A Nesting-Free Normal Form for Nested Conditions in Finite Lattices of Subgraphs

Jens Kosiol[1], Steffen Zschaler[2]

[1]Philipps-Universität Marburg, Marburg, Germany
[2]King's College London, London, UK

Abstract

We present a nesting-free normal form for the formalism of *nested conditions and constraints* in the context of finite lattices of subgraphs.

Keywords

First-order graph logic, nested graph conditions, normal forms, finite lattices of subgraphs

1. Introduction

In this paper, we present a nesting-free normal form for the formalism of *nested conditions and constraints* [1, 2] in the context of finite lattices of subgraphs. Nested conditions express properties of morphisms, while constraints express properties of objects. This formalism has established itself as the central logical formalism to be used in the context of *graph transformation* because it allows to elegantly compute the interactions of the applications of transformation rules and the validity of constraints, e.g., [3, 4, 5, 6, 7, 8, 9, 10]. On (typed) graphs, nested constraints have been shown to be expressively equivalent to first-order logic [1, 3]. *Moreover, every additional level of nesting increases the expressivity of the formalism [1].*

While predominantly used for typed graphs, the logic of nested conditions and constraints can be used in any category. In ongoing work [11], we employ this formalism in the context of finite lattices of subgraphs, i.e., categories whose objects are the (equivalence classes of) subgraphs of a given finite *container graph* T and whose morphisms are inclusions; we denote such a category as $\mathbf{Sub}(T)$. Transformations of these objects can be realised via so-called *subgraph transformation systems* [12, 13], where transformation rules themselves consist of subgraphs of the given container graph T and are applied to subgraphs of T in such a way that the result is again a subgraph of T.

In this paper, we investigate the formalism of nested conditions in the context of finite lattices of subgraphs and make two contributions. First, we work out a normalisation construction, called *flattening*, that transforms every nested condition/constraint into a *nesting-free* Boolean combination of literals (i.e., graph patterns). In that respect, our central result is the correctness of that construction (Theorem 3), implying that every condition/constraint can be equivalently transformed into a conjunctive normal form on the basis of literals (Corollary 4). We present our flattening construction in Section 4.

Furthermore, every graph G in a lattice of subgraphs of a container graph T can also be considered as a graph in the category $\mathbf{Graph_{TG}}$, the category of all graphs typed over TG, the type graph over which the container graph T is typed. In particular, a nested constraint formulated in the category $\mathbf{Graph_{TG}}$ can be evaluated for subgraphs of T. We develop a translation that, given a nested condition/constraint for $\mathbf{Graph_{TG}}$ and a *finite* container graph T (typed over TG) transforms that condition/constraint into one formulated in the category $\mathbf{Sub}(T)$. Our main contribution in that regard is showing our translation to preserve the semantics of the given constraints (Theorem 8). This ensures that, while nesting of constraints collapses in the context of lattices of subgraphs, no expressivity of the formalism is lost given a finite container graph T. Moreover, this translation makes specifying and working

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29–July 3, 2026
kosiolje@mathematik.uni-marburg.de (J. Kosiol); szschaler@acm.org (S. Zschaler)
0000-0003-4733-2777 (J. Kosiol); 0000-0001-9062-6637 (S. Zschaler)
© 2026 Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26

CEUR-WS.org/Vol-4338/GCM-paper1.pdf

with nested conditions/constraints in the context of finite lattices of subgraphs much more practical, since nested conditions/constraints in $\mathbf{Graph}_{\mathbf{TG}}$ are much more compact than their counterparts in $\mathbf{Sub}(T)$. We present our translation in Section 5.

The results in this paper are not surprising. It is intuitive that in a context where all finitely many (graph) elements that can ever appear are known beforehand, properties of these graphs can be expressed as a Boolean combination of non-nested graph patterns that list the different ways in which the property in question can (should not) hold. Still, making this precise is somewhat subtle. Moreover, in [11], we develop an interesting application of the results presented here. Considerably generalising work of Horcas et al. [14], we there present an approach to *non-blocking consistency-preserving subgraph transformation*, where, from a given subgraph transformation rule, we derive a family of rules that (i) contain the given rule as subrule, (ii) cannot introduce violations of the given constraints upon application and (iii) are not blocked by application conditions. Instead, the original rule is amended by additional actions that immediately remedy introduced constraint violations. We see considerable potential of this rule construction in particular in the context of *model-driven optimisation* [15]. Importantly, the rule construction we develop in [11] is based on the normal form for conditions in $\mathbf{Sub}(T)$ we develop here.

Related Work Nested conditions and constraints have been introduced by Rensink [1] and by Habel and Pennemann [2] to increase the expressiveness of non-nested constraints and application conditions that had been considered before [16]. Nested conditions are expressively equivalent to first-order logic on graphs [1, 3, Theorems 1 and 3 or Theorems 10 and 11, respectively]. Searching for normal forms of formulas is a standard technique in logic (see, e.g., [17] for classic propositional and first-order logic). On graphs, every level of nesting increases expressiveness of nested conditions [1, Theorem 4]; consequently, normal forms for this formalism cannot generally avoid nesting but recursively require the individual levels of nesting to be in a certain normal form like CNF (see, e.g., [18, ch. 6.3]). Furthermore, there exist normal forms for nested constraints that require quantifiers to alternate with nesting level. These, however, either need to allow for isomorphisms in the conditions [19] or are only guaranteed to exist for *linear* conditions (i.e., conditions without conjunctions and disjunctions) [20]. In this paper, we develop a non-nested normal form for nested conditions in the very restricted setting of the lattice of subgraphs of a given graph.

Structure Before presenting our contribution in Sections 4 and 5, we begin by introducing the *class–responsibility assignment (CRA) problem* as a running example we use to illustrate our constructions (Section 2) and by presenting preliminaries (Section 3). We conclude in Section 6 and provide all proofs in an extended version of this paper [21]. *Sections 2 and 3 are taken over from [11] with only minor adaptations.*

2. Running Example: The CRA Case

In the CRA problem [22], a set of methods and attributes must be allocated to a (to be determined) number of classes to optimise cohesion and coupling metrics of the resulting software design. It is not immediately obvious that the search space for this problem is finite, as it might involve the generation of an unspecified number of new classes (which are nodes in a graph representing allocations of methods/attributes to classes). However, once one realises that any meaningful solution contains at most as many classes as there are features (methods and attributes), it becomes clear that the problem is about determining the correct set of edges between features and classes. In particular, we will be able to understand the solutions of a given problem instance as a lattice of subgraphs $\mathbf{Sub}(T)$.

Figure 1 shows the metamodel (type graph) for the CRA problem—a class diagram whose instances describe specific sets of classes, attributes, and methods for which to determine an allocation. Figure 2 then shows a model that instantiates the meta-model in Figure 1 and describes a concrete problem instance P with 3 methods and 3 attributes that need to be allocated to classes. Figure 3 shows a

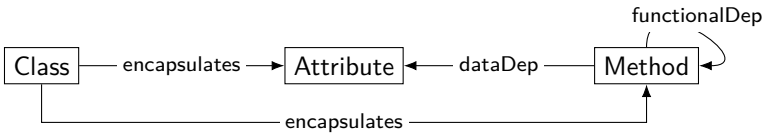


Figure 1: A meta-model (type graph) TG for defining instances of the CRA problem (based on [22])

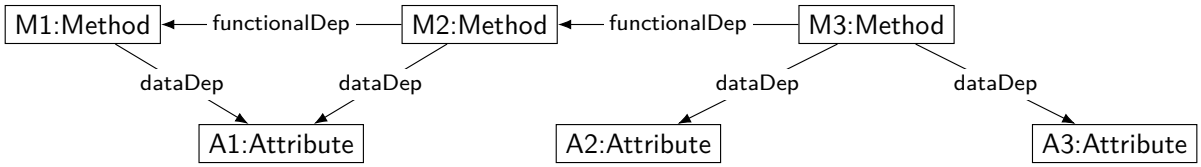


Figure 2: Example instance P of the CRA problem

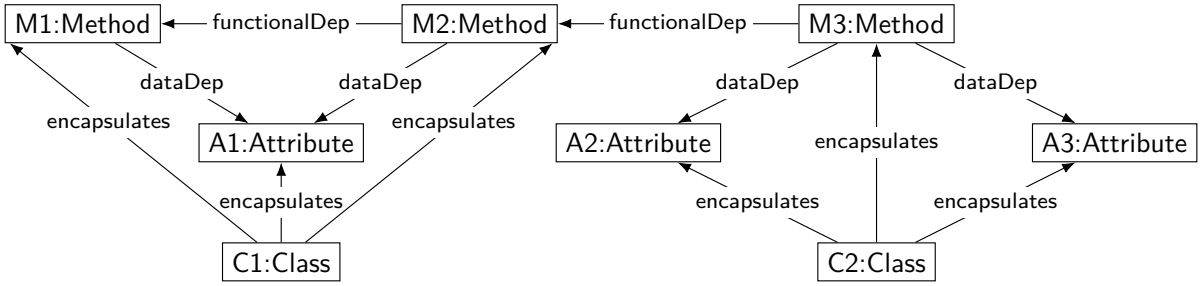


Figure 3: Example of a solution S to the CRA problem instance depicted in Fig. 2

corresponding instance S of the metamodel that represents one possible solution to the problem instance defined by P. Here, we have introduced two classes and made specific assignments of methods and attributes to these classes. The distinction between *problem instances* and *solutions* is not of importance for this paper; we have discussed this distinction in more detail in [11], where it becomes relevant, and we will shortly comment on it in the following section.

3. Preliminaries

In this section, we introduce our preliminaries. As our general background, we recall *typed graphs* [16] and *nested conditions and constraints* [1, 3]. Moreover, we present *finite lattices of subgraphs*, for which we investigate nested conditions and constraints more closely in this paper.

3.1. Typed Graphs

Graphs are sets of nodes with edges between them. Typing offers a way to equip the individual elements of a graph with meaning; a *type graph* provides the available types for elements, and morphisms assign the elements of *typed graphs* to those types.

Definition 1 (Graph. Typed graph)**.** A *graph* $G = (V_G, E_G, src_G, tar_G)$ consists of a set of nodes V_G, a set of edges E_G, and source and target functions $src_G, tar_G \colon E_G \to V_G$. A *graph morphism* $f \colon G \to H$ between two graphs is a pair of functions $f_V \colon V_G \to V_H$ and $f_E \colon E_G \to E_H$ that is compatible with the source and target functions, such that $src_H \circ f_E = f_V \circ src_G$ and $tar_H \circ f_E = f_V \circ tar_G$.

A *type graph* $TG = (V_{TG}, E_{TG}, src_{TG}, tar_{TG})$ is a fixed graph. A *graph typed over a fixed type graph* TG consists of a graph G and a graph morphism $type_G \colon G \to TG$. A *typed graph morphism* $f \colon G \to H$ between graphs G and H typed over TG is a graph morphism that is compatible with typing, i.e., such that $type_H \circ f = type_G$.

A typed graph morphism is *injective/surjectiv/bijective/an inclusion* if both of its components are. We denote injective morphisms via hooked arrows $f\colon G \hookrightarrow H$ and regularly denote inclusions as $\iota_H^G\colon G \subseteq H$, where the indices denote the involved graphs.

A typed graph G is *finite* if both its sets of nodes V_G and of edges E_G are.

Remark 1. To ease language and notation, we adopt the following conventions and assumptions.

- In this paper, we work with typed graphs and always assume a fixed type graph TG over which all graphs are typed. Except for the type graph, we will just speak of *graphs*, and we do not introduce a special notation for typed graphs. Thus, speaking of a graph G, we tacitly assume a type graph TG and a type morphism $type_G\colon G \to TG$ to exist. Similarly, we also regularly just speak of morphisms (and not of typed ones).
- Via $\emptyset$ we denote the *initial* or *empty graph*, also when it is considered to be typed over the given type graph TG (via the empty function).
- By $\mathbf{Graph_{TG}}$ we denote the category of all graphs typed over the given type graph TG.

Example 1. The graph TG from Fig. 1 can serve as a type graph, modelling the domain of the CRA problem. The nodes provide the types Class, Attribute and Method. Methods can have dependencies on Methods (functionalDep) and on Attributes (dataDep). Classes can encapsulate Methods and Attributes.

The further graphs from Sect. 2, namely graph P (Fig. 2) and graph S (Fig. 3), are typed over that type graph TG (and the typing morphisms are indicated by the annotations of nodes and edges).

3.2. Lattices of Subgraphs

Subobject transformation systems have been developed as a tool to study different notions of causality between rule applications in (double-pushout) rewriting systems [12, 13]. Intuitively, one studies transformations of subobjects of a given object from an ($\mathcal{M}$-)adhesive category. In this paper, we do not care about this notion of transformation but more closely investigate nested conditions for the categories for which these transformations are defined. In that, we restrict ourselves to the case where underlying objects are graphs. For simplicity of presentation, we work with subgraphs instead of equivalence classes of injective morphisms.

Definition 2 (Subgraph. (Finite) lattices of subgraphs)**.** Given a typed graph G, a *subgraph* of it is a typed graph H, such that $V_H \subseteq V_G$, $E_H \subseteq E_G$ and src_H, tar_H, and $type_H$ are restrictions of the according functions of G (in particular, with every edge $e \in E_H$ its source and target nodes in V_G belong to V_H). Via $H \subseteq G$ we denote the subgraph relationship between graphs.

Given a (finite) typed graph T, the *(finite) lattice of subgraphs of* T, denoted as $\mathbf{Sub}(T)$, has graphs $G \subseteq T$ as objects and inclusions $\iota_H^G\colon G \subseteq H$ between such subgraphs as morphisms. We call T the *container graph* of that category.

Remark 2. Finiteness of the container graph T obviously implies finiteness of $\mathbf{Sub}(T)$. What we call the *container graph* would be called *type graph* in the terminology of [12]. We deviate from that terminology to distinguish the container graph T from the type graph TG over which T (and all graphs from $\mathbf{Sub}(T)$) are typed.

Example 2. Given the type graph TG from Fig. 1 (see Sect. 2 and Example 1), also the graph T depicted in Fig. 4 is typed over it. Considering the CRA problem instance defined by the graph P shown in Fig. 2, $\mathbf{Sub}(T)$ defines a lattice that contains all interesting solutions to that problem instance (disregarding solutions with more than six Classes of which several will be necessarily empty). The graph S depicted in Fig. 3 is one of the elements of $\mathbf{Sub}(T)$. Note that $\mathbf{Sub}(T)$ is actually too large to be considered the search space for finding solutions for the problem instance defined by the graph P: we are only interested in graphs also containing that instance P; otherwise, the problem definition would be changed (containing a different set of Methods and Attributes and/or other dependencies). In [11], we address that problem by only considering graphs $G \in \mathbf{Sub}(T)$ for which $P \subseteq G$, where

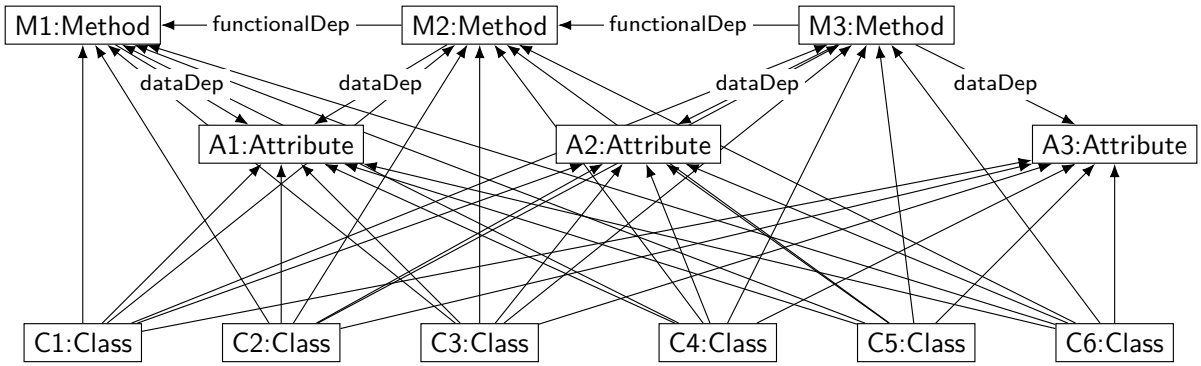


Figure 4: A container graph T defining a lattice of subgraphs $\mathbf{Sub}(T)$ that includes all interesting solutions to the CRA problem instance P defined in Fig. 2. In this graph, we omit the encapsulates annotation to not further clutter the figure.

$P \subseteq T$ is considered to define a problem instance; a category that we denote as $\mathbf{Sub}(T \supseteq P)$. In this paper, however, we can safely ignore this subtlety.

Categories $\mathbf{Sub}(T)$ consisting of the subgraphs of T actually constitute lattices [12, 13]:

Remark 3. In any category of subgraphs $\mathbf{Sub}(T)$, if there exists a morphism between subgraphs A and B, this morphism is unique (by left-cancelability of monomorphisms). Moreover, given an adhesive category $\mathcal{C}$ (such as the category of graphs typed over a given type graph TG), for every object T of $\mathcal{C}$, $\mathbf{Sub}(T)$ constitutes a distributive lattice [23, Theorem 5.1 and Corollary 5.2]. Given two subobjects $A, B \subseteq T$, their *meet* $A \cap B$ in $\mathbf{Sub}(T)$ arises from pulling back $A \to T \leftarrow B$ in $\mathcal{C}$ and their *join* $A \cup B$ in $\mathbf{Sub}(T)$ from pushing out $A \leftarrow A \cap B \to B$ in $\mathcal{C}$. In our setting, where we work with subgraphs instead of equivalence classes of injections, this means that the meet $A \cap B$ is literally the intersection of A and B while the join $A \cup B$ is their union.

3.3. Nested Conditions and Constraints

Next, we define *nested conditions* and *constraints* [3, 1, 4]. Nested conditions provide a logic to express properties of morphisms in a category; constraints express properties of objects. For (typed) graphs, nested conditions are expressively equivalent to standard first-order logic on graphs [1, 3], and every level of nesting increases the expressivity [1].

In this work, given a type graph TG and a (finite) graph T typed over TG, we consider two different categories: The category of graphs typed over TG, $\mathbf{Graph_{TG}}$, and the category $\mathbf{Sub}(T)$ of subgraphs of T (with inclusions as morphisms). Since every graph from $\mathbf{Sub}(T)$ is also a graph typed over TG (by composition of the inclusion with the typing of T), we obtain two variants of defining formulas for it. We define conditions and constraints category-theoretically and afterwards illustrate the definitions for $\mathbf{Graph_{TG}}$ and $\mathbf{Sub}(T)$.

Definition 3 ((Nested) conditions and constraints)**.** In a category $\mathcal{C}$, given an object C_0, a *(nested) condition* over C_0 is defined recursively as follows:

- true is a condition over C_0.
- If $a_1 \colon C_0 \hookrightarrow C_1$ is a monomorphism and d is a condition over C_1, $\exists\,(a_1 \colon C_0 \hookrightarrow C_1, d)$ is a condition over C_0.
- Moreover, Boolean combinations of conditions over C_0 are conditions over C_0.

In a category with a strict initial object $\emptyset$, a *(nested) constraint* is a condition over $\emptyset$.

We recursively define the *nesting level* of a condition c as

$$\mathrm{nl}(\mathsf{true}) = 0; \qquad \mathrm{nl}(\exists(a_1 \colon C_0 \hookrightarrow C_1, d)) = \mathrm{nl}(d) + 1;$$

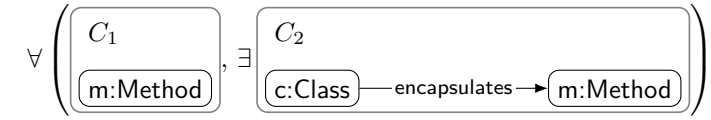


Figure 5: Constraint c_{lb} in $\mathbf{Graph_{TG}}$ requiring every Method to be assigned to at least one Class

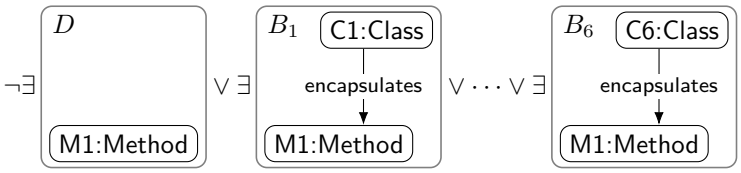


Figure 6: One conjunct of constraint $c_{\text{lb}}^{\mathbf{Sub}(T)}$ in $\mathbf{Sub}(T)$, requiring Method M1 to be assigned to at least one of Classes C1, …, C6 from T (in case it exists)

$$\text{nl}(\neg d) = \text{nl}(d); \qquad \text{nl}(d_1 \odot d_2) = \max\{\text{nl}(d_1), \text{nl}(d_2)\}, \text{ where } \odot \in \{\vee, \wedge\}.$$

Satisfaction of a nested condition c over C_0 for a morphism $g\colon C_0 \to G$, denoted as $g \models c$, is defined as follows:

- Every morphism satisfies true.
- The morphism g satisfies a condition of the form $c = \exists\,(a_1\colon C_0 \hookrightarrow C_1, d)$ if there exists a monomorphism $q\colon C_1 \hookrightarrow G$ such that $g = q \circ a_1$ and $q \models d$.
- For Boolean operators, satisfaction is defined as usual.

An object G satisfies a constraint c, denoted as $G \models c$, if the unique monomorphism $\iota_G\colon \emptyset \hookrightarrow G$ does so.

Remark 4. In the notation of conditions, we drop the domains of the involved morphisms and occurrences of true whenever they can be unambiguously inferred. Moreover, we introduce $\forall\,(C_1, d)$ as an abbreviation for the condition $\neg\exists\,(C_1, \neg d)$. For example, we write $\forall\,(C_1, \exists\, C_2)$ instead of $\neg\exists\,(a_1\colon \emptyset \hookrightarrow C_1, \neg\exists\,(a_2\colon C_1 \hookrightarrow C_2, \text{true}))$. At other places, especially where it is convenient for compact presentation of computations, we will not display the graphs of a condition but merely the identifiers of the involved morphisms, writing, e.g., $\forall\,(a_1, \exists\, a_2)$.

To be able to notationally distinguish between satisfaction of constraints and conditions in the category $\mathbf{Graph_{TG}}$ from the one in $\mathbf{Sub}(T)$ (where T is typed over TG), we use the symbol $\models_{\mathcal{I}}$ to denote satisfaction in $\mathbf{Sub}(T)$ and $\equiv_{\mathcal{I}}$ to denote equivalence in $\mathbf{Sub}(T)$. Moreover, we denote the morphisms in conditions over graphs from $\mathbf{Sub}(T)$ as $b_i\colon B_{i-1} \subseteq B_i$ to visually differentiate these from (i) morphisms $a_i\colon C_{i-1} \hookrightarrow C_i$ in conditions over a graph from $\mathbf{Graph_{TG}}$ and (ii) from inclusions $\iota_G^{C_i}\colon C_i \subseteq G \in \mathbf{Sub}(T)$ we use to check the satisfaction of such conditions.

Example 3. The definition of the CRA problem comes with additional constraints (compare Sect. 2): As a minimum, one requires that every Attribute and every Method is encapsulated by exactly one Class.

Figure 5 depicts in $\mathbf{Graph_{TG}}$ the constraint expressing the *lower bound* on encapsulation of Methods, namely stating that every Method shall be encapsulated by (at least) one Class (the names of the elements in the graphs indicate the morphism from C_1 to C_2); in the following, we refer by c_{lb} to that constraint. In $\mathbf{Sub}(T)$, in contrast, we can only express properties of concrete elements (as the constraints in $\mathbf{Sub}(T)$ are defined in terms of subgraphs of T); thus, the constraint in Fig. 6 expresses that the concrete Method M1 is encapsulated by *a* Class—either C1, C2, … or C6. The same constraint has to be added for Methods M2 and M3. By $c_{\text{lb}}^{\mathbf{Sub}(T)}$, we refer to the conjunction of these constraints.

Continuing, the constraint from Fig. 7 expresses in $\mathbf{Graph_{TG}}$ that no Method is encapsulated by more than one Class, that is, it states an *upper bound* on these encapsulations; in the following, we refer

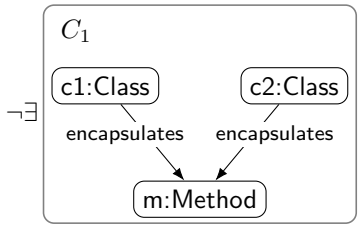


Figure 7: Constraint c_{ub} in $\mathbf{Graph_{TG}}$ forbidding any Method to be assigned to more than one Class

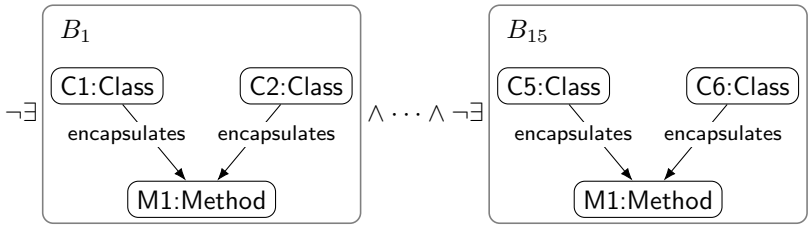


Figure 8: Conjunct of constraint $c_{\text{lb}}^{\mathbf{Sub}(T)}$ in $\mathbf{Sub}(T)$, forbidding Method M1 to be assigned to more than one of the Classes C1, …, C6 from T

by c_{ub} to that constraint. Again, in $\mathbf{Sub}(T)$, we have to express this for every Method individually and exclude any possible pair of Classes; the resulting set of constraints for Method M1 is shown in Fig. 8. By $c_{\text{ub}}^{\mathbf{Sub}(T)}$, we refer to the conjunction of these constraints.

4. A Normal Form for Conditions in Lattices of Subgraphs

In this section, we introduce a normal form for conditions in lattices of subgraphs $\mathbf{Sub}(T)$. In Sect. 3.3, we have already discussed that, given a type graph TG and a (finite)[1] graph T typed over it, we can consider conditions and constraints in two different categories: the category of graphs typed over TG, $\mathbf{Graph_{TG}}$, and the category $\mathbf{Sub}(T)$ of subgraphs of T. Working with conditions in $\mathbf{Sub}(T)$ facilitates the construction of activation diagrams and subsequent derivation of non-blocking consistency-preserving rules we develop in [11]. In particular, these constructions depend on the normal form we develop in this section. However, conditions in $\mathbf{Sub}(T)$ are inconvenient for the specification of general constraints (because of their expressing properties of concrete elements of T). To address this, in Sect. 5, we provide a way to *instantiate* a condition defined for graphs from $\mathbf{Graph_{TG}}$ into a condition for graphs from $\mathbf{Sub}(T)$ such that its semantics is preserved.

The key feature of the normal form we construct in this section is that it avoids any nesting. We show that in $\mathbf{Sub}(T)$ for every condition there is an equivalent condition without nesting. (This is in stark contrast to the case of $\mathbf{Graph_{TG}}$, where every level of nesting increases the expressivity [1].) We start by defining *literals*, the building blocks of our normal form.

Definition 4 (Literal)**.** Given a type graph TG and a graph T typed over it, a *literal* in $\mathbf{Sub}(T)$ is a condition of the form true, false, $\exists\,(b\colon B_0 \subseteq B_1, \text{true})$ or $\neg\exists\,(b\colon B_0 \subseteq B_1, \text{true})$, where $B_0, B_1 \in \mathbf{Sub}(T)$.

Next, we develop our *flattening construction* that simultaneously pushes negations inward and removes all nesting in a condition in $\mathbf{Sub}(T)$. For two positive existential quantifiers it applies the well-known equivalence $\exists\,(b_1, \exists\,(b_2, d)) \equiv \exists\,(b_2 \circ b_1, d)$ [2, Fact 4]. A positive existential condition that is followed by a negative one—that is, a condition of the form $\exists\,(b_1\colon B_0 \subseteq B_1, \neg d)$—is replaced by $\exists\,(b_1\colon B_0 \subseteq B_1) \wedge \neg\tilde{d}$, where $\tilde{d}$ is an adaptation of d so that it becomes a condition over B_0 (instead of B_1). In $\mathbf{Graph_{TG}}$, this generally is not a semantic equivalence because the first variant forbids an

[1]The results in this section actually also hold when one allows graphs to be infinite, in contrast to our results in Sect. 5.

occurrence of d at a fixed location for b_1 whereas the second variant forbids any occurrence of $\tilde{d}$. For satisfaction in $\mathbf{Sub}(T)$, however, both notions coincide because there can only be one inclusion of a graph into another one. This is why nesting collapses in $\mathbf{Sub}(T)$. In the following, we first prove the just stated equivalence and then use it in our *flattening construction*, of which it is the crucial step to transform conditions in $\mathbf{Sub}(T)$ into a non-nested normal form.

Lemma 1 (Correctness of extracting negation)**.** *Given a type graph TG and a graph T typed over it, for any condition d in $\mathbf{Sub}(T)$ over a graph $B_1 \subseteq T$, it holds that*

$$\exists\,(b_0\colon B_0 \subseteq B_1, \neg d) \equiv_{\mathcal{I}} \exists\,(b_0\colon B_0 \subseteq B_1, \mathit{true}) \wedge \neg\exists\,(b_0\colon B_0 \subseteq B_1, d).$$

The lemma just stated is the crucial ingredient to proving that our following *flattening construction* preserves semantics (in $\mathbf{Sub}(T)$). In flattening, we push negations over Boolean operators and extract them from existentially bound conditions; moreover, we compose existentially bound conditions. We end up with a condition that is nesting-free and where every negation either appears in front of an existentially bound condition or at nesting level zero (where the root of the condition can be some Boolean combination of true). We define flattening *over a morphism*, where we use that morphism to collect all morphisms we have to compose in the resulting condition (that somewhat eases notation). Flattening of a condition can then just be performed by flattening that condition over the identity morphism of the graph over which the condition is defined.

Construction 1 (Flattening)**.** Given a type graph TG, a graph T typed over it, a graph $B_0 \in \mathbf{Sub}(T)$, a morphism b_0 with co-domain B_0 and a condition c over B_0 in $\mathbf{Sub}(T)$, we recursively define to *flatten* c over b_0 via:

$$\mathrm{Flat}(b_0, \mathsf{true}) = \exists\,(b_0, \mathsf{true}) \tag{F1}$$
$$\mathrm{Flat}(b_0, \neg\mathsf{true}) = \exists\,(b_0, \mathsf{false}) \equiv \mathsf{false} \tag{F2}$$
$$\mathrm{Flat}(b_0, d_1 \odot d_2) = \mathrm{Flat}(b_0, d_1) \odot \mathrm{Flat}(b_0, d_2) \tag{F3}$$
$$\mathrm{Flat}(b_0, \neg(d_1 \odot d_2)) = \mathrm{Flat}(b_0, \neg d_1)\ \overline{\odot}\ \mathrm{Flat}(b_0, \neg d_2) \tag{F4}$$
$$\mathrm{Flat}\,(b_0, \exists\,(b_1\colon B_0 \subseteq B_1, d)) = \mathrm{Flat}(b_1 \circ b_0, d) \tag{F5}$$
$$\mathrm{Flat}(b_0, \neg\exists\,(b_1\colon B_0 \subseteq B_1, d) = \exists\,(b_0, \mathsf{true}) \wedge \neg\,\mathrm{Flat}(b_1 \circ b_0, d) \tag{F6}$$
$$\mathrm{Flat}(b_0, \neg\neg c) = \mathrm{Flat}(b_0, c) \tag{F7}$$

where $\odot \in \{\wedge, \vee\}$, $\overline{\wedge} = \vee$, and $\overline{\vee} = \wedge$.

Example 4. One regularly encounters constraints of the structure $\forall\,(b_0\colon \emptyset \hookrightarrow C_1, \exists\,(b_1\colon C_1 \hookrightarrow C_2, \mathsf{true}))$ in practice—compare, for instance, the constraint c_{lb} defining the lower bound for assignments to Classes (Fig. 5) in Example 3. When we instantiate such a constraint from $\mathbf{Graph_{TG}}$ in $\mathbf{Sub}(T)$, we will obtain a conjunction of constraints of which each one has the structure (compare Example 6 for this claim)

$$c = \neg\exists\,\big(b_1\colon \emptyset \subseteq B_1, \mathsf{true}\big) \vee \exists\,\big(b_1\colon \emptyset \subseteq B_1, \bigvee_{j=1}^{t} \exists\,(b_2^j\colon B_1 \subseteq B_2^j, \mathsf{true})\big).$$

Flattening such a constraint c (over the identity morphism $id_\emptyset$) proceeds as follows (where, for shorter notation, we merely note down the morphisms and omit the graphs over which they are defined):

$$\mathrm{Flat}(id_\emptyset, c) \overset{(F3)}{=} \mathrm{Flat}\,\big(id_\emptyset, \neg\exists\,(b_1, \mathsf{true})\big) \vee \mathrm{Flat}\,\big(id_\emptyset, \exists\,(b_1, \bigvee_{j=1}^{t} \exists\,(b_2^j, \mathsf{true}))\big)$$
$$\overset{(F6),(F5)}{=} \Big(\exists\,(id_\emptyset, \mathsf{true}) \wedge \neg\,\mathrm{Flat}\,\big(id_\emptyset, \exists\,(b_1, \mathsf{true})\big)\Big) \vee \mathrm{Flat}\,\big(b_1 \circ id_\emptyset, \bigvee_{j=1}^{t} \exists\,(b_2^j, \mathsf{true})\big)$$

$$\overset{(F5),(F3)}{=} \left(\exists\,(id_\emptyset, \mathsf{true}) \wedge \neg\,\mathrm{Flat}\,\big(b_1 \circ id_\emptyset, \mathsf{true}\big)\right) \vee \bigvee_{j=1}^{t} \mathrm{Flat}\,\big(b_1 \circ id_\emptyset, \exists\,(b_2^j, \mathsf{true})\big)$$

$$\overset{(F1),(F5)}{=} \left(\exists\,(id_\emptyset, \mathsf{true}) \wedge \neg\exists\,(b_1 \circ id_\emptyset, \mathsf{true})\right) \vee \bigvee_{j=1}^{t} \mathrm{Flat}\,\big(b_2^j \circ b_1 \circ id_\emptyset, \mathsf{true}\big)$$

$$\overset{(F1)}{=} \left(\exists\,(id_\emptyset, \mathsf{true}) \wedge \neg\exists\,(b_1 \circ id_\emptyset, \mathsf{true})\right) \vee \bigvee_{j=1}^{t} \exists\,(b_2^j \circ b_1 \circ id_\emptyset, \mathsf{true})$$

$$\equiv \neg\exists\,(b_1, \mathsf{true}) \vee \bigvee_{j=1}^{t} \exists\,(b_2^j \circ b_1, \mathsf{true}).$$

This is exactly the structure of the constraint $c_{\mathrm{lb}}^{\mathbf{Sub}(T)}$ in Fig. 6.

Next, though intuitively clear, we prove that applying the flattening construction computes a condition of nesting level ≤ 1. In other words, it outputs a Boolean combination of literals.

Lemma 2 (Flattening results in non-nested condition)**.** *Given a type graph TG, a graph T typed over it, a morphism b_0 with co-domain B_0 in $\mathbf{Sub}(T)$ and a condition c over B_0, then*

$$\mathrm{nl}(\mathrm{Flat}(b_0, c)) \leq 1.$$

In particular, the condition resulting from flattening over a morphism is a Boolean combination of literals.

Next, we show that flattening preserves the semantics of satisfaction in $\mathbf{Sub}(T)$; the proof proceeds via structural induction where the most interesting inductive step can be performed because of Lemma 1.

Theorem 3 (Flattening preserves satisfaction in $\mathbf{Sub}(T)$)**.** *Given a type graph TG, a graph T typed over it, a morphism $b_0 \colon X \subseteq B_0$ in $\mathbf{Sub}(T)$ and a condition c over B_0 in $\mathbf{Sub}(T)$, then, for any graph G in $\mathbf{Sub}(T)$ and any inclusion $\iota_G^X \colon X \subseteq G$,*

$$\iota_G^X \models_{\mathcal{I}} \exists\,(b_0, c) \iff \iota_G^X \models_{\mathcal{I}} \mathrm{Flat}(b_0, c).$$

In particular,

$$\iota_G^{B_0} \models_{\mathcal{I}} c \iff \iota_G^{B_0} \models_{\mathcal{I}} \mathrm{Flat}(id_{B_0}, c).$$

The ability to flatten constraints in $\mathbf{Sub}(T)$ in a semantics-preserving way means we can assume all constraints and conditions to be given in some of the normal forms that are known from propositional logic. The following corollary states this—it just combines the results stated in Lemma 2 and Theorem 3 with the well-known existence of normal forms for propositional logic (see, e.g., [17, Proposition 2.6.1]).

Corollary 4. *Given a type graph TG, a graph T typed over it and a set $\mathcal{C}$ of conditions in $\mathbf{Sub}(T)$, there is a set $\mathcal{C}^{\mathrm{NF}}$ of conditions in $\mathbf{Sub}(T)$ such that (i) all conditions from $\mathcal{C}^{\mathrm{NF}}$ are Boolean combinations of literals over $\mathbf{Sub}(T)$ and are all in the same normal form known from propositional logic (e.g.,* conjunctive *or* disjunctive normal form*) and (ii) there is a function $j \colon \mathcal{C} \to \mathcal{C}^{\mathrm{NF}}$ such that for every $c \in \mathcal{C}$ and every $S \in \mathbf{Sub}(T)$*

$$\iota_S^{B_0} \models_{\mathcal{I}} c \iff \iota_S^{B_0} \models_{\mathcal{I}} j(c),$$

where B_0 is the graph over which condition c is defined and $\iota_S^{B_0} \colon B_0 \subseteq S$.

For our work in [11], the *conjunctive normal form* (CNF) provides a particularly convenient basis. That is, every constraint or condition is a conjunction of disjunctions of literals over $\mathbf{Sub}(T)$. Considering the separate clauses (i.e., disjunctions of literals) as individual conditions further simplifies our work. The following semantic equivalence allows us to further restrict the kinds of conditions we need to consider: we can unite conjunctions of positive literals to a single literal.

Lemma 5 (Union of conjunctions). *Given a type graph TG and a graph T typed over it, let $c_i = \exists\,(b_1^i\colon B_0 \subseteq B_1^i, \mathit{true})$, where $i = 1, \dots, k$, be a finite set of positive literals in $\mathbf{Sub}(T)$. Then, for every $S \in \mathbf{Sub}(T)$,*

$$\iota_S^{B_0} \models_{\mathcal{I}} \bigwedge_{i=1}^{k} c_i \iff \iota_S^{B_0} \models_{\mathcal{I}} \exists\,(b_1\colon B_0 \subseteq \bigcup_{i=1}^{k} B_1^i, \mathit{true})$$

where $\iota_S^{B_0}\colon B_0 \subseteq S$ is the inclusion of B_0 in S and b_1 the inclusion of B_0 in $\bigcup_{i=1}^{k} B_1^i$.

Presentation of Constraints. Wrapping up this section, in a lattice of subgraphs $\mathbf{Sub}(T)$, given a set $\mathcal{C}$ of conditions and constraints, we can assume each of it to be given in one of the following forms.

- the *trivial* cases: $c = \mathsf{true}$ or $c = \mathsf{false}$;
- the *purely positive* case:
$$c = \exists\,(B_1, \mathsf{true}) \vee \dots \vee \exists\,(B_t, \mathsf{true});$$
- the *purely negative* case:
$$\neg\exists\,(D, \mathsf{true}),$$
which—by De Morgan's laws and Lemma 5—covers disjunctions of negative literals:
$$\begin{aligned} c = \neg\exists\,(B_1, \mathsf{true}) \vee \dots \vee \neg\exists\,(B_s, \mathsf{true}) &\equiv \neg(\exists\,(B_1, \mathsf{true}) \wedge \dots \wedge \exists\,(B_s, \mathsf{true})) \\ &\equiv_{\mathcal{I}} \neg\exists\,(D, \mathsf{true}), \text{ where } D \coloneqq \bigcup_{i=1}^{s} B_i; \end{aligned}$$
- the *mixed* or *implication* case:
$$\exists\,(D, \mathsf{true}) \implies \exists\,(B_1, \mathsf{true}) \vee \dots \vee \exists\,(B_t, \mathsf{true}),$$
which—again by De Morgan's laws and Lemma 5—covers clauses that contain both positive and negative literals:
$$\begin{aligned} c &= \exists\,(B_1, \mathsf{true}) \vee \dots \vee \exists\,(B_t, \mathsf{true}) \vee \neg\exists\,(D_1, \mathsf{true}) \vee \dots \vee \neg\exists\,(D_s, \mathsf{true}) \\ &\equiv_{\mathcal{I}} \exists\,(B_1, \mathsf{true}) \vee \dots \vee \exists\,(B_t, \mathsf{true}) \vee \neg\exists\,(\bigcup_{i=1}^{s} D_i, \mathsf{true}) \\ &\equiv \exists\,(D, \mathsf{true}) \implies \exists\,(B_1, \mathsf{true}) \vee \dots \vee \exists\,(B_t, \mathsf{true}), \text{ where } D \coloneqq \bigcup_{i=1}^{s} D_i. \end{aligned}$$

5. Instantiation of Conditions and Constraints

Given a type graph TG and a finite graph T typed over it, specifying formulas in $\mathbf{Sub}(T)$ is inefficient: rather than capturing the logical essence of the constraint, we have to enumerate every possible situation explicitly. Instead, we would like to use the power of quantification and express constraints using the more common nested conditions in $\mathbf{Graph_{TG}}$. In this section, we show how to translate a nested condition from $\mathbf{Graph_{TG}}$ into an equivalent nested condition in $\mathbf{Sub}(T)$. Practically, this increases the usability of nested conditions and constraints in $\mathbf{Sub}(T)$, and, formally, it demonstrates their expressiveness in that context: first-order properties can still be expressed in that logic, even though nesting collapses.[2]

[2]Note that this is not longer true for lattices of subgraphs $\mathbf{Sub}(T)$ where T is infinite. In the CRA example, for instance, when given a graph T containing infinitely many nodes of type Method, the constraint c_{lb} in $\mathbf{Graph_{TG}}$ (see Fig. 5) still expresses that every Method is assigned to a Class. To express its analogue $c_{\mathrm{lb}}^{\mathbf{Sub}(T)}$ in $\mathbf{Sub}(T)$, however, we would generally need the possibility to build an infinite conjunction (of disjunctions), explicitly requiring for every concrete Method to be assigned to one of the available Classes.

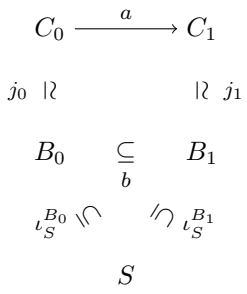


Figure 9: Instantiation of a morphism $a\colon C_0 \to C_1$ in S

Our translation of conditions and constraints is based on *instantiations*. In an instantiation, a graph (or a morphism) from $\mathbf{Graph_{TG}}$ is identified with an isomorphic subgraph (inclusion) in $\mathbf{Sub}(T)$. Such instantiations are the central technical ingredient in the transformation of a nested constraint/condition in $\mathbf{Graph_{TG}}$ into one in $\mathbf{Sub}(T)$.

Definition 5 (Instantiation)**.** Given a type graph TG and a graph T typed over it, an *instantiation* $\mu = (j, \iota_S^B)$ *in* $S \in \mathbf{Sub}(T)$ of a graph C from $\mathbf{Graph_{TG}}$ consists of an isomorphism $j\colon C \xrightarrow{\sim} B$, where $B \subseteq S \in \mathbf{Sub}(T)$ and $\iota_S^B\colon B \subseteq S$ denotes that inclusion; we sometimes also just call the image B of an instantiation μ an instantiation.

An *instantiation in* $S \in \mathbf{Sub}(T)$ of a morphism $a\colon C_0 \to C_1$ from $\mathbf{Graph_{TG}}$ is a morphism $b\colon B_0 \subseteq B_1$ between the images of instantiations $\mu_0 = C_0 \xrightarrow{\sim} B_0 \subseteq S$ and $\mu_1 = C_1 \xrightarrow{\sim} B_1 \subseteq S$ of C_0 and C_1 in S, respectively, such that the diagram in Fig. 9 commutes. For instantiations in $T \in \mathbf{Sub}(T)$ we just speak of *instantiations in* $\mathbf{Sub}(T)$.

Example 5. Consider the graphs C_1 and C_2 from the constraint c_{lb}, depicted in Fig. 5, requiring every Method to be assigned to a Class in the CRA problem. Instantiations of C_1 in the graph $S \subseteq T$ (see Fig. 3) consist of a graph that contains one of the Methods M1, M2 or M3 as its single node, the isomorphism between C_1 and that graph, and the inclusion of that graph into S. Similarly, instantiations of C_2 in S consist of a graph that contains one of the Classes C1 or C2, one of the Methods M1, M2 or M3 and an encapsulates-edge between them, of the isomorphism between C_2 and that graph, and the inclusion of that graph into S. Given instantiations of C_1 and C_2 in S, the instantiation of the morphism $a_2\colon C_1 \hookrightarrow C_2$ is the inclusion between the instantiated graphs in $\mathbf{Sub}(T)$ (if this exists). Instantiations in T (instead of S) are similar, just that in T more Classes are available to map to.

As a rather obvious but important technical tool we note that instantiations correspond to injective morphisms; this will provide us the bridge between the ordinary notion of satisfaction of a condition in $\mathbf{Graph_{TG}}$ and the one in $\mathbf{Sub}(T)$ (denoted as $\models_{\mathcal{I}}$).

Lemma 6 (Correspondence of instantiations and injectives)**.** *Given a type graph TG and a graph T typed over it, for any graph C from $\mathbf{Graph_{TG}}$ and any graph $S \in \mathbf{Sub}(T)$ there exists a one-to-one correspondence between injective morphisms $q\colon C \hookrightarrow S$ in $\mathbf{Graph_{TG}}$ and instantiations $\mu = (j, \iota_S^B)$ of C in S.*

Furthermore, any instantiation $\mu = (j, \iota_S^B)$ of a graph $C \in \mathbf{Graph_{TG}}$ in a graph $S \in \mathbf{Sub}(T)$ can be extended to an instantiation $\mu^T = (j, \iota_T^B)$ of C in $\mathbf{Sub}(T)$.

Furthermore, instantiations of morphisms are determined by the instantiations of their domain and codomain, and they are inclusions themselves.

Lemma 7 (Uniqueness of instantiation)**.** *Let TG be a type graph and T be a graph typed over it. For every morphism $a\colon C_0 \to C_1$ in $\mathbf{Graph_{TG}}$ and all pairs of instantiations $\mu_0 = (j_0, \iota_S^{B_0})$ of C_0 and $\mu_1 = (j_1, \iota_S^{B_1})$ of C_1 in some graph $S \in \mathbf{Sub}(T)$ there is an instantiation of the morphism a if and only if*

$$\iota_S^{B_1} \circ j_1 \circ a = \iota_S^{B_0} \circ j_0. \tag{1}$$

In that case, moreover, the instantiating morphism $b\colon B_0 \subseteq B_1$ in $\mathbf{Sub}(T)$ is unique and an inclusion and the morphism a is injective.

In the following, we will often consider instantiations that stem from an injective morphism q. We will denote such an instantiation as $\mu_q = (q_\mathrm{e}, q_\mathrm{m})$.

The following theorem is the central result of this section, stating that conditions from $\mathbf{Graph_{TG}}$ can equivalently be transformed into conditions from $\mathbf{Sub}(T)$ for finite graphs T typed over TG. The operation Inst appearing in its statement refers to the instantiation of constraints we develop in the following. Together with Theorem 3 this means that, when working in $\mathbf{Sub}(T)$ one can really assume every first-order constraint to be given as discussed at the end of Sect. 4.

Theorem 8 (Instantiation preserves semantics)**.** *Given a type graph TG and a* finite *graph T typed over it, let c be a nested condition in $\mathbf{Graph_{TG}}$ over some graph C_0. Then, for every graph $S \in \mathbf{Sub}(T)$ and every injective morphism $g\colon C_0 \hookrightarrow S$,*

$$g \models c \iff g_\mathrm{m} \models_{\mathcal{I}} \mathrm{Inst}(c, \mu_g^T, \mathbf{Sub}(T))$$

where $\mu_g = (g_\mathrm{e}, g_\mathrm{m})$ is the instantiation of C_0 in S that corresponds to g and μ_g^T is the corresponding instantiation in T (both guaranteed to uniquely exist by Lemma 6).

Next, we develop the instantiation process Inst referred to in Theorem 8. This process relies on two intuitive ideas: First, an existentially quantified graph C is replaced by a disjunction over all ways in which it can be instantiated in $\mathbf{Sub}(T)$. Second, a subcondition $\neg\exists\,(C, d)$ states that at all occurrences of C (that are compatible with the considered instantiation of the graph C_0 over which the condition is defined) the condition d must *not* hold. We therefore replace such subconditions by a conjunction which, for every possible instantiation of C, states that the occurrence of such an instantiation implies the condition $\neg d$. The formal treatment of this second idea is somewhat technical as we need to account for the fact that no occurrence of C might exist and need to take care to still obtain a nested condition over the correct graph. We clarify these details in Lemma 9, after we have developed and illustrated our instantiation process.

Construction 2 (Instantiating conditions and constraints)**.** Given a type graph TG and a finite graph T typed over it, let c be a nested condition over C_0 in $\mathbf{Graph_{TG}}$. Given an instantiation $\mu_0 = (j\colon C_0 \xrightarrow{\sim} B_0, \iota_T^{B_0}\colon B_0 \subseteq T)$ of C_0 in $\mathbf{Sub}(T)$, the *instantiated variant of c* (in $\mathbf{Sub}(T)$ with respect to μ_0), denoted as $\mathrm{Inst}(c, \mu_0, \mathbf{Sub}(T))$, is recursively constructed as follows:

- for $c = \mathsf{true}$, $\mathrm{Inst}\left(c, \mu_0, \mathbf{Sub}(T)\right) = \mathsf{true}$;
- for $c = d_1 \odot d_2$, where $\odot \in \{\wedge, \vee\}$,
$$\mathrm{Inst}\left(c, \mu_0, \mathbf{Sub}(T)\right) = \mathrm{Inst}\left(d_1, \mu_0, \mathbf{Sub}(T)\right) \odot \mathrm{Inst}\left(d_2, \mu_0, \mathbf{Sub}(T)\right);$$
- for $c = \exists\,(a_1\colon C_0 \hookrightarrow C_1, d)$, where d is a condition over C_1,
$$\mathrm{Inst}\left(c, \mu_0, \mathbf{Sub}(T)\right) = \bigvee_{b_1 \in \mathcal{J}} \exists\left(b_1\colon B_0 \subseteq B_1, \mathrm{Inst}\left(d, \mu_1, \mathbf{Sub}(T)\right)\right),$$
where $\mathcal{J}$ indexes instantiations of a_1 with domain μ_0 in $\mathbf{Sub}(T)$ (and μ_1 denotes the respective codomain); the empty disjunction is interpreted as false $= \neg\mathsf{true}$;
- for $c = \neg d$, we make use of a look ahead of size 1 and set $\mathrm{Inst}\left(c, \mu_0, \mathbf{Sub}(T)\right) =$
$$\begin{cases} \mathsf{false}, \text{ for } d = \mathsf{true} \\ \mathrm{Inst}\left(\neg e_1, \mu_0, \mathbf{Sub}(T)\right) \overline{\odot}\, \mathrm{Inst}\left(\neg e_2, \mu_0, \mathbf{Sub}(T)\right), \text{ for } d = e_1 \odot e_2 \\ \mathrm{Inst}\left(e, \mu_0, \mathbf{Sub}(T)\right), \text{ for } d = \neg e \\ \bigwedge\limits_{b_1 \in \mathcal{J}} \Big(\neg\exists\,(b_1\colon B_0 \subseteq B_1, \mathsf{true}) \vee \\ \qquad \exists\left(b_1\colon B_0 \subseteq B_1, \mathrm{Inst}(\neg e, \mu_1, \mathbf{Sub}(T))\right)\Big), \text{ for } d = \exists\,(a_1\colon C_0 \hookrightarrow C_1, e), \end{cases}$$

where, again, $\mathcal{J}$ indexes instantiations of a_1 with domain μ_0, μ_1 denotes the respective codomain, the empty conjunction is interpreted as true, and $\odot \in \{\wedge, \vee\}$, $\overline{\wedge} = \vee$, and $\overline{\vee} = \wedge$.

Given a constraint c, the *instantiated variant of* c *in* $\mathbf{Sub}(T)$, denoted as $\mathrm{Inst}(c, \mathbf{Sub}(T))$, is defined as

$$\mathrm{Inst}(c, \mathbf{Sub}(T)) = \mathrm{Inst}(c, \mu_0, \mathbf{Sub}(T)),$$

where μ_0 is the unique instantiation of $\emptyset$ in T.

Example 6. A kind of constraint one regularly encounters in practice and that is tedious to define in $\mathbf{Sub}(T)$ are constraints of the structure $\forall\,(C_1, \exists\, C_2)$. In our running example of the CRA problem, this concerns the constraint c_{lb}, requiring every Method and Attribute to be assigned to at least one Class (see Fig. 5 in Example 3). Instantiating such a constraint amounts to the following: First, $\forall\,(C_1, \exists\, C_2)$ is syntactic sugar for $\neg\exists\,(a_1 \colon \emptyset \hookrightarrow C_1, \neg\exists\,(a_2 \colon C_1 \hookrightarrow C_2, \mathsf{true}))$. Thus, by construction, we obtain

$$\begin{aligned}\mathrm{Inst}(c_{\mathrm{lb}}, \mathbf{Sub}(T)) = \bigwedge_{b_1 \in \mathcal{J}} \Big(& \neg\exists\,(b_1 \colon \emptyset \subseteq B_1, \mathsf{true}) \vee \\ & \exists\,\Big(b_1 \colon \emptyset \subseteq B_1, \mathrm{Inst}\left(\neg(\neg\exists\,(a_2 \colon C_1 \hookrightarrow C_2, \mathsf{true})), \mu_1, \mathbf{Sub}(T)\right)\Big)\Big),\end{aligned}$$

where $\mathcal{J}$ indexes morphisms b_1 from $\emptyset$ to instantiations of C_1 in T. Continuing the computation, we receive

$$\begin{aligned}& \mathrm{Inst}(\neg(\neg\exists\,(a_2 \colon C_1 \hookrightarrow C_2, \mathsf{true}), \mu_1, \mathbf{Sub}(T)) \\ =\ & \mathrm{Inst}(\exists\,(a_2 \colon C_1 \hookrightarrow C_2, \mathsf{true}), \mu_1, \mathbf{Sub}(T)) \\ =\ & \bigvee_{b_2 \in \mathcal{J}_{b_1}} \Big(\exists\,\big(b_2 \colon B_1 \subseteq B_2, \mathrm{Inst}(\mathsf{true}, \mu_2, \mathbf{Sub}(T))\big)\Big) \\ =\ & \bigvee_{b_2 \in \mathcal{J}_{b_1}} \exists\,\big(b_2 \colon B_1 \subseteq B_2, \mathsf{true}\big),\end{aligned}$$

where $\mathcal{J}_{b_1}$ indexes those instantiations of the morphism a_2 whose domain is instantiated to the codomain of b_1. Thus, in total, we receive a conjunction of constraints of the form

$$\begin{aligned}& \neg\exists\,(b_1 \colon \emptyset \subseteq B_1, \mathsf{true}) \vee \exists\,\Big(b_1 \colon \emptyset \subseteq B_1, \bigvee_{b_2 \in \mathcal{J}_{b_1}} \exists\,\big(b_2 \colon B_1 \subseteq B_2, \mathsf{true}\big)\Big) \\ =\ & \neg\exists\,(b_1 \colon \emptyset \subseteq B_1, \mathsf{true}) \vee \bigvee_{b_2 \in \mathcal{J}_{b_1}} \exists\,\big(b_2 \circ b_1 \colon \emptyset \subseteq B_2, \mathsf{true}\big),\end{aligned}$$

where B_1 ranges over the possibilities to instantiate C_1 in T and B_2 over those to instantiate C_2 in a way conformant to B_1, the instantiation chosen for C_1.

Given T as introduced in Fig. 4 in Example 2, for c_{lb} the graph C_1 can be instantiated in three different ways, mapping its single node of type Method to one of the Methods M1, M2, M3 of T. For each such instantiation of C_1, C_2 (and the morphism from C_1 to C_2) can be instantiated compatibly by mapping its Method in the same way, its Class to one of the Classes C1, …, C6 of T, and the encapsulates-edge accordingly. Figure 6 shows one of the conjuncts of the resulting constraint $c_{\mathrm{lb}}^{\mathbf{Sub}(T)}$, namely the one mapping C_1's node to Method M1.

Like Theorem 3, also Theorem 8 is proved by structural induction. The following lemma, to which we already pointed earlier, is the key technical ingredient in the induction step, clarifying that our instantiation of negated conditions preserves the semantics of conditions.

Lemma 9 (Negation in instantiation)**.** *Given a type graph TG and a finite graph T typed over it, let $c = \neg d$ be a nested condition over C_0 that is also typed over TG. Then, for any instantiation $\mu_0 = (j \colon C_0 \xrightarrow{\sim} B_0, \iota_T^{B_0} \colon B_0 \subseteq T)$ of C_0 in $\mathbf{Sub}(T)$ and any inclusion $\iota_S^{B_0} \colon B_0 \subseteq S$, it holds that*

$$\iota_S^{B_0} \models_{\mathcal{I}} \mathrm{Inst}\left(d, \mu_0, \mathbf{Sub}(T)\right) \iff \iota_S^{B_0} \not\models_{\mathcal{I}} \mathrm{Inst}\left(\neg d, \mu_0, \mathbf{Sub}(T)\right).$$

Finally, we shortly clarify that an instantiated condition still consists of finitely many graphs; in particular, this ensures that the conjunction/disjunction indexed by a set $\mathcal{J}$ of instantiations is well-defined. It also clarifies the severity of the growth of the number of graphs occurring in a condition when instantiating it.

Proposition 10 (Order of number of instantiated constraints/conditions)**.** *Given a type graph TG and a finite graph T that is typed over it, let c be some condition over a graph C_0 that is typed over TG, and let $|c|$ denote the number of morphisms occurring in c. Then, for every instantiation μ_0 of C_0 in $\mathbf{Sub}(T)$,*

$$|\mathrm{Inst}(c, \mu_0, \mathbf{Sub}(T))| \in \mathcal{O}(|c| \cdot |T|^{|C_{\max}|}),$$

where $C_{\max}$ is the largest graph that occurs in c and $|X|$ denotes the number of elements of a graph X. In particular, all conjunctions and disjunctions in $\mathrm{Inst}(c, \mu_0, \mathbf{Sub}(T))$ are finite, i.e., it is a nested condition. Moreover, the total number of instantiated variants of c in $\mathbf{Sub}(T)$ is also in $\mathcal{O}(|c| \cdot |T|^{|C_{\max}|})$.

While the size of $\mathrm{Inst}(c, \mu_0, \mathbf{Sub}(T))$, thus, grows exponentially in the size of the largest graph occurring in c, this tends to be not a problem in practice as graphs in nested graph conditions are usually small.

6. Conclusion

In this paper, we have presented a semantics-preserving translation of nested conditions and constraints from the category $\mathbf{Graph_{TG}}$ (of graphs typed over a given type graph TG) into nested conditions and constraints formulated in a category $\mathbf{Sub}(T)$, the lattice of subgraphs of a fixed finite graph T typed over TG (Theorem 8). In $\mathbf{Sub}(T)$, moreover, every nested condition and constraint can be equivalently transformed into a nesting-free normal form (Theorem 3 and Corollary 4). Together, both results show that in $\mathbf{Sub}(T)$ first-order properties can be expressed by the nesting-free fragment of the formalism of nested conditions and constraints (given finiteness of T).

These results are not surprising—we work in a fixed, finite universe and can replace nesting by explicitly enumerating all concrete situations in a formula. Our theoretical contribution, therefore, consists in making the expected precise, also clarifying details that are slightly involved (the treatment of negation). Practically, however, we see potential for interesting applications of our results: A finite lattice of subgraphs $\mathbf{Sub}(T)$ is a quite natural setting for many problems in model-driven optimisation [15] or model repair [24], and with our work we provide the formal bridge for conveniently specifying constraints as general nested conditions (in $\mathbf{Graph_{TG}}$) but using the far more concrete constraints (in $\mathbf{Sub}(T)$) for downstream tasks. In fact, we have developed the theory in this paper to enable such an application, namely the derivation of non-blocking consistency-preserving rules in a finite lattice of subgraphs $\mathbf{Sub}(T)$ [11].

There are two immediate avenues for generalising our theoretical results: First, in our proofs we do not make use of the fact that our objects are graphs; what we critically use, though (in Sect. 5), is the fact that the graph T, whose subgraphs form our category of interest, is finite and, therefore, there is only a finite number of injective morphisms from a graph C to a subgraph of T. This suggests that our results should directly transfer from graphs to the far more general setting of categories of subobjects of a container object that stems from any *finitary* $\mathcal{M}$-adhesive category [25]. Moreover, in our setting of a finite lattice of subgraphs, it is known that first- and second-order logics coincide (see [26], in particular also for far more interesting graph classes where first-order logic coincides with some fragment of second-order logic). This means that it is possible to develop translations, similar to the one presented here, also for formalisms that have been developed to extend nested conditions and constraints beyond first-order as, e.g., [27, 28, 29].

Acknowledgments

The authors thank Alexander Lauer for helpful comments on a first draft of this paper. This work was

partially funded by the German Research Foundation (DFG), project “Model-Driven Optimization in Software Engineering” (TA 294/19-1).

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. Rensink, Representing first-order logic using graphs, in: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 – October 2, 2004, Proceedings, Springer, 2004, pp. 319–335. doi:10.1007/978-3-540-30203-2_23.

[2] A. Habel, K. Pennemann, Nested Constraints and Application Conditions for High-Level Structures, in: H. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, G. Taentzer (Eds.), Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday, Springer, 2005, pp. 293–308. doi:10.1007/978-3-540-31847-7_17.

[3] A. Habel, K. Pennemann, Correctness of high-level transformation systems relative to nested conditions, Math. Struct. Comput. Sci. 19 (2009) 245–296. doi:10.1017/S0960129508007202.

[4] H. Ehrig, U. Golas, A. Habel, L. Lambers, F. Orejas, $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation, Math. Struct. Comput. Sci. 24 (2014). doi:10.1017/S0960129512000357.

[5] H. Ehrig, U. Golas, A. Habel, L. Lambers, F. Orejas, $\mathcal{M}$-Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence, Fundam. Informaticae 118 (2012) 35–63. doi:10.3233/FI-2012-705.

[6] U. Golas, A. Habel, H. Ehrig, Multi-amalgamation of rules with application conditions in $\mathcal{M}$-adhesive categories, Math. Struct. Comput. Sci. 24 (2014). doi:10.1017/S0960129512000345.

[7] H. Ehrig, C. Ermel, U. Golas, F. Hermann, Graph and Model Transformation – General Framework and Applications, Springer, 2015. doi:10.1007/978-3-662-47980-3.

[8] J. Kosiol, D. Strüber, G. Taentzer, S. Zschaler, Sustaining and improving graduated graph consistency: A static analysis of graph transformations, Sci. Comput. Program. 214 (2022) 102729. doi:10.1016/J.SCICO.2021.102729.

[9] A. Lauer, J. Kosiol, G. Taentzer, Granular conflict analysis for transformation rules with application conditions, in: J. Endrullis, M. Tichy (Eds.), Graph Transformation – 18th International Conference, ICGT 2025, Held as Part of STAF 2025, Koblenz, Germany, June 11–12, 2025, Proceedings, Springer, 2025, pp. 63–90. doi:10.1007/978-3-031-94706-3_4.

[10] B. König, A. Rensink, L. Stoltenow, F. Urrigshardt, Counterexample-guided abstraction refinement for generalized graph transformation systems, in: J. Endrullis, M. Tichy (Eds.), Graph Transformation – 18th International Conference, ICGT 2025, Held as Part of STAF 2025, Koblenz, Germany, June 11–12, 2025, Proceedings, Springer, 2025, pp. 157–176. doi:10.1007/978-3-031-94706-3_8.

[11] J. Kosiol, S. Zschaler, Non-blocking consistency-preserving subgraph transformation systems via activation diagrams, 2026. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6184192, under review.

[12] A. Corradini, F. Hermann, P. Sobocinski, Subobject Transformation Systems, Appl. Categorical Struct. 16 (2008) 389–419. doi:10.1007/s10485-008-9127-6.

[13] F. Hermann, A. Corradini, H. Ehrig, Analysis of permutation equivalence in $\mathcal{M}$-adhesive transformation systems with negative application conditions, Math. Struct. Comput. Sci. 24 (2014). doi:10.1017/S0960129512000382.

[14] J. M. Horcas, D. Strüber, A. Burdusel, J. Martinez, S. Zschaler, We’re Not Gonna Break It!

Consistency-Preserving Operators for Efficient Product Line Configuration, IEEE Trans. Software Eng. 49 (2023) 1102–1117. doi:10.1109/TSE.2022.3171404.

[15] S. John, J. Kosiol, L. Lambers, G. Taentzer, A graph-based framework for model-driven optimization facilitating impact analysis of mutation operator properties, Softw. Syst. Model. 22 (2023) 1281–1318. doi:10.1007/S10270-022-01078-X.

[16] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer, 2006. doi:10.1007/3-540-31188-2.

[17] J. Avigad, Mathematical Logic and Computation, Cambridge University Press, 2022. doi:https://doi.org/10.1017/9781108778756.

[18] K. Pennemann, Development of correct graph transformation systems, Ph.D. thesis, University of Oldenburg, Germany, 2009. URL: https://nbn-resolving.org/urn:nbn:de:gbv:715-oops-9483.

[19] L. Stoltenow, B. König, S. Schneider, A. Corradini, L. Lambers, F. Orejas, Coinductive techniques for checking satisfiability of generalized nested conditions, in: R. Majumdar, A. Silva (Eds.), 35th International Conference on Concurrency Theory, CONCUR 2024, Calgary, Canada, September 9–13, 2024, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 39:1–39:20. doi:10.4230/LIPICS.CONCUR.2024.39.

[20] C. Sandmann, A. Habel, Rule-based graph repair, in: R. Echahed, D. Plump (Eds.), Proceedings Tenth International Workshop on Graph Computation Models, GCM@STAF 2019, Eindhoven, The Netherlands, 17th July 2019, 2019, pp. 87–104. doi:10.4204/EPTCS.309.5.

[21] J. Kosiol, S. Zschaler, A nesting-free normal form for nested conditions in finite lattices of subgraphs – extended version, CoRR abs/2601.18376 (2026). URL: https://doi.org/10.48550/arXiv.2601.18376. doi:10.48550/ARXIV.2601.18376. arXiv:2601.18376.

[22] M. Fleck, J. Troya, M. Wimmer, The Class Responsibility Assignment Case, in: A. García-Domínguez, F. Krikava, L. M. Rose (Eds.), Proceedings of the 9th Transformation Tool Contest, co-located with the 2016 Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 8, 2016, CEUR-WS.org, 2016, pp. 1–8. URL: https://ceur-ws.org/Vol-1758/paper1.pdf.

[23] S. Lack, P. Sobociński, Adhesive and quasiadhesive categories, RAIRO Theor. Informatics Appl. 39 (2005) 511–545. doi:10.1051/ITA:2005028.

[24] N. Macedo, J. Tiago, A. Cunha, A feature-based classification of model repair approaches, IEEE Trans. Software Eng. 43 (2017) 615–640. URL: https://doi.org/10.1109/TSE.2016.2620145. doi:10.1109/TSE.2016.2620145.

[25] K. Gabriel, B. Braatz, H. Ehrig, U. Golas, Finitary $\mathcal{M}$-adhesive categories, Math. Struct. Comput. Sci. 24 (2014). doi:10.1017/S0960129512000321.

[26] M. Elberfeld, M. Grohe, T. Tantau, Where first-order and monadic second-order logic coincide, ACM Trans. Comput. Log. 17 (2016) 25. URL: https://doi.org/10.1145/2946799. doi:10.1145/2946799.

[27] H. Radke, HR* Graph Conditions Between Counting Monadic Second-Order and Second-Order Graph Formulas, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 61 (2013). doi:10.14279/TUJ.ECEASST.61.831.

[28] M. Navarro, F. Orejas, E. Pino, L. Lambers, A navigational logic for reasoning about graph properties, J. Log. Algebraic Methods Program. 118 (2021) 100616. doi:10.1016/J.JLAMP.2020.100616.

[29] F. Drewes, B. Hoffmann, M. Minas, Systems of graph formulas and their equivalence to alternating graph automata, CoRR abs/2510.25260 (2025). doi:10.48550/ARXIV.2510.25260. arXiv:2510.25260.