= c_1 \text{ and } q |= \text{Shift}(a, c_2) \tag{Def. $|=\cdot$, Lem. \[6\]} \\
 \iff & p |= \exists(a, c_1 \land \text{Shift}(a, c_2)) \tag{Def. $|=\cdot$}
\end{align*}
\]

\[6\] $C[u=v]$ is the graph $C$ with the nodes named $u$ and $v$ identified.
5. Translation of Essential OCL Invariants

In this section, we present the translation of Essential OCL to nested graph constraints. The translation process consists of three main steps: In a pre-processing step, OCL expressions are refactored such that a few features do no longer occur and control variables of different collection operations have distinct names. The resulting OCL invariants are then translated to nested graph conditions. The translation is performed recursively along the syntax structure of OCL expressions. The resulting graph constraints may be simplified using proven equivalences on graph conditions. The translation is illustrated at several examples. Limitations that are imposed by our translation are summarized at the end of this section.

\[
\text{OCL constraint} \xrightarrow{\text{preprocess}} \text{OCL constraint} \xrightarrow{\text{translate}} \text{graph condition} \xrightarrow{\text{simplify}} \text{graph condition}
\]

As a pre-requisite for the translation, we show in the next subsection how an object model corresponds to a node in the type graph and every association in \(M\) by bijective mappings relating every class in \(M\) to a node in the type graph, every attribute in \(M\) to an attribute node and every association in \(M\) to an edge in the type graph being named after the target role name.

### 5.1. Type and state correspondences

The correspondence of an object model \(M\) and an attributed type graph with inheritance \(ATGI\) is given by bijective mappings relating every class in \(M\) to a node in the type graph, every attribute in \(M\) to an attribute node in the type graph, and every association in \(M\) to an edge in the type graph.

#### Definition 13 (Type Correspondence).

An object model \(M\) over \(DSIG\) corresponds to an attributed type graph with inheritance \(ATGI = (TG, Z, Inh)\) with type graph \(TG = (TG_V, TG_D, TG_E, TG_A, src_G, tgt_G, src_A, tgt_A)\), final \(DSIG'\)-Algebra \(Z\) for \(DSIG' = (S \cup \text{ENUM}, OP \cup \{=\text{ENUM}, \neq\text{ENUM}\})\), and inheritance relation \(Inh\), if there is a correspondence relation \(corr_{type} = (corr_{CLASS}, corr_{ATT}, corr_{ASSOC})\) with bijective mappings

- \(corr_{CLASS} : \text{CLASS} \to TG_V\) with \(corr_{CLASS}(c) = c\) for \(c \in \text{CLASS}\) such that \(\forall c_1, c_2 \in \text{CLASS}: c_1 \prec c_2 \iff (corr_{CLASS}(c_1), corr_{CLASS}(c_2)) \in Inh\),
- \(corr_{ATT} : \text{ATT} \to TG_A\) with \(corr_{ATT}(att) = \text{att}\) for \(att : c \to s \in \text{ATT}\) such that \(src_A(corr_{ATT}(att)) = corr_{CLASS}(c)\) and \(tgt_A(corr_{ATT}(att)) = x\) if \(x \in Z\) with \(s \in S \cup \text{ENUM}\),
- \(corr_{ASSOC} : \text{ASSOC} \to TG_E\) with \(corr_{ASSOC}(a) = r_{tgt}(a)\) for \(a \in \text{ASSOC}, \text{assoc}(a) = (c, c')\), \(pr_1(a) = c, pr_2(a) = c'\) and \(c, c' \in \text{CLASS}\) such that \(src_G(corr_{ASSOC}(a)) = corr_{CLASS}(pr_1(a))\) and \(tgt_G(corr_{ASSOC}(a)) = corr_{CLASS}(pr_2(a))\). See Figure [5](left) for an illustration.

To show the correctness of our translation, we also need to establish a correspondence relation between system states being meta-model instances and typed attributed graphs. The correspondence relation is a bijection mapping from objects to nodes, attributes to attribute nodes and associations to edges in the graph.

#### Definition 14 (State Correspondence).

Let \(M\) be an object model over \(DSIG\) and \(ATGI = (TG, Z, Inh)\) an attributed type graph with inheritance \(corr_{type}(M) = ATGI\). A system state \(\sigma(M) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})\) 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 \(\text{type}\) if there is a state correspondence relation \(corr_{state} = (corr_{CLASS}, corr_{ATT}, corr_{ASSOC})\) from a system state to a typed attributed graph, defined by the bijective mappings

- \(corr_{CLASS} : \sigma_{CLASS} \to G_V\) with \(type_{G_V}(corr_{CLASS}(o)) = corr_{CLASS}(c)\) for \(o \in \sigma_{CLASS}(c)\) and \(c \in \text{CLASS}\),
\[ \sigma_{\text{Att}} : \sigma_{\text{Att}} \to G_{A} \text{ with } \text{src}_{A}(\sigma_{\text{Att}}(a)) = c_{\text{Class}}(o), \text{tgt}_{A}(\sigma_{\text{Att}}(a)) = d \text{ for } \text{type}_{G_{A}}(c_{\text{Att}}(\sigma_{\text{Att}}(a))) = \text{corr}_{\text{ATT}}(\text{att}) \text{ and } a \in \sigma_{\text{Att}}(\text{att}) \text{ if } \text{att} : c \to s \in \text{ATT}_{c}, \sigma_{\text{Att}}(\text{att}) : c_{\text{Class}}(c) \to D_{s}, o \in \sigma_{\text{Class}}(c), c \in \text{CLASS} \text{ and } \sigma_{\text{Att}}(\text{att})(o) = d, \]

\[ \sigma_{\text{Assoc}} : \sigma_{\text{Assoc}} \to G_{E} \text{ with } \text{src}_{G}(\sigma_{\text{Assoc}}) = c_{\text{Class}} \circ \text{pr}_{1} \text{ and } \text{tgt}_{G}(\sigma_{\text{Assoc}}) = c_{\text{Class}} \circ \text{pr}_{2} \text{ with } l = (o_{1}, o_{2}) \in \sigma_{\text{Assoc}}(a), \text{pr}_{1}(l) = o_{1}, \text{ and } \text{pr}_{2}(l) = o_{2}. \text{ Moreover, } \text{type}_{G_{E}}(\sigma_{\text{Assoc}}) = \text{corr}_{\text{ASSOC}}(\text{ASSOC}). \]

See Figure 8 (right) for an illustration state correspondence.

5.2. Pre-processing of OCL expressions

In order to make the translation of Essential OCL invariants as simple as possible, we perform several pre-processing steps. All steps are semantics preserving and can be regarded as refactorings on OCL expressions [31, 32].

- Removing “let” expressions and local variable calls.
  Let \( \text{expr} := \text{context self:C inv: let } v = e \text{ in } e' \) an Essential OCL invariant over an object model \( M \) with \( v \in \text{Var}_{t}, e \in \text{Expr}_{t}, e' \in \text{Expr}_{\text{Boolean}} \) and \( t \in T_{M} \). Then \( \text{expr} \) is equivalent to the Essential OCL invariant \( \text{expr}' := \text{context self:C inv: } e'' \) with \( e'' := e'[v/e] \) meaning that each occurrence of variable \( v \) in \( e' \) is substituted by (a copy of) the initializing expression \( e \) of \( v \).

- Inlining non-recursive query operation calls.
  The call of a non-recursive query operation can be replaced by (a copy of) the body expression defining the operation and replacing the context variable \( \text{self} \) by the source of the operation call.

- Replacing multiple iterators.
  Multiple iterators within a \text{forAll} iterator expression can be replaced by inserting a new \text{forAll} iterator expression for each additional iterator on the same source expression and in a nested way.

- Ensuring variable names to be unique.
  We give each variable a distinct name, e.g. \( \text{self.a->collect(v | v.b)->exists(v | expr)} \) becomes \( \text{self.a->collect(v | v.b)->exists(v' | expr)} \).

Example 5 (Pre-processing of OCL constraints).

- The Essential OCL invariant including a locally defined variable
  \[ \text{context PetriNet inv: let allPlaces:Set(Place)=Place.allInstances() in (allPlaces->forall(p1:Place | allPlaces -> forall(p2:Place | p1<>p2 implies p1.name<>p2.name))} \]
  is refactored to
  \[ \text{context PetriNet inv: Place.allInstances() ->forall(p1:Place| Place.allInstances() -> forall(p2:Place | p1 <> p2 implies p1.name <> p2.name)} \].
• The call of the (non-recursive) query operation
  context Place::noTok():Integer body: self.tokens -> size()
  within the OCL expression
  context PetriNet inv: places -> exist(p:Place | p.noTok() > 0) is refactored to
  context PetriNet inv: places -> exist(p:Place | p.tokens -> size() > 0).

• OCL invariant context PetriNet inv: self.place -> forAll(p1:Place, p2:Place | p1<>p2
  implies p1.name<>p2.name) is refactored to
  context PetriNet inv: self.place -> forAll(p1:Place | self.place -> forall(p2:Place
  | p1 <> p2 implies p1.name<>p2.name)).

5.3. Translation of Essential OCL Invariants

In the following, we present the translation of Essential OCL invariants to nested constraints. It consists of
several parts: Invariants are translated by the translation function $tr_I$, OCL expressions yielding a Boolean
value by $tr_E$, expressions yielding single objects by $tr_N$, and expressions yielding collections (i.e., sets) of
objects by $tr_S$.

<table>
<thead>
<tr>
<th>OCL expression</th>
<th>translation function</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCL invariants</td>
<td>$tr_I$</td>
</tr>
<tr>
<td>OCL expressions yielding a Boolean value</td>
<td>$tr_E$</td>
</tr>
<tr>
<td>OCL expressions yielding single objects</td>
<td>$tr_N$</td>
</tr>
<tr>
<td>OCL expressions yielding collections of objects</td>
<td>$tr_S$</td>
</tr>
</tbody>
</table>

The latter two translations take a single node as their second parameter; this node represents the object (or
the set of objects) yielded by the expression. For set expressions, this node can be matched by any object
that is member of the set; so it represents the full set. On the other hand, the resulting condition restricts
matchings to nodes inside the set.

In general, our OCL translation implements the following design decisions:

• OCL expressions are translated along their abstract syntax structure. For example, given an expression
  $a$-$\cup$($b$)$\rightarrow$notEmpty(), we translate notEmpty, then union, and then its arguments $a$ and $b$.

• The constraint header, declaring that the constraint shall hold for all instances of the context class,
is translated to a $\forall$ constraint just containing an object of the context class. The translated body is
  nested into this graph constraint.

• Set operations such as union are translated with the characteristic function of sets in mind: E.g.
  $v \in A \cup B$ if $v \in A$ or $v \in B$. An expression $expr1 \rightarrow select(v:T | expr2)$ is translated by
  yielding all elements in the translations of $expr1$ and $expr2$. We pass a single node $v$ as an extra
  parameter to the translation function $tr_S$ serving as representative of the set: $tr_S(a \rightarrow union(b), v) :=
  tr_S(a, v) \lor tr_S(b, v)$.

• The operations $\exists$ and $\forall$ are translated to their counterparts in nested graph constraints.

• Navigation expressions, which yield a single object, are treated like single-element sets.

As an introduction to OCL translation, we present an example translation. To keep track of different
constraint parts during the translation process, we mark them with different colors.

**Example 6 (The name of a transition is not empty).** The OCL invariant context Transition
inv: self.name <> '' states that transitions have non-empty names. This will be translated as follows:
The overall expression is an invariant, so we use $tr_I$:

$tr_I($context Transition inv: self.name <> '')
Invariants have to hold for all objects of the specified type. The variable self represents one object of that type, so we have

\[ \forall (\text{self}: \text{Tr}) \ tr_E(\text{self}.\text{name} <> '') \]

Then the translation proceeds at inner expressions, in this case self.name <> '. We translate along the abstract syntax tree, look at the attribute name of transitions and introduce a new node v:Tr as a representative of the rest of the expression (in this case self).

\[ \forall (\text{self}: \text{Tr}) \ \exists (v: \text{Tr}) \ tr_N(\text{self}, v: \text{Tr}) \land \exists (v: \text{Tr}) \ (\text{name} <> '') \]

In the next step, the variable expression self is translated. The node v:Tr introduced in the last step is equal to the node self:Tr, as expressed by the self=v:Tr node below.

\[ \forall (\text{self}: \text{Tr}) \ \exists (\text{self}: \text{Tr}) \Rightarrow \exists (\text{self}=\text{v:Tr}) \land \exists (\text{v:Tr} \\text{name} <> '') \]

This expression can be further simplified by applying equivalence E2a in Fact 2 to remove the conjunction, followed by E1c to remove the outer \( \exists \)-clause. The result is

\[ \forall (\text{self}: \text{Tr}) \ \exists (\text{self}: \text{Tr}) \Rightarrow \exists (\text{self}=\text{v:Tr}) \land \exists (\text{v:Tr} \\text{name} <> '') \]

**Example 7 (Set expressions).** As a more advanced example, let’s have a look at OCL expressions of the form \( s \rightarrow \exists (v: \text{Tr} | v.\text{name}='a') \) as part of an invariant. We start at the outermost part \( \exists (v: \text{Tr} | v.\text{name}='a') \) and translate this to \( \exists (v: \text{Tr} \ tr_E(\text{name}='a')) \). We next formalize that \( v: \text{Tr} \) is an element of the set described by \( s \). This is done by giving a predicate \( tr_S(s, v: \text{Tr}) \) that describes the set precisely and passes \( v: \text{Tr} \) as a parameter to \( tr_S \). The translation of the whole expression \( s \rightarrow \exists (v: \text{Tr} | v.\text{name}='a') \) becomes \( \exists (v: \text{Tr} \ tr_S(s, v: \text{Tr}) \land tr_E(\text{name}='a')) \), because \( v: \text{Tr} \) has to fulfill both \( tr_E(\text{name}='a') \) and \( tr_S(s, v: \text{Tr}) \).

**Definition 15 (Constraint translation).** Let \( M \) be an object model over DSIG with ATGI = corr_type(\( M \)). Let \( t : \text{Expr} \rightarrow T \) be a typing function which returns the type of an OCL expression. Let Invariant\_M be the set of Essential OCL invariants over \( M \) and CompactCondition\_ATGI be the set of all compact conditions as defined in Definition 11. The translation functions

\begin{align*}
\text{invariant translation} & \quad tr_I : \text{Invariant}_M \rightarrow \text{CompactCondition}_\text{ATGI} \\
\text{expression translation} & \quad tr_E : \text{Expr}_{\text{Boolean}} \rightarrow \text{CompactCondition}_\text{ATGI} \\
\text{navigation translation} & \quad tr_N : \text{Expr}_C \times \text{Graph}_\text{ATGI} \rightarrow \text{CompactCondition}_\text{ATGI} \\
\text{set translation} & \quad tr_S : \text{Expr}_{\text{Set}} \times \text{Graph}_\text{ATGI} \rightarrow \text{CompactCondition}_\text{ATGI}
\end{align*}

where \( C \in \text{CLASS} \), are defined as follows:

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

1. (a) \( tr_I(\text{context } C \ \text{inv}: \ \text{expr}) := \forall (\text{self}: C) \ tr_E(\text{expr}) \)
   (b) \( tr_I(\text{context } \text{var}: C \ \text{inv}: \ \text{expr}) := \forall (\text{var}: C) \ tr_E(\text{expr}) \)

\footnote{Rule (12b) is a slightly changed against the version in [33]: For looping navigational edges, node \( v \) has to be of a type in clan(T) \( \cap \) clan(T') instead of simply type \( T \).}
2. Translation of Boolean operators is straightforward: $tr_E(\text{true}) := true$, $tr_E(\text{not expr}) := \neg tr_E(\text{expr})$, $tr_E(\text{expr1 and expr2}) := tr_E(\text{expr1}) \land tr_E(\text{expr2})$ and likewise for or and implies.

3. Translation of existential quantifiers

(a) $tr_E(\text{exists}(v : T \mid \text{expr2})) := \exists \{v : T\} \ tr_S(\text{expr1}, v : T) \land tr_E(\text{expr2})$

(b) $tr_E(\text{exists}(v : T \mid \text{forall}(v : T \mid \text{expr2}))) := \forall \{v : T\} \ tr_S(\text{expr1}, v : T) \Rightarrow tr_E(\text{expr2})$.

4. Translation of inclusion

(a) $tr_E(\text{includesAll}(\text{expr2})) := \forall \{v : T\} \ tr_S(\text{expr2}, \{v : T\}) \Rightarrow tr_S(\text{expr1}, \{v : T\})$

(b) $tr_E(\text{excludesAll}(\text{expr2})) := \forall \{v : T\} \ tr_S(\text{expr2}, \{v : T\}) \Rightarrow \neg tr_S(\text{expr1}, \{v : T\})$

5. Translation of not

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

6. Translation of size

(a) $tr_E(\text{size}() \geq n) := \exists \{v_1 : T, \ldots, v_n : T\} \bigwedge_n tr_S(\text{expr}, v : T)$

(b) $tr_E(\text{size}() > 0) := true$

7. Translation of includes

(a) $tr_E(\text{includes}(\text{expr1} : \text{expr2})) := \exists \{v : T\} \ tr_S(\text{expr1}, \{v : T\}) \land tr_R(\text{expr2}, v : T)$

(b) if $tr_E(\text{includes}(\text{expr1} : \text{expr2})) = T$ for some class $T$, $t(\text{expr1}) = t(\text{expr2}) = \text{Set}(T)$ for some class $T$.

8. Translation of the sets described by expr1 and sets of nodes, respectively. Translations (b) and (c) are identical, since single nodes are treated as single-element sets.

$tr_E(\text{attr1 op attr2}) := \exists \{v : T\} \ tr_N(\text{expr1}, v : T) \land \exists \{v : T\} \ tr_N(\text{expr2}, v : T)$

9. Translation of attr1 op con

$\exists \{v : T\} \ tr_N(\text{expr1}, v : T) \land \exists \{v : T\} \ tr_N(\text{expr2}, v : T)$

10. Translation ofoclIsKindof(T)

(a) $tr_E(\text{oclIsKindof}(T)) := \exists \{v : T\} \ tr_N(\text{expr1}, v : T)$

(b) $tr_E(\text{oclIsTypeof}(T)) := \exists \{v : T\} \Big( v : T \Rightarrow \forall \xi \ tr_{T \cap \xi}(\xi) \Rightarrow tr_N(\text{expr1}, v : T) \Big)$

11. Translation ofoclAsType(T)

$tr_N(\text{oclAsType}(T), v : T) := \exists \{v : T\} \ tr_N(\text{expr1}, v : T)$

12. Translation ofoclAsType(T)

$tr_N(\text{oclAsType}(T), v : T) := \exists \{v : T\} \ tr_N(\text{expr1}, v : T)$

$\forall \xi \ tr_{T \cap \xi}(\xi) \Rightarrow tr_N(\text{expr1}, v : T\xi)$

Note that in this step and the following, non-strict morphisms are used.

We can express $\exists v : T \mid \text{expr2}$ as “there exist objects $v$ of type $T$, such that $v$ is contained in the set described by expr1 and $v$ satisfies expr2”, and $\text{exists}(v : T \mid \text{forall}(v : T \mid \text{expr2}))$ as “for all nodes $v$ of type $T$, if $v$ is contained in the set described by expr1 then $v$ also satisfies expr2”.$^8$

$^8$The part before $\forall$ is omitted if $\text{clan}(t(\text{expr1})) \cap \text{clan}(t(\text{expr2})) = \emptyset$, and the part after $\forall$ is omitted if $\text{expr1} = \text{expr2}$.

$^9$Case (a) presents the final step in a chain of navigations, while cases (b) and (c) present the navigation to single nodes and sets of nodes, respectively. Translations (b) and (c) are identical, since single nodes are treated as single-element sets.
13. \( t_{S}(\text{expr}1->\text{select}(v:T | \text{expr}2), \mathcal{V}:T) := \exists (v:T) \ t_{S}(\text{expr}1,v:T) \land t_{E}(\text{expr}2)\{v/v'\}) \)

The construction for \( \text{reject} \) is analogous. where \( \text{expr}2\{v/v'\} \) means replacing \( v \) in \( \text{expr}2 \) with \( v' \).

14. (a) \( t_{S}(\text{expr}1->\text{collect}(v:T | \text{expr}2), \mathcal{V}:T) := \exists (v:T) \ t_{S}(\text{expr}1,v:T) \land t_{S}(\text{expr}2,\mathcal{V}:T) \) if \( \text{expr}2 \) yields a set, and

(b) \( t_{S}(\text{expr}1->\text{collect}(v:T | \text{expr}2), \mathcal{V}:T) := \exists (v:T) \ t_{S}(\text{expr}1,v:T) \land t_{N}(\text{expr}2,\mathcal{V}:T) \) if \( \text{expr}2 \) yields an object. \(^{12}\)

15. (a) \( t_{S}(\text{expr}1->\text{union}(\text{expr}2), \mathcal{V}:T) := t_{S}(\text{expr}1,\mathcal{V}:T) \lor t_{S}(\text{expr}2,\mathcal{V}:T) \)

(b) \( t_{S}(\text{expr}1->\text{intersect}(\text{expr}2), \mathcal{V}:T) := t_{S}(\text{expr}1,\mathcal{V}:T) \land t_{S}(\text{expr}2,\mathcal{V}:T) \)

(c) \( t_{S}(\text{expr}1 - \text{expr}2, \mathcal{V}:T) := t_{S}(\text{expr}1, \mathcal{V}:T) \land \neg t_{S}(\text{expr}2, \mathcal{V}:T) \)

(d) \( t_{S}(\text{expr}1->\text{symmetricDifference}(\text{expr}2), \mathcal{V}:T) := t_{S}(\text{expr}1, \mathcal{V}:T) \not\subseteq t_{S}(\text{expr}2, \mathcal{V}:T) \) where \( c \subseteq d \) abbreviates \( c \land \neg d \lor \neg c \land d \).

16. \( t_{S}(\text{T.allInstances}(), \mathcal{V}:T) := \exists (\mathcal{V}:T) \)

17. \( t_{S}(\text{Set}[\text{expr}1, \ldots, \text{expr}N], \mathcal{V}:T) := \)

\( t_{N}(\text{expr}1, \mathcal{V}:T) \lor \cdots \lor t_{N}(\text{expr}N, \mathcal{V}:T) \)

where \( \text{expr}1, \ldots, \text{expr}N \) are OCL expressions of type \( T \).

Further translations of Essential OCL constraints can be derived from equivalences of OCL expressions. Most of these equivalences follow from basic set theory and logic axioms, cf. Richters \(^{18} \) Tables 4.4 and 4.5 and page 73.

**Definition 16 (further constraint translation).**

1. \( t_{E}(\text{expr}1->\text{includes}(\text{expr}2)) := t_{E}(\text{expr}1->\text{includesAll}(\text{Set}[\text{expr}2])) \)

2. \( t_{E}(\text{expr}1->\text{excludes}(\text{expr}2)) := t_{E}(\text{expr}1->\text{excludesAll}(\text{Set}[\text{expr}2])) \)

3. \( t_{E}(\text{expr}1 -> \text{expressing}(\text{expr}2)) := t_{E}(\text{not expr1} = \text{expr2}) \)

4. \( t_{E}(\text{expr}1 -> \text{isEmpty}()) := t_{E}(\text{not expr1} -> \text{notEmpty}()) \)

5. \( t_{E}(\text{expr} -> \text{size}() > n) := t_{E}(\text{expr} -> \text{size}() >= n+1) \)

6. \( t_{E}(\text{expr} -> \text{size}() = n) := t_{E}(\text{expr} -> \text{size}() >= n) \) and \( \text{not expr} -> \text{size}() >= n+1 \)

7. \( t_{E}(\text{expr} -> \text{size}() < n) := t_{E}(\text{not expr} -> \text{size}() > n) \)

8. \( t_{E}(\text{expr} -> \text{size}() <= n) := t_{E}(\text{not expr} -> \text{size}() > n) \)

where \( \text{expr}, \text{expr1} \) and \( \text{expr2} \) are OCL expressions and \( n \) is an integer constant.

**Examples**

In the following, we present further examples which illustrate the translation given above. For further examples, please refer to the OCL constraints in Section 2.1 and their translations to graph conditions in Section 2.3.

---

\(^{12}\)The translation \( t_{S}(\text{expr}1->\text{collect}(v:T | \text{expr}2), \mathcal{V}:T) \) (point 14) is a condition over \( v' \) that is true iff there is a node \( v \) such that (a) \( v \) is contained in the set described by \( \text{expr1} \) (i.e. \( v \) satisfies \( t_{S}(\text{expr}1, \mathcal{V}:T) \)) and (b) the relation between \( v \) and \( v' \) given by \( \text{expr2} \) is satisfied. This is described by \( t_{S}(\text{expr}2, \mathcal{V}:T) \).
Example 8 (Further invariant translations).
Distinct places of a Petri net have distinct names.

\[
\text{tr}_I(\text{context PetriNet inv:})
\]

\[
\forall (self:PN) \exists (p1:Pl, p2:Pl) \Rightarrow \exists (p1:Pl, p2:Pl) \quad \text{p1.name <> p2.name}
\]

This is equivalent to the corresponding example constraint in Section 2.3.

Example 9 (The name of a transition is not empty).  The first example we show here is Example 6, now indicating which translation rules are used.

\[
\text{tr}_I(\text{context Transition inv: self.name <> ' '}) \quad \text{1} \quad \equiv \quad \forall (self:Tr) \exists (self:Tr, name <> ' ')
\]

\[
\forall (self:Tr) \exists (self:Tr, name <> ' ')
\]

Example 10 (There is no isolated transition).

\[
\text{tr}_I(\text{context Transition inv: self.preArc->notEmpty() or self.postArc->notEmpty()}) \quad \text{1} \quad \equiv \quad \forall (self:TR) \exists (self:TR, name <> ' ')
\]

\[
\forall (self:TR) \exists (self:TR, name <> ' ')
\]
Alternatively:

\[
\begin{align*}
tr_I(\text{context Petrinet} \ inv: & \ \text{self.transition}\rightarrow\forall t:\text{Transition} \ | \\
& \ t.\text{preArc}\rightarrow\text{notEmpty()} \ \text{or} \ t.\text{postArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:PN}, trE(\text{self.transition}\rightarrow\forall t:\text{Transition} | \\
& \ t.\text{preArc}\rightarrow\text{notEmpty()} \ \text{or} \ t.\text{postArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:PN}, trS(\text{self.transition, t.Tr}) \Rightarrow \\
& trE(t.\text{preArc}\rightarrow\text{notEmpty()} \ \text{or} \ t.\text{postArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:PN}, t.Tr, trN(u.PN, t.N) \cap \exists (u.PN\rightarrow v1.TArc)) \Rightarrow \\
& trE(t.\text{preArc}\rightarrow\text{notEmpty()} \ \text{or} \ t.\text{postArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:PN}, t.Tr, \exists (u.PN\rightarrow v1.TArc) t.N) \Rightarrow \\
& \exists v1.TArc, trN(t.\text{preArc}, v1.TArc)) \vee \\
& \exists v2.TArc, trN(t.\text{postArc}, v2.TArc)) \equiv \ E3 \\
\forall (\text{self:PN}, t.Tr, \exists (u.PN\rightarrow v1.TArc) t.N) \Rightarrow \\
& \exists v1.TArc, \exists (w1.TTr, \exists (w1.TTr\rightarrow v1.TArc)) \vee \\
& \exists v2.TArc, \exists (w2.TTr, \exists (w2.TTr\rightarrow v2.TArc)) \equiv \ E1b \\
\forall (\text{self:PN}, t.Tr, \exists (u.PN\rightarrow v1.TArc) t.N) \Rightarrow \\
& \exists v1.TArc, \exists (w1.TTr, \exists (w1.TTr\rightarrow v1.TArc)) \vee \\
& \exists v2.TArc, \exists (w2.TTr, \exists (w2.TTr\rightarrow v2.TArc)) \equiv \ E3 \equiv \ E1b \\
\forall (\text{self:PN}, t.Tr, \exists (u.PN\rightarrow v1.TArc) t.N) \Rightarrow \\
& \exists v1.TArc, \exists (w1.TTr, \exists (w1.TTr\rightarrow v1.TArc)) \vee \\
& \exists v2.TArc, \exists (w2.TTr, \exists (w2.TTr\rightarrow v2.TArc)) \equiv \ E3 \equiv \ E1b \\
\end{align*}
\]

Example 11 (There is no isolated place).

\[
\begin{align*}
tr_I(\text{context Place} \ inv: & \ \text{self.preArc}\rightarrow\text{notEmpty()} \ \text{or} \ \text{self.postArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:Pl}, trE(\text{self.preArc}\rightarrow\text{notEmpty()} \ \text{or} \ \text{self.postArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:Pl}, trE(\text{self.preArc}\rightarrow\text{notEmpty()})) \equiv {} \\
\forall (\text{self:Pl}, trS(\text{self.preArc, v.TArc})) \equiv \ E1b \\
\forall (\text{self:Pl}, \exists (v.TArc, trS(\text{self.preArc, v.TArc}))) \equiv \ E1b \\
\forall (\text{self:Pl}, \exists (v.TArc, \exists (\text{self.preArc}\rightarrow v.TArc))) \equiv \ E1b \\
\forall (\text{self:Pl}, \exists (v.TArc, \exists (\text{self.postArc}\rightarrow w.TArc))) \equiv \ E1b \\
\end{align*}
\]
Example 12 (Each two places of a Petri net have different names).

\[
\forall (\text{self.PN} \quad tr_E(\text{self.place} \Rightarrow \forall p(\text{Place} \Rightarrow \forall (p2:\text{Place} \Rightarrow \text{self.name} \neq p2.\text{name}) ) )) \quad 2 \times 3 \quad 20
\]

\[
\forall (\text{self.PN} \quad \forall (p1:Pl) \quad tr_S(\text{self.place} \Rightarrow \forall (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \text{tr_E}(\text{p1} \neq \text{p2} \Rightarrow \text{self.name} < \text{p1}.\text{name} \neq \text{p2}.\text{name})))) \quad 2 \times 1 \times 2 \quad 20
\]

\[
\forall (\text{self.PN} \quad \forall (p1:Pl) \quad \exists (\text{self.PN} \quad \exists (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \text{tr_E}(\text{p1} \neq \text{p2} \Rightarrow \text{self.name} < \text{p1}.\text{name} \neq \text{p2}.\text{name})))) \quad 2 \times 1 \times 2 \times 2 \times 1 \times 2 \quad 40
\]

\[
\forall (\text{self.PN} \quad \forall (p1:Pl) \quad \forall (p2:Pl \Rightarrow \forall (\text{self.PN} \quad \exists (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \\
\text{tr_E}(\text{p1} \neq \text{p2} \Rightarrow \text{self.name} < \text{p1}.\text{name} \neq \text{p2}.\text{name})))) \quad 7 \times 9 \quad 63
\]

To simplify this lax condition, we transform it to a nested condition.

\[
\neg \exists (\text{self.PN} \quad \exists (\text{self.PN} \quad \exists (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \exists (\text{self.PN} \quad \exists (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \\
\text{tr_E}(\text{p1} \neq \text{p2} \Rightarrow \text{self.name} < \text{p1}.\text{name} \neq \text{p2}.\text{name})))) \quad 7 \times 9 \quad 63
\]

This constraint can be transformed back to a lax condition.

\[
\forall (\text{self.PN} \quad \exists (\text{self.PN} \quad \exists (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \exists (\text{self.PN} \quad \exists (p1:Pl \Rightarrow \forall (p2:Pl \Rightarrow \\
\text{tr_E}(\text{p1} \neq \text{p2} \Rightarrow \text{self.name} < \text{p1}.\text{name} \neq \text{p2}.\text{name})))) \quad 7 \times 9 \quad 63
\]

30
Example 13 (There is at least one place in a Petri net having at least one token).

\[ \text{tr}_I(\text{context PetriNet inv: } \text{self.place} \rightarrow \text{exists}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}())) \]

\[ \forall (\text{self: PN}) \text{tr}_E(\text{self.place} \rightarrow \text{exists}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}())) \]

\[ \forall (\text{self: PN}) \exists (p: \text{Pl}) \text{tr}_S(\text{self.place}, p: \text{Pl}) \land \exists (t: \text{Tk}) \text{tr}_S(p.\text{token}, t: \text{Tk})) \]

\[ \forall (\text{self: PN}) \exists (p: \text{Pl}) \exists (p: \text{Pl} \rightarrow \text{notEmpty}) \land \exists (t: \text{Tk}) \text{tr}_S(p.\text{token}, t: \text{Tk})) \]

Alternatively:

\[ \text{tr}_I(\text{context PetriNet inv: } \text{self.place} \rightarrow \text{select}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}()) \rightarrow \text{notEmpty}()) \]

\[ \forall (\text{self: PN}) \text{tr}_E(\text{self.place} \rightarrow \text{select}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}()) \rightarrow \text{notEmpty}()) \]

\[ \forall (\text{self: PN}) \exists (p: \text{Pl}) \text{tr}_S(\text{self.place} \rightarrow \text{select}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}())), p: \text{Pl}) \]

\[ \forall (\text{self: PN}) \exists (p: \text{Pl}) \exists (p: \text{Pl} \rightarrow \text{notEmpty}) \land \exists (t: \text{Tk}) \text{tr}_S(p.\text{token}, t: \text{Tk})) \]

Alternatively:

\[ \text{tr}_I(\text{context PetriNet inv: } \text{self.place} \rightarrow \text{collect}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}()) \rightarrow \text{notEmpty}()) \]

\[ \forall (\text{self: PN}) \text{tr}_E(\text{self.place} \rightarrow \text{collect}(p: \text{Place}|p.\text{token} \rightarrow \text{notEmpty}()) \rightarrow \text{notEmpty}()) \]

\[ \forall (\text{self: PN}) \exists (v: \text{Tk}) \text{tr}_S(\text{self.place} \rightarrow \text{collect}(p: \text{Place}|p.\text{token})), v: \text{Tk}) \]

Alternatively:

\[ \text{tr}_I(\text{context PetriNet inv: } \text{Token.allInstances}() \rightarrow \text{notEmpty}()) \]

\[ \forall (\text{self: PN}) \text{tr}_E(\text{Token.allInstances}() \rightarrow \text{notEmpty}()) \]

\[ \forall (\text{self: PN}) \exists (v: \text{Tk}) \text{tr}_S(\text{Token.allInstances}(), v: \text{Tk}) \]

**Remark 13:** The resulting nested graph constraint slightly differs from those presented above, but is equivalent under the assumption that a token always has to be in a place which must be in a Petri net.
Example 14 (The weight of an arc is positive).

\[ tr_I(context \text{ Arc inv: self.weight} \geq 1) \]
\[ \forall (\text{self:Arc}, tr_E(self.weight) \geq 1) \equiv \]
\[ \forall (\text{self:Arc}, \exists (v:Arc, tr_N(self, v:Arc) \land \exists (v:Arc, v.weight \geq 1))) \equiv E^3 \]
\[ \forall (\text{self:Arc}, \exists (v:Arc, v.weight \geq 1)) \]

Example 15 (Each Petri net has at least two places).

\[ tr_I(context \text{ Petrinet inv: self.place->size()} \geq 2) \]
\[ \forall (\text{self:PN}, tr_E(self.place->size()) \geq 2) \equiv \]
\[ \forall (\text{self:PN}, \exists (v1:Pl, v2:Pl, tr_S(self.place, v1:Pl) \land tr_S(self.place, v2:Pl))) \equiv E^2 \]
\[ \forall (\text{self:PN}, \exists (v1:Pl, v2:Pl, \exists (self:PN v1:Pl.place)) \land \exists (self:PN v2:Pl.place)) \equiv E^3 \]
\[ \forall (\text{self:PN}, \exists (v1:Pl, v2:Pl)) \]

Lemma 4 (Translations of Core and Essential OCL). The translation of Core OCL constraints in [6] is closely related to the translation of Essential OCL constraints presented in this paper: Every Core OCL constraint translated as in [6] yields a nested graph constraint which is equivalent to the translation result in this paper.

**Proof.** Let \( tr' \) denote the translations of Core OCL constraints in [6] and \( tr \) denote the translation of Essential OCL constraints in this paper. The proof is done according to the items in [6, Definition 12] and uses the definition of \( tr' \), the equivalences of Fact 2, and the definition of \( tr \). Moreover, the proof for item (11–12) makes use of an induction over the structure of Core OCL constraints.

In more detail: Items 1 and from \( tr' \) correspond to item 1 from \( tr \), items 4–8 correspond to 2, 16 and 17 from \( tr' \) to 9 and 8 in \( tr \), respectively. The other items are proved as follows; item numbers are taken from Definition 12 in [6].

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

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

\[ tr'_E(v.role->notEmpty()) \]
\[ \exists(tr'_N(v.role)) = \] (Def. \( tr'_E \))
\[ \exists(\emptyset \rightarrow v:T \rightarrow v'.T') \equiv \] (Def. \( tr'_N \))
\[ \exists(\emptyset \rightarrow v:T \rightarrow (v:T \rightarrow v'.T')) = \] (Def. \( tr_N \))
\[ \exists(\emptyset \rightarrow v:T \rightarrow tr_N(v.role, v'.T')) = \] (Def. \( tr_E \))

The proof for isNotEmpty is analogous.

(11,12) This is proven by induction over the structure of OCL constraints.

Induction base: \( tr'_E(true) = true = tr_E(true) \).

Hypothesis: For sub-constraints \( expr, tr'_E(expr) = tr_E(expr) \).
\( tr'_E(v \rightarrow u) = \)  
\( \{ tr'_N(u \rightarrow v) \land tr'_E(expr) \} \)  
(\( \text{Def. Shift}\))

 Since we focus on the use of OCL within DSML definitions, we restrict our translation to \( invariants \).

### 5.4. Limitations

Since nested graph constraints are restricted to a \( first-order, two-valued logic \), our OCL translation is straightened to corresponding OCL features, focusing on the equivalence of constraints to \( true \) in our proofs. Therefore, we do not consider expressions related to hierarchical sets (e.g., \( \text{Sequence} \)) such as \( \text{sortedBy} \) and \( \text{isUnique} \) as well as expressions related to flattened sets of primitive values (e.g., \( \text{sum} \)).

Since nested graph constraints are restricted to a \( first-order, two-valued logic \), our OCL translation is straightened to corresponding OCL features, focusing on the equivalence of constraints to \( true \) in our proofs. Therefore, we do not consider expressions related to hierarchical sets (e.g., \( \text{Sequence} \)) such as \( \text{sortedBy} \) and \( \text{isUnique} \) as well as expressions related to flattened sets of primitive values (e.g., \( \text{sum} \)).

\(^{13}\text{This is a silent assumption in } [2]; \text{ indeed, } tr'_N \text{ does not work for models with direct loops in the meta-model.} \)
The translation of If-expressions is restricted to those which consist of Boolean expressions only. The size operation may only be used in comparison with constants, since it is expressed structurally in graph constraints. Furthermore, we do not consider iterate, which is not first-order. Finally, OperationCall expressions are considered only if underlying operations are not user-defined.

6. Correctness and completeness of the translation

In this section, we show the correctness and completeness of the translation tr given in Definition 15. In order to do this, we need to recall the formal semantics for the Essential OCL invariants we translate, as given in the OCL specification [1].

6.1. Semantics of Essential OCL invariants

The essence of OCL semantics is recalled from the doctoral thesis by Richters [18]. We prefer this formalization, in contrast to the UML-based specification, since it is more suitable for proving the semantics preservation of our translation.

Definition 17 (Semantics of a Signature). Let $\Sigma = (T_M, \preceq_M, \Omega_M)$ be a signature over an object model $M$. The semantics of $M$ is a structure $I(\Sigma_M) = (I(T_M), I(\preceq_M), I(\Omega_M))$ where

- $I(T_M)$ assigns to each type $t \in T_M$ an interpretation $I(t)$, e.g., $I(\text{Real}) = \mathbb{R}$, $I(t_c) = \sigma^c_{\text{Class}}(c)$ for class $c$, and $I(\text{Set}(t))$ is the set of all finite subsets of $I(t)$.
- $I(\preceq_M)$ implies for all types $t, t' \in T_M$ that $I(t) \subseteq I(t')$ if $t \preceq_M t'$,
- and $I(\Omega_M)$ assigns each operation $\omega : t_1 \times \cdots \times t_n \rightarrow t \in \Omega_M$ a total function $I(\omega) = I(t_1) \times \cdots \times I(t_n) \rightarrow I(t)$, e.g., $I(\text{42}) = 42$, $I(\text{+Integer})(i, j) = i + j$ for integers $i$ and $j$, $I(\text{att}: t_c \rightarrow t) = \sigma_{\text{Att}}(\text{att})(c')$ with $c' \in \sigma_{\text{Class}}(c)$, and $I(c': t_c \rightarrow \text{Set}(t_c')) = \{c' | (c, c') \in \sigma_{\text{Assoc}}(a)\}$.

With the help of the signature semantics, we now recall the semantics of Essential OCL expressions, laying the basis for the correctness proof of our OCL translation.

Definition 18 (Semantics of Essential OCL Expressions). Let $\Sigma_M = (T_M, \preceq_M, \Omega_M)$ be a signature over an object model $M$. Let $\text{Var} = \{\text{Var}_t\}_{t \in T_M}$ be a family of variable sets indexed by types $t \in T_M$. Let $\text{Env} = \{\tau \ | \ \tau = (\sigma, \beta)\}$ be a set of environments with system states $\sigma$ and variable assignments $\beta : \text{Var}_t \rightarrow I(t)$ which map variable names to values. The semantics of an Essential OCL expression $e \in \text{Expr}_t$ is a function $I[e] : \text{Env} \rightarrow I(t)$ and is defined recursively as given in Tables 1 and 2 for each $\tau = (\sigma, \beta) \in \text{Env}$.

Remark 2. An invariant context $v : C \ \text{inv: expr}$ can be expressed by $\text{C.allInstances->forAll}(v\text{expr})$ [36] p. 213]. Therefore, the semantics of this invariant is equal to the semantics of the corresponding Essential OCL expression.

6.2. Main theorem and proof

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

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

$I[\text{inv}] \|(\sigma, \beta) = \text{true} \Leftrightarrow G \models tr_I(\text{inv})$. 

34
The proof proceeds by induction over the items of Definition 15. The semantics of the respective OCL expressions is taken from Definition 18 and then reformed until it fits the semantics of the graph condition according to Definition 10.

**Proof.** We prove three statements concerning the translation functions $tr_N$, $tr_E$, and $tr_S$, respectively:

Given an object model $M$ with attributed type graph $corrType(M)$, for all Essential OCL expressions $expr$ in the domain of $tr_N$, $tr_E$, and $tr_S$, all environments $(\sigma, \beta) \in Env$ and morphism $p$ from some graph $P$ graph to $G = corrState(\sigma)$,

\[ I[expr](\sigma, \beta) = true \iff p |\triangleright tr_E(expr) \]

\[(i) \quad I[expr](\sigma, \beta) = v \iff p \circ id_v |\triangleright tr_N(expr, v:T) \]

\[(ii) \quad I[expr](\sigma, \beta) = V \iff \forall v \in V. p \circ id_v |\triangleright tr_S(expr, v:T) \]

The domain of $p$ resp. $p \circ id_v$ contains a node $v$ with name $v$ and $p(v) = c_{Class}(\beta(v))$. By Remark 2 and the definition of $tr$, for invariants $inv = context C inv: expr$ and morphisms $p: \emptyset \rightarrow G$,

\[ I[inv](\sigma, \beta) = true \rightarrow C.allInstances->forAll(self|expr) \]

\[ \iff p |\triangleright \forall(v:T) tr_E(expr) \rightarrow p |\triangleright tr_I(inv) \rightarrow G |\triangleright tr_I(inv). \]

**Base Case.** $I[context C inv: true](\sigma, \beta) = true = \forall v \in \sigma_{Class}(C).true = \forall(v:T) true = tr_I(context C inv: true).$

**Hypothesis.** For all subexpressions $expr$, objects $v, v_1, \ldots, v_n$ and morphisms $p: \{v:T\} \in c_{Class}(\beta(v)) \rightarrow corrState(\sigma)$, let statements (i), (ii) and (iii) be true.

**Induction Step.**

(1) Let $texpr1 = T$. Then $I[expr1->exists(v:T) expr2](\sigma, \beta) = \true$. \hspace{1cm} (Def. 18)

(2) Since Boolean operators in OCL have corresponding operators in conditions, proofs are straightforward.

(3) Let $t(expr_1) = t$. Then $I[expr1\rightarrow exists(v:T) expr2](\sigma, \beta) \iff I[expr1](\sigma, \beta) = \{v_1, \ldots, v_n\} \land \\bigvee_{1 \leq i \leq n} I[expr2](\sigma, \beta\{v/v_i\})$ (set axioms)

(4) Let $t(expr_1) = t(expr_2) = \true$. Then $I[expr1\rightarrow exists(v:T) expr2](\sigma, \beta)$ (Ind. hyp.)

The proof of $forall$ is analogous.

---

14For morphisms $p: P \rightarrow G$, let function composition $p \circ id_v$ be the morphism $p': P \circ v:T \rightarrow G$, with $p'(v) = p(v)$ if $v \in \text{Dom}(p)$ and $p'(v) = v$ otherwise; we tag the node $v$ in the domain of $p'$ with name $v$. Note that $P = \emptyset$ for constraints.
I[expr1->includesAll(expr2)]((σ,β) ⇔ I[expr2][(σ,β) ⊆ I[expr1]](σ,β)  
(set axioms)
⇒ ∀v ∈ σ(T).v ∈ I[expr2][σ,β] implies v ∈ I[expr1][σ,β]  
(Ind. hyp.)
⇒ [v:T] ∈ cClass(σ(T)).p ⊔ id_v = trS(expr2, v:T) implies p ⊔ id_v = trS(expr1, v:T)  
(Def. |=)
⇒ p = [v:T] trS(expr2, v:T) implies trS(expr1, v:T))  
(Def. 15.4)
⇒ p = tr_E(expr1->includesAll(expr2))  

The proof of excludesAll is analogous.

(5) I[expr->notEmpty()]((σ,β) ⇔ I[expr]((σ,β)) ≠ ∅  
(set axioms)
⇒ ∃v ∈ σClass(T).v ∈ I[expr][σ,β]  
(Ind. hyp.)
⇒ [v:T] ∈ cClass(σClass(T)).p ⊔ id_v = trS(expr, v:T)  
(Def. |=)
⇒ p = [v:T] trS(expr, v:T)  
(Def. 15.5)
⇒ p = tr_E(expr->notEmpty())  

(6) I[expr->size()] > n[(σ,β) ⇔ |{v | I[expr](σ,β)}| ≥ n  
(set axioms)
⇒ ∃v_1, ..., v_n ∈ σ(T). ∨_i=1 to n (v_i ≠ v_j) ∧ (v_i ∈ I[expr][σ,β])  
(Ind. hyp.)
⇒ [v:T] · · · [v:T] ∈ cClass(σClass(T)). ∨_i=1 to n (p ⊔ id_v_i = trS(expr, v:T))  
(Def. |=)
⇒ p = [v:T] · · · [v:T] _i=1 to n trS(expr, v:T))  
(Def. 15.6)
⇒ p = tr_E(expr->size())  

(7a) For t(expr1) = t(expr2) = T for some class T,
I[expr1 = expr2][(σ,β) ⇔ I[expr1][σ,β] = I[expr2][σ,β]  
(use variable)
⇒ ∀v ∈ σClass(T).v = I[expr1][σ,β] ∧ v ∈ I[expr2][σ,β]  
(Ind. hyp.)
⇒ [v:T] ∈ cClass(σClass(T)).p ⊔ id_v = trS(expr1, v:T) ∧ p ⊔ id_v = trS(expr2, v:T)  
(Def. |=)
⇒ p = [v:T] trS(expr1, v:T) ∧ trS(expr2, v:T))  
(Def. 15.7)
⇒ p = tr_E(expr1 = expr2)  

(7b) For t(expr1) = t(expr2) = Set(T) for some class T,
I[expr1 = expr2][(σ,β) ⇔ I[expr1][σ,β] = I[expr2][σ,β]  
(set axioms)
⇒ ∀v ∈ σClass(T).v = I[expr1][σ,β] ∧ v ∈ I[expr2][σ,β]  
(Ind. hyp.)
⇒ [v:T] ∈ cClass(σClass(T)).p ⊔ id_v = trS(expr1, v:T) if p ⊔ id_v = trS(expr2, v:T)  
(Def. |=)
⇒ p = [v:T] trS(expr1, v:T) if trS(expr2, v:T))  
(Def. 15.7)
⇒ p = tr_E(expr1 = expr2)  

(8) I[v.attr op x] ⇔ I[(op)(S,β,I[v.attr])(S,β), I(x)(S,β)]  
(I[expr](S,β)(σ_{At}(attr(β)(v)), x)  
(Ind. hyp.)
⇒ ∃q_1, q_2, p = q_2 op q_1 ∧ (q_1)  
(v:T) attr op x) → Dom(q_2)) jointly surj. inclusions  
(Def. |=, Def. 15.9)
⇒ p = ∃[v:T] attr op x = tr_E(v.attr op x)  

(9) Let T=T1(expr1), T'= t(expr2), att(v, att)=σ_{Att}(att)(I[v](σ,β)), p_v=p ⊔ id_v, p_{v'}=p ⊔ id_{v'}.  
I[ex1.a1 op ex2.a2][σ,β] ⇔ att(ex1.a1) op att(ex2.a2)  
(Def. 18)
⇒ ∃v, v', v = I[ex1][σ,β] ∧ v' = I[ex2][σ,β] ∧ att(v, a1) op att(v', a2)  
(Ind. hyp.)
⇒ :∃[v:T] [v:T] p_v = (tr_{ex1}(ex1, v:T) ∧ ∃(v:T)) ∧ p_{v'} = (tr_{ex2}(ex2, v:T) ∧ ∃(v:T)  
(Equiv. 2)
⇒ [v:v:T] p_v = tr_{ex1}(ex1, v:T ∧ p_{v'} = tr_{ex2}(ex2, v:T)  
(Def. |=)
⇒ p = ∃[v:v:T] tr_{ex1}(ex1, v:T) ∧ tr_{ex2}(ex2, v:T)  
(Def. 15.9)
⇒ p = tr_E(ex1.a1 op ex2.a2)
(10) Let $t(expr) = T'$ and $T \in \text{clan}(T')$.

$I[\text{oclIsTypeOf}(T)](\sigma, \beta) \iff I[\text{oclAsType}(\sigma, \beta) \land I[\text{oclAsType}(\sigma, \beta) \in I(T)]$ (set axioms)

$\iff \exists v = I[\text{oclAsType}(\sigma, \beta)].v \in I(T) \land \bigwedge_{T'' \leq M} v \notin I(T'')$ (Def. 3, 2)

$\iff \exists v = I[\text{oclAsType}(\sigma, \beta) \land I[\text{oclAsType}(\sigma, \beta) \in I(T)]$ (Ind. hyp.)

$\iff \exists v = I[\text{oclAsType}(\sigma, \beta) \land I[\text{oclAsType}(\sigma, \beta) \in I(T)]$ (Def. 15, 10)

$\iff p \models \exists v = I[\text{oclAsType}(\sigma, \beta) \land I[\text{oclAsType}(\sigma, \beta) \in I(T)]$ (Def. 15, 10)

(The proof is analogous to the one for $\text{oclIsTypeOf}(T)$ (without the $\cup$-part).

(11) Let $t(expr) = T'$.

$v = I[\text{oclAsType}(\sigma, \beta) \iff I[\text{oclAsType}(\sigma, \beta) \land I[\text{oclAsType}(\sigma, \beta) \in I(T)]$ (Def. 3, 2)

$\iff v = I[\text{oclAsType}(\sigma, \beta) \land v \in I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\iff v = I[\text{oclAsType}(\sigma, \beta) \land v \in I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 11)

(12a) Let $t(v) = t(v') = T$.

$v' = I[v](\sigma, \beta) \iff \exists v' \in I(T), \beta(v) = v'$ (cClass)

$\iff v' = I[v'] \in I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 10)

$\iff \exists v' \in I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 10)

(12b) Let $\xi = T \cap T'$.

$v = I[\text{oclAsType}(\sigma, \beta) \iff I[\text{oclAsType}(\sigma, \beta) \in I[\text{oclAsType}(\sigma, \beta)$ (Def. 3)

$\iff \exists v' = I[\text{oclAsType}(\sigma, \beta) \land v \in I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\iff \exists v' = I[\text{oclAsType}(\sigma, \beta) \land v \in I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 12)

The proof of the $trS$ cases is analogous to the $trN$ cases.

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

$v = I[\text{oclAsType}(\sigma, \beta) \iff I[\text{oclAsType}(\sigma, \beta) \in I[\text{oclAsType}(\sigma, \beta)$ (set axioms)

$\iff v \in I[\text{oclAsType}(\sigma, \beta) \land v \notin I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\iff v \in I[\text{oclAsType}(\sigma, \beta) \land v \notin I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\iff v \in I[\text{oclAsType}(\sigma, \beta) \land v \notin I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 13)

The proof for $\text{rej}$ is analogous.

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

$v = I[\text{oclAsType}(\sigma, \beta) \iff I[\text{oclAsType}(\sigma, \beta) \in I[\text{oclAsType}(\sigma, \beta)$ (set axioms)

$\iff v \in I[\text{oclAsType}(\sigma, \beta) \land v \notin I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\iff v \in I[\text{oclAsType}(\sigma, \beta) \land v \notin I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\iff v \in I[\text{oclAsType}(\sigma, \beta) \land v \notin I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 14)

The proof for $\text{oclAsType}$ yielding an object is analogous.

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

$\sigma, \beta, v \in I[\text{oclAsType}(\sigma, \beta) \land \forall v \notin I[\text{oclAsType}(\sigma, \beta)$ (set axioms)

$\sigma, \beta, v \in I[\text{oclAsType}(\sigma, \beta) \land \forall v \notin I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\sigma, \beta, v \in I[\text{oclAsType}(\sigma, \beta) \land \forall v \notin I[\text{oclAsType}(\sigma, \beta)$ (Ind. hyp.)

$\sigma, \beta, v \in I[\text{oclAsType}(\sigma, \beta) \land \forall v \notin I[\text{oclAsType}(\sigma, \beta)$ (Def. 15, 14)

The proof for $\text{oclAsType}$ yielding an object is analogous.
yielding a graph $g$ as follows: Select an injective morphism $L$ and an application condition $ac_L$ over $L$. The application of the rule $g$ to a graph $G$ means to select an occurrence of $L$ in $G$, (i.e., an injective morphism $g: L \rightarrow G$), check the dangling condition “No edge in $G\setminus g(L)$ is incident to a node in $g(L-K)$” and the left application condition $ac_L$, delete $g(L-K)$ from $G$ and to add a fresh copy of $R-K$. Several examples of direct graph transformations have been shown in Section 2.6 above.

**Definition 19 (Rules and direct transformations).** A rule $g = (p, ac_L)$ consists of a plain rule $p = (L \leftarrow K \rightarrow R)$ with two injective, type-strict morphisms $l: K \rightarrow L$ and $r: K \rightarrow R$ and an application condition $ac_L$ over $L$. A graph $G$ directly derives $H$ by $g$ and $g$ if $H$ is isomorphic to the graph constructed as follows: Select an injective morphism $g: L \rightarrow G$ and check the dangling condition “No edge in $G\setminus g(L)$ is incident to a node in $g(L-K)$” and the left application condition $ac_L$, delete $g(L-K)$ from the graph $G$, yielding a graph $D$, and add $R-K$ to $D$. We write $G \Rightarrow_{g,h} H$ or $G \Rightarrow_{g} H$ if there exists such a direct transformation.
7.2. From graph constraints to right application conditions

In this subsection, we sketch how constraints are shifted over morphisms to right application conditions. The corresponding transformation is denoted by $\text{Shift}$.

A constraint holds over the empty graph, i.e., does not need any premise to hold. Shifting a constraint from the empty graph to the right-hand side of a rule adjusts the constraint to the right-hand side by adding items (nodes and edges) to every graph in the constraint, such that the resulting condition is a right application condition for the considered rule.

Shifting a constraint $\exists \left( P \xrightarrow{a} C, c \right)$ over an injective morphism $b: P \to P'$ means to calculate all possible overlappings $C'$ of $P'$ and $C$, and to shift the subcondition $c$ over the induced injective morphism $b'$ yielding a condition $c'$. Hence, shifting the constraint $\exists \left( P \xrightarrow{a} C, c \right)$ results in a right application condition $\bigvee_{i \in I} \exists \left( a'_i, c'_i \right)$.

Lemma 6 (Shift of conditions over morphisms [27]). There is a $\text{Shift}$ construction such that, for each condition $c$ over $P$ and for each morphism $b: P \to P'$, $\text{Shift}$ transforms $c$ via $b$ into a condition $\text{Shift}(b, c)$ over $P'$ such that, for each morphism $n$ with domain $P'$,

$$n \circ b |\approx c \iff n |\approx \text{Shift}(b, c).$$

Lemma 6 is an injective version of [27, Lemma 2] where arbitrary conditions with arbitrary matching ($n: P' \to H$ arbitrary) are shifted over arbitrary morphisms. In Lemma 6, injective conditions with injective matching ($n: P' \to H$ injective) are shifted over injective morphisms.

Construction. The $\text{Shift}$ construction is inductively defined as follows:

$$\text{Shift}(b, \text{true}) = \text{true},$$

$$\text{Shift}(b, \exists(a, c)) = \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}(b', c))$$

where

$$\mathcal{F} = \{(a', b') \mid (a', b') \text{ jointly surjective, } a', b' \text{ injective, (1) commutes}\}$$

$$\text{Shift}(b, \neg c) = \neg \text{Shift}(b, c), \text{Shift}(b, \land c_i) = \land c_i \text{Shift}(b, c_i).$$

Proof. By inspection of the proof of [27, Lemma 2]. The Only-if case $n \models \text{Shift}(b, c) \Rightarrow n \circ b \models c$ follows from [27] when restricting $\mathcal{M}$ to type-strict morphisms only. Because injective morphisms are closed under decomposition, $q, n, m$ injective implies $a', b'$ injective. Thus, $(a', b') \in \mathcal{F}$. The If case follows with $b', m$ being arbitrary injective morphisms. 

Example 16 (From graph constraints to right application conditions). Consider the rule $p = \text{createPN}(n: \text{String})$ which creates a Petri net with name $n$ and the graph constraint $c_1$ stating that every Petri net has at least one token.

$$p = \langle \emptyset \leftarrow \emptyset \rightarrow \text{pPN} \rangle \text{name} = n$$

$$c_1 = \forall \text{1:PN} \exists \text{1:PN token} \land \text{Tk}$$

Shifting the constraint $c_1$ over the morphism $a$ from the empty graph to the right-hand side of the rule $p$
yields the right application condition \( \text{Shift}(a, c_1) = rc_{c_11} \land rc_{c_12} \) with the subconditions

\[
rc_{c_11} = \forall (p:PN \text{name}=n) \rightarrow 1.p:PN \text{name}=n \exists (\exists (p:PN \text{name}=n) \rightarrow 1.p:PN \text{name}=n)
\]

\[
rc_{c_12} = \forall (p:PN \text{name}=n) \rightarrow 1.p:PN \text{name}=n \exists (\exists (p:PN \text{name}=n) \rightarrow 1.p:PN \text{name}=n)
\]

While \( rc_{c_11} \) requires that the newly created Petri net has at least one token, \( rc_{c_12} \) requires this for all other Petri nets in the result graph of the considered transformation.

The construction is illustrated in Figure 9

1. Draw the constraint vertically on the left-hand side. Decorate the morphisms of the constraint by quantors: if \( a \) is the morphism of the subcondition \( Q(a, c) \) with \( Q \in \{ \forall, \exists \} \), then \( a \) is decorated by \( Q \).
2. Draw the right-hand side of the rule on the right-hand side, on the level of the empty graph.
3. Complete the rows of the figure by constructing all possible overlappings of the corresponding graph of the constraint and the graphs of the previous row (as well as their morphisms).
4. The vertical morphisms on the right are decorated by quantors as their corresponding morphisms on the left.
5. Connect the \( \exists (\forall) \)-decorated morphisms outgoing from the same graph by \( \lor (\land) \).
6. Read the application condition over the right-hand side from the completed figure.

![Figure 9: Construction of \( \text{Shift}(a, c_1) \).](image)

7.3. From right to left application conditions

In this subsection, we sketch how right application conditions can be shifted over rules to left application conditions. The corresponding transformation is denoted by \( \text{Left} \). The transformation \( \text{Left} \) is based on rule application, more precisely, the application of the inverse rule.

The application of \( \text{Left} \) to the right application condition \( \exists (a, ac) \) yields a left application condition \( \exists (a', ac') \) where the graph \( L' \) is obtained from the graph \( R' \) by applying the inverse rule \( \langle R \leftarrow K \rightarrow L \rangle \) to \( R' \) according to the injective morphism \( R \rightarrow R' \) provided that the dangling condition is met. The left application condition \( ac' \) is obtained by shifting the subcondition \( ac \) over the induced rule \( \langle L' \leftarrow K' \rightarrow R' \rangle \). If the inverse rule is not applicable to a right condition, the resulting left condition is \( \text{false} \).

40
Lemma 7 (shift of conditions over rules [7]). There is a construction \( \text{Left} \) such that, for each rule \( p = (L \leftarrow K \rightarrow R) \) and each condition \( \text{ac} \) over \( R \), \( \text{Left} \) transforms \( \text{ac} \) via \( p \) into a condition \( \text{Left}(p, \text{ac}) \) over \( L \) such that, for each direct transformation \( G \Rightarrow p,g,h \ H \), we have \( g \models \text{Left}(p, \text{ac}) \iff h \models \text{ac} \).

\[
\begin{array}{c}
L \leftarrow K \rightarrow R \xrightarrow{\text{ac}} \\
\text{ac} \\
\downarrow \text{g} \downarrow (1) \downarrow \text{h} \\
G \leftarrow D \rightarrow H
\end{array}
\]

Construction. The construction \( \text{Left} \) is inductively defined as follows:

- \( \text{Left}(p, \text{true}_p) = \text{true}_{p'} \).
- \( \text{Left}(p, \exists(a, \text{ac})) = \exists(a', \text{Left}(p', \text{ac})) \)
- if \( R' \Rightarrow p^{-1},a,a' \ L' \) is a direct transformation by \( p^{-1} = (R \leftarrow K \rightarrow L) \) and \( p' = (L' \leftarrow K' \rightarrow R') \) is the rule derived of \( L' \Rightarrow p,a,a' \ R' \) and \( \text{false} \), otherwise.
- \( \text{Left}(p, \neg \text{ac}) = \neg \text{Left}(p, \text{ac}) \) and \( \text{Left}(p, \land i \in J \text{ac}_i) = \land i \in J \text{Left}(p, \text{ac}_i) \).

Example 17 (From right to left application conditions). Consider the rule \( p = \text{createPN}(\text{name}=\text{String}) \) and right application condition \( r_{c1} = r_{c11} \land r_{c12} \) constructed in Example 16. The construction \( \text{Left} \) yields the left application condition \( \text{Left}(p, r_{c1}) = l_{c11} \land l_{c12} \) with

\[
l_{c11} = \text{Left}(p, r_{c11}) = \text{false} \\
l_{c12} = \text{Left}(p, r_{c12}) = \forall(\emptyset \rightarrow \exists(1:PN \xrightarrow{\text{name}=\text{String}} \exists(1:PN \xrightarrow{\text{place}=\text{Pl}} \exists(1:PN \xrightarrow{\text{token}=\text{Tk}})))
\]

The result can be obtained as follows:

1. Draw the rule horizontally as top row of the figure.
2. Draw the right application condition vertically below the right-hand side. This yields several rows.
3. Complete the rows by trying to apply the (derived) inverse rule according to morphisms of the right application condition. If this is possible, we get a corresponding morphism on the left-hand side, if not, for an existential condition, we give back \( \text{false} \) (and, for an universal condition, \( \text{true} \)).
4. Read the left application condition (over the left-hand side) from the completed figure.

The application of the \( \text{Left} \)-construction to the subcondition \( r_{c11} \) yields \( \text{false} \) because the dangling condition is not fulfilled for the inverse rule applied to the morphism from the first to the second graph of the constraint (denoted by crosses in Figure 10(a)). The application of the \( \text{Left} \)-construction to the subcondition \( r_{c12} \) of \( r_{c1} \) yields the left application condition \( l_{c12} \) (see Figure 10(b)). The application of the \( \text{Left} \)-construction to the \( r_{c1} \) yields the left application condition \( l_{c11} \land l_{c12} = \text{false} \).

![Figure 10: Construction of \( \text{Left}(p, r_{c11}) \) and \( \text{Left}(p, r_{c12}) \).](image-url)
7.4. Constraints on Rule Parameters

We illustrate the effect that constraints may have on rule parameters such as names of created elements at the following example.

**Example 18 (From constraints on attribute values to conditions on rule parameters).** Consider the rule \(\text{createPN}(n:\text{String})\) and the constraint

\[ c_2 = \neg \exists \begin{cases} 1:PN \\ name=' ' \end{cases} \]

stating that a Petri net must not have an empty name. Translating the constraint by \texttt{Shift} and \texttt{Left} shows a special case: The PN-node is removed from the constraint, but the parameter \(n\) (used to refer to the Petri net name) remains. Hence, the corresponding right application condition states that \(n\) must not be the empty string.

In more detail, performing Shift over the morphism \(a\) from the empty graph to the right-hand side of the rule leads to \(rc_2 = \text{Shift}(a, c_2) = rc_{21} \land rc_{22}\) with

\[ rc_{21} = \neg \exists \begin{cases} p:PN \\ name=n \\ \rightarrow p:PN \\ name=n \end{cases} \]

\[ rc_{22} = \neg \exists \begin{cases} p:PN \\ name=n \\ \rightarrow 1:p:PN \\ name=n=' ' \end{cases} \]

Transforming \(rc_{21}\) to a left application condition results in the original constraint \(c_2\), whereas transforming \(rc_{22}\) leads to \(n \neq \' \'\). The construction is illustrated in Figure 10.

As a result of this section, the translation of OCL invariants to graph constraints together with the \texttt{Shift} and \texttt{Left} constructions can be used to equip a set of rules with suitable (left) application conditions. In Section 2.4, we have introduced generation rules for Petri nets. Their application conditions are shown in Section 2.5. They ensure that after each transformation step all OCL invariants are satisfied. The generation of an example Petri net from the empty start graph using these extended rules is presented in Section 2.6.

8. Related Work

In the literature, there are several approaches to translate OCL to formal frameworks. Most of them are logic-oriented; they translate class models with OCL invariants into logical facts and formulas. The motivations for translating OCL to formal frameworks are manifold and include defining a clear semantics, generating model instances, and performing formal verification of UML/OCL models. In the following, we sketch logic-oriented approaches using the Key prover, the Alloy project, and Constraint Logic Programming, respectively, and compare them roughly with graph-based approaches.

In [37], 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 based on the KeY system.

Formal methods such as Alloy [4] 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 possible instances. In [33], UML models are automatically transformed to corresponding Alloy representations. Alloy models can then be analyzed automatically, with the help of the Alloy Analyzer. Defining a finite scope, i.e., an upper bound for the number of elements, the Alloy Analyzer can check if there exists an instance within that scope which fulfills all constraints. If the answer is “no”, the user cannot infer that there does not exist an instance fulfilling all constraints. A recent work translating OCL to relational logic is presented in [3] covering more features than UML2Alloy. Comparing our instance-generation approach to Alloy-based ones, we can state that we do not establish scopes for instance generation but provide a rule set which guarantees the generation of valid instances.
an extreme case, however, our translation may lead to application condition false for certain rules which may lead to an inapplicable rule set while Alloy would not find instances in given scopes. These are two different ways to deal with the undecidability of the problem. It is up to future work to develop ideas how instance-generating rules can be adapted to rules that do not become inapplicable by our constraint translation. Although this will not always be possible, it would be interesting to push the limit.

In [2], Cabot et al. present UML2CSP, a tool that is able to automatically check correctness properties of UML class model with OCL constraints based on Constraint Logic Programming. UML2CSP translates a UML diagram and a set of OCL invariants into a Constraint Satisfaction Problem (CSP) that is solved by the constraint solver ECLiPSe. As for the Alloy-based instances, the system searches within user-defined upper bounds for an instance satisfying all constraints. In contrast to UML2Alloy and [3], Constraint Programming does not restrict the expressive power of OCL.

The USE tool [18, 39] can be used for generating snapshots that conform to the model or for checking the conformity of a specific instance. Gogolla et al. follow a more imperative approach than the CSP- and SAT-based tools: The user provides a sequence of commands to sequentially generate snapshots, allowing backtracking. Inside the command sequence, OCL expressions are used to guide the snapshot generation. Additionally a set of OCL invariants can be loaded to further restrict the search results or to check the implication problem for OCL invariants on the generated snapshots. In contrast to our approach and the ones presented above, the USE tool uses OCL directly instead of translating invariants to other representations. Opposed to solver-based approaches, the user has a more fine-grained control over the instance generation.

In contrast to logic-oriented approaches, graph-based approaches translate OCL constraints to graph patterns or graph constraints. Pennemann has shown in [30] that a theorem prover for graph conditions works more efficiently than theorem provers for logical formulas being applied to graph conditions. The key idea is here that graph axioms are always satisfied by default when using a theorem prover for graph conditions. Lambers and Orejas [40] have shown that this theorem prover is not only correct but also complete, i.e., whenever an implication holds, it can be shown by the theorem prover.

Bergmann [9] has translated OCL constraints to graph patterns. He considers a pretty similar subset of OCL as we do (extended to some kind of OCL expressions not being first-order), and in fact, the way of translation shows a lot of similarities. The focus of that work, however, is not a formal translation showing correctness and completeness but an efficient implementation of constraint checking which is tested at example constraints. Since the implementation uses incremental pattern matching, it outperforms other implementations such as a naive Java implementation, the OCL interpreter of Eclipse, etc. for large model graphs. It has not yet been compared with the theorem prover by Pennemann.

In [41], an interesting extension of our OCL translation is considered: Special “trace” edges are used to represent ordered sets. Another extension of our OCL translation is presented in [42], where HR* conditions, an extension of nested conditions, are used to translate a subset of OCL expressions using iterate as operation.

9. Conclusion

This article presents a translation of Essential OCL to nested graph constraints. While navigation expressions in OCL are translated to graph patterns, the main idea for translating constraints with set operations is to use the characteristic function of sets which assigns each set operation its corresponding Boolean operation. For this translation, we introduce a compact notion of nested graph conditions. They permit a comparable complexity of OCL constraints and translated graph constraints. Hence, they present a new graphical representation of OCL invariants being slightly more abstract since several navigation paths can be combined in graphs and set operations are reduced to Boolean operations. Compact conditions are extensively used in our OCL translation. As the central contribution of this article we show that the translation of Essential OCL invariants to nested graph constraints is correct, i.e., a model satisfies an Essential OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint. Moreover, we show that our translation is complete for the chosen set of OCL invariants.
Translating Essential OCL invariants to nested graph constraints and further to application conditions of rules provides the missing link between meta-modeling and transformation systems which may be advantageously used by MDE activities. Applications of our results include but are not limited to the following: The generation of meta-model instances for, e.g., testing of model transformations and other kinds of algorithms on models. In addition, the generation of edit operations as well as model repair actions from meta-models may be interesting: For example, model change recognition as well as model patching may be lifted to recognizing and packaging edit operations to patches. To adapt such a general approach to domain-specific needs, complete sets of edit operations have to be specified being able to build up and destroy all models of a DSML. The automatic generation of edit operations from a given meta-model would be of great help. A generation approach for meta-models without OCL constraints is presented in [44]; the incorporation of OCL constraints is left for future work. Another kind of application of the presented translation is the following: Algebraic graph transformation (AGT) may be used as a formal foundation for model transformation languages: In [45], Richa et al. translate core concepts of ATL (including OCL constraints) to AGT to produce weakest preconditions of model transformations. To ensure that ATL developers will understand them, the backward translation of graph conditions to OCL is also of interest.

For an application in practice, we plan to implement the presented translation of OCL to application conditions in the context of the Eclipse Modeling Framework and Henshin [26], a model transformation environment based on graph transformation concepts. In this context, we will also reconsider the OCL translation by Bergmann [9] to EMF-IncQuery. Our ultimate goal is a well-defined approach to the translation of meta-models to language-specific model transformation systems for generating, completing or repairing models such that they are valid w.r.t. their meta-models afterwards.

Acknowledgement. We are grateful to Jens Kosiol and Nebras Nassar who intensively tested the transformations and to Christian Sandmann, Tillmann Teusch, and the anonymous reviewers for their helpful comments on previous versions of this paper.

References

### Syntax and semantics of operation expressions in Essential OCL

#### Table 1: OCL operations on pre-defined and custom types.

<table>
<thead>
<tr>
<th>Syntax and Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e, e_1, e_2 \in Expr$</td>
<td>$I<a href="%5Ctau">e</a>$ with $\tau = (S, \beta) \in Env$</td>
</tr>
<tr>
<td>$e_1 : T = e_2 : T \rightarrow Boolean$</td>
<td>$I<a href="%5Ctau">e_1</a> = I<a href="%5Ctau">e_2</a>$</td>
</tr>
<tr>
<td>$e_1 : T \leftrightarrow e_2 : T \rightarrow Boolean$</td>
<td>$I<a href="%5Ctau">e_1</a> \neq I<a href="%5Ctau">e_2</a>$</td>
</tr>
<tr>
<td>$e_1 + e_2 \rightarrow Int$</td>
<td>$I<a href="%5Ctau">e_1</a> + I<a href="%5Ctau">e_2</a>$</td>
</tr>
<tr>
<td>$e_1 \leq e_2 \rightarrow Boolean$</td>
<td>$I<a href="%5Ctau">e_1</a> \leq I<a href="%5Ctau">e_2</a>$</td>
</tr>
<tr>
<td>$e_1 \text{ and } e_2 \rightarrow Boolean$</td>
<td>$I<a href="%5Ctau">e_1</a> \land I<a href="%5Ctau">e_2</a>$</td>
</tr>
<tr>
<td><code>'a string' \rightarrow String</code></td>
<td><code>'a string'</code></td>
</tr>
<tr>
<td>$e : C.\text{allInstances() \rightarrow Set(C)}$</td>
<td>$\sigma_{Class}(C)$</td>
</tr>
<tr>
<td>$e : C.\text{attr} \rightarrow T$ with $\text{attr} \in \sigma_{Att}(C)$</td>
<td>$\sigma_{Att}(\text{attr})(I<a href="%5Ctau">e_1</a>)$</td>
</tr>
<tr>
<td>$e : C.\text{nav} \rightarrow C'$ with $(e, \text{nav}) \in \sigma_{Assoc}$</td>
<td>$\text{nav}$ with $(I<a href="%5Ctau">e</a>, \text{nav}) \in \sigma_{Assoc}$</td>
</tr>
<tr>
<td>$e : C.\text{nav} \rightarrow \text{Set}(C')$ with $(e, \text{nav}) \in \sigma_{Assoc}$</td>
<td>${\text{nav} \mid (I<a href="%5Ctau">e</a>, \text{nav}) \in \sigma_{Assoc}}$</td>
</tr>
<tr>
<td>$e : T.\text{oclIsKindOf}(T')$</td>
<td>$I<a href="%5Ctau">e</a> \in I(T')$</td>
</tr>
<tr>
<td>$e : T.\text{oclIsTypeOf}(T')$</td>
<td>$I<a href="%5Ctau">e</a> \in I(T') - \bigcup_{T'' \leq M^{\preceq} T'} I(T'')$</td>
</tr>
<tr>
<td>$e : T.\text{oclAsType}(T')$</td>
<td>$I<a href="%5Ctau">e</a>$ if $I<a href="%5Ctau">e</a> \in I(T')$ and $\emptyset$ otherwise.</td>
</tr>
</tbody>
</table>
Table 2: OCL operations on sets.

<table>
<thead>
<tr>
<th>Syntax and Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e, e' \in \text{Expr}, S = \text{Set}(T)$</td>
<td>$I<a href="%5Ctau">e</a>$ with $\tau = (S, \beta) \in \text{Env}$</td>
</tr>
<tr>
<td>$\text{Set}(e_1, \ldots, e_n) \rightarrow S$ with $e_1, \ldots, e_n$ of type $T$</td>
<td>${I<a href="%5Ctau">e_1</a>, \ldots, I<a href="%5Ctau">e_n</a>}$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{union}(e':S) \rightarrow S$</td>
<td>$I<a href="%5Ctau">e</a> \cup I<a href="%5Ctau">e'</a>$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{intersection}(e':S) \rightarrow S$</td>
<td>$I<a href="%5Ctau">e</a> \cap I<a href="%5Ctau">e'</a>$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{diff}(e':S) \rightarrow S$</td>
<td>$I<a href="%5Ctau">e</a> - I<a href="%5Ctau">e'</a>$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{symmetricDifference}(e':S) \rightarrow S$</td>
<td>$(I<a href="%5Ctau">e</a> \cup I<a href="%5Ctau">e'</a>) - (I<a href="%5Ctau">e</a> \cap I<a href="%5Ctau">e'</a>)$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{select}(v:T</td>
<td>e':\text{Boolean}) \rightarrow S$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{collect}(v:T</td>
<td>e':T') \rightarrow \text{Set}(T')$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{size()} \rightarrow \text{Int}$</td>
<td>$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{notEmpty()} \rightarrow \text{Boolean}$</td>
<td>$I<a href="%5Ctau">e</a> \neq \emptyset$</td>
</tr>
<tr>
<td>$e:S \rightarrow \exists(v:T</td>
<td>e':\text{Boolean}) \rightarrow \text{Boolean}$</td>
</tr>
<tr>
<td>$\text{if } I<a href="%5Ctau">e</a> = {x_1, \ldots, x_n} \text{, else false}$</td>
<td></td>
</tr>
<tr>
<td>$e:S \rightarrow \forall(v:T</td>
<td>e':\text{Boolean}) \rightarrow \text{Boolean}$</td>
</tr>
<tr>
<td>$\text{if } I<a href="%5Ctau">e</a> = {x_1, \ldots, x_n}, \text{ else true}$</td>
<td></td>
</tr>
<tr>
<td>$e:S \rightarrow \text{includesAll}(e':S) \rightarrow \text{Boolean}$</td>
<td>$I<a href="%5Ctau">e'</a> \subseteq I<a href="%5Ctau">e</a>$</td>
</tr>
<tr>
<td>$e:S \rightarrow \text{excludesAll}(e':S) \rightarrow \text{Boolean}$</td>
<td>$I<a href="%5Ctau">e</a> \cap I<a href="%5Ctau">e'</a> = \emptyset$</td>
</tr>
</tbody>
</table>