Incremental Consistency Checking of Heterogeneous Multimodels

Zinovy Diskin[1,2] and Harald König[3(✉)]

[1] NECSIS, McMaster University, Hamilton, Canada
[2] Generative Software Development Lab, University of Waterloo, Waterloo, Canada
zdiskin@uwaterloo.ca
[3] University of Applied Sciences FHDW Hannover, Hannover, Germany
harald.koenig@fhdw.de

Abstract. The local approaches to global consistency checking (GCC) of heterogeneous multimodels strive to reduce the model merging and matching workload within GCC. The paper's contribution to such approaches is a framework allowing the user to do matching incrementally: to build the match required for checking the multimodel w.r.t. a new constraint, the user employs matches produced in previous GCC sessions.

1 Introduction

Modeling a complex system normally results in a *(heterogeneous) multimodel*, i.e., a set of heterogenous (component) models each one conforming to its own metamodel. A fundamental fact about multimodeling is that if even each of the component model perfectly conforms to its metamodel, taken together they may violate some global consistency (GC) rules, i.e., be globally inconsistent [2,7]. An accurate mathematical definition of GC based on model merge was proposed in [9] for the homogeneous case, and extended for the heterogeneous multimodeling in [1]. Moreover, while in [9], the merge-based definition of GC was also used as a practical procedure for GC checking (GCC), in [1] we proposed a more efficient *local* approach, in which consistency is only checked at the overlaps of the component metamodels, which reduces the model merge workload in GCC. The local idea was significantly developed in our paper [4], in which we proposed to check each global constraint c individually, and correspondingly do matching and merging as minimally as required for checking c, i.e., only using those (meta)model elements that affect the validity of c. In this way, not only model merging, but also matching workload is reduced. As model matching is a very expensive procedure, the local approach of [4] provides significant gains for GCC.

The present paper makes a new contribution to the local GCC by reducing the model matching workload even more by doing it incrementally. Suppose that

This work is supported by the Automotive Partnership Canada via the Network on Engineering Complex Software Intensive Systems (NECSIS).

P. Milazzo et al. (Eds.): STAF 2016, LNCS 9946, pp. 274–288, 2016.
DOI: 10.1007/978-3-319-50230-4_21

the user performed GCC of a given multimodel w.r.t. a set of global constraints C, but after that the user needs to make yet another GCC session for a bigger set of constraints $C' \supset C$. We show how the user can effectively perform the new matching procedure required for the latter GCC by using results of the former match rather than building the new match from scratch. In a nutshell, the mathematical framework we develop allows us to transform a constraint increment $C' - C$ into a respective increment in the inter-model correspondence specification.

The paper is structured as follows. Sections 2 and 3 provide the required background: in Sect. 2, we explain the main concepts and challenges of GCC of heterogeneous multimodels with a simple example, and in Sect. 3, we outline our mathematical framework, particularly, the machinery of diagrammatic constraints. Section 4 presents the contribution of the paper—incremental model matching within GCC. In Conclusion we outline directions for future work.

2 Background I: Multimodeling, Global Constraints and Global Consistency

Modeling a complex system normally results in a *multimodel*, i.e., a set of heterogenous models (class diagrams, sequence diagrams, statecharts, activity diagrams, etc.), each one conforming to its own metamodel. For illustrating the main concepts, we will consider a toy example in Fig. 1, which shows two class diagrams $A_{1,2}$, each one conforming to its own metamodel $M_{1,2}$. Metamodel M_1 specifies classes implementing interfaces with operations implemented by methods. Metamodel M_2 says that classes can be abstract, they have attributes, and also implement interfaces. Each of the metamodels has its own constraints, e.g., all directed association are assumed to have multiplicities [0..1] at the target

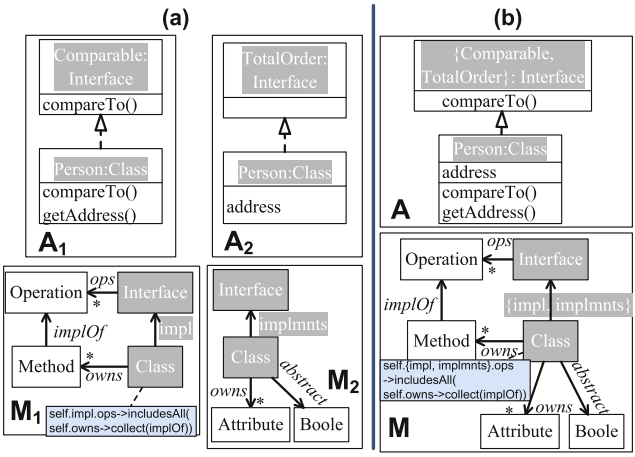


Fig. 1. Sample multimodel

end by default, and the OCL constraint in M_1 prescribes that each implemented operation in a class belongs to this class' implemented interface. In addition, we may want to require that every class owns at least one either method or attribute (or have both). This constraint cannot be declared in any of the two metamodels as M_1 knows nothing about attributes while M_2 knows nothing about methods. Following [1], we call such constraints *inter-metamodel* or *global*; correspondingly, metamodels $M_{1,2}$ and their constraints are called *local*.

What is the metamodel to which a global constraint can be attached? A reasonable answer seems obvious: we need to merge local metamodels into a *global* metamodel M, which in our case can be easily done manually as shown in Fig. 1(b). For this merge, we have silently assumed (1) merging elements of $M_{1,2}$ (i.e., glueing them together) with the same names except two associations *owns*, and (2) merging associations *impl*@M_1 and *implmnts*@M_2 even though they have different names (elements to be merged as well as their merge are shaded in grey). The merged metamodel M clearly violates two basic constraints of the metametamodel: (C1) different associations from the same metaclass must be named differently, and (C2) any element only has one name. Thus, while local metamodels do conform to the metametamodel, their merge does not, and we say that metamodels $M_{1,2}$ are *globally* inconsistent. Fixing global inconsistency in our case is easy: we need to rename homonymic elements (say, into *owns_a* and *owns_m*), and choose one of the synonymic names (say, *impl*) or generate a new one. These fixes are not shown in Fig. 1(b), but below we will assume them done. After the merged metamodel M is built and fixed, we can attach global constraints to it, and check global consistency of the multimodel (A_1, A_2). For this, we need first to merge the local models into a global model A as shown in Fig. 1(b) (again shaded in grey in $A_{1,2}$ and A_0), and then check validity of global constraints for A. Specifically, we see that the "one method or attribute"-constraint described above is satisfied by model A.

In the toy example above, all manipulations were easy, but in practice, merging and checking global consistency may be a far more complicated issue. Specifically, (meta)model matching (together with subsequent merging) are very expensive operations which need intelligent tool support, but anyway cannot be fully automated. A key observation made in [1] and further developed in [4] is that for checking a particular constraint or a group of constraints C, the user can match only small parts of the (meta)models that matter for C's validity rather than match and merge the entire (meta)models. For instance, in the example above, the "one method or attribute"-constraint does not cover interfaces, such that manual effort for the decision whether to match "Comparable" and "TotalOrder" can be omitted.

To make the simple example above generalizable and applicable to practically interesting cases, we need a precise mathematical framework and tools built on the base of such a framework. Specifically, we need a formal specification of models and their merge, metamodels and constraints, and conformance of a model to a metamodel. A suitable mathematical framework is outlined in the next section.

3 Background II: Mathematical Framework

We assume the reader to be familiar with the concept of (directed multi-)graphs and graph morphisms (mappings), which together constitute the category $\mathbb{G}$ of graphs. We also assume some basic knowledge of categories and functors. In Sect. 3.1, we explain model mappings and spans, and in Sect. 3.2 model merge. Section 3.3 explains the fundamental tool for our considerations — diagrammatic constraints, and Sect. 3.4 explains local checking of global constraints in detail. To make the paper self-contained, we sketched some technical material heavily used in the paper in the appendix.

3.1 Model Mappings and Spans

Models' structures are governed by metamodels. Since many models are graphical, this can be formalized via typed graphs by defining a model A as a triple (M_A, G_A, τ_A) with M_A a graph of *types* or A's *metamodel graph*, G_A a graph specifying model's data, and $\tau_A : G_A \to M_A$ a graph morphism called *typing*, which assigns to each model element its type in the metamodel graph M_A (see, for example, models A_1 and A_2 in Fig. 2). We will often omit subindex $_A$ near model's components. In a homogeneous environment determined by a single metamodel M, all models are typed over M and thus are pairs (G, τ). We will also use the latter notation if M is clear from the context. We will denote the class of all models over M by MODEL$[M]$.

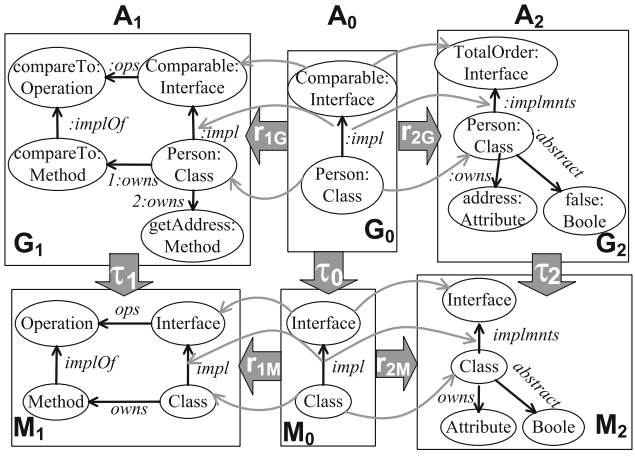
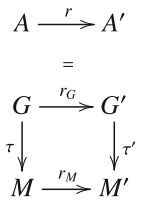


Fig. 2. Model overlapping via spans



The following notion is fundamental for our work with heterogeneous models. A *model mapping* or *morphism* $r: A \to A'$ is a pair (r_G, r_M) of graph morphisms $r_G: G \to G'$ and $r_M: M \to M'$ such that the inset diagram commutes, i.e. $\tau; r_M = r_G; \tau'$. For example, Fig. 2 presents two model mappings, $r_1 = (r_{1G}, r_{1M}): A_1 \leftarrow A_0$ and $r_2 = (r_{2G}, r_{2M}): A_0 \to A_2$. Note the importance of commutativity, which enforces mapping models' data elements to preserve their types. We will often omit subindexes $_{M,G}$ if they are clear from the context.

Three special types of model maps are important. Two models are *isomorphic*, written $A \cong A'$, if both mappings r_M and r_G are isomorphisms. Model A is a *submodel* of A', written $A \hookrightarrow A$, if both r_M and r_G are inclusions. Finally, if the square in the inset diagram above is a pullback (see Appendix), then we write $r: A \overset{\mathsf{pb}}{\to} A'$ and call model A the *(retyped) restriction (*or *reduction)* of model A' *along* map r_M.

Model overlap can be specified by a pair of model mappings $A_1 \xleftarrow{r_1} A_0 \xrightarrow{r_2} A_2$ with a common source as illustrated in Fig. 2 (curved arrows denote mapping behavior). Such a configuration of models and mappings is called a *span*; the common model is the *head* of the span, and the two mappings are its *legs*. In more detail, a model span consists of two graph spans: a metamodel span $M_1 \xleftarrow{r_{1M}} M_0 \xrightarrow{r_{2M}} M_2$ and a data span $G_1 \xleftarrow{r_{1G}} G_0 \xrightarrow{r_{2G}} G_2$. In each of the graph spans, an element x in the head represents a common/shared concept, while legs show how this concept is represented in each of the components. For example, each element $x \in M_0$ declares that elements $r_{1M}(x) \in M_1$ and $r_{2M}(x) \in M_2$ refer to the same (meta)classifier. Particularly, associations *impl* in metamodel M_1 and *implmnts* in M_2 are declared to be the same in Fig. 2 despite their different names. Analogously, the upper span declares that classes $Person@A_1$ and $Person@A_2$ refer to the same class. Note that it is no restriction to assume that the overlap span is *jointly injective*, i.e., for any two elements $x, x \in M_0$, if $r_{1M}(x) = r_{1M}(x')$ and $r_{2M}(x) = r_{2M}(x')$, then $x = x'$.

When a span specifies a model overlap, we will refer to it as an *overlap* or *correspondence* span. Thus, the metamodel of our sample multimodel is actually a span $\mathcal{M} = (M_1, M_2, M_0, r_{1M}, r_{2M})$ or shorter $\mathcal{M} = (r_{1M}, r_{2M})$ rather than a pair (M_1, M_2), and the multimodel itself is a span $\mathcal{A} = (A_1, A_2, A_0, r_1, r_2)$ or shorter $\mathcal{A} = (r_1, r_2)$ rather than a pair (A_1, A_2). We will call (A_1, A_2) the *base* of multimodel $\mathcal{A}$. Thus, a multimodel is essentially richer than its base (cf. [1]).

3.2 Model Merge and Global Constraints

After model overlap is specified by a span, we can merge the component models in an entirely automatic way by employing an operation called *pushout (PO)*. Figure 3 explains the idea by showing how the two metamodels are merged. Intuitively, we first take the disjoint union of M_1 and M_2, and then glue together those elements, which are declared to be the same by the span. The result is a merged graph M together with two mappings $\overline{r_1}: M_1 \to M$ and $\overline{r_2}: M \leftarrow M_2$

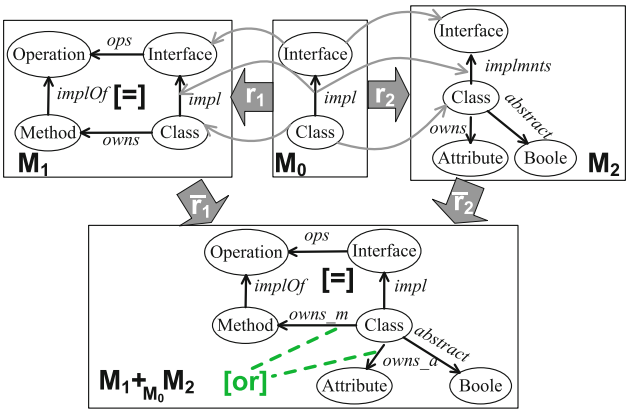


Fig. 3. Merging metamodels (Color figure online)

specifying embedding of the local metamodel graphs into the merge. We will denote it by $M_1 +_{M_0} M_2$.

Local constraints are directly carried into the merged graph along the maps $\overline{r_1}$ and $\overline{r_2}$, in this way the commutativity constraint (note the label [=]) and multiplicities (not shown) are carried into the merge. Thus, PO takes a span of metamodels as its input, and outputs a cospan (two mappings with a common target), encompassing all data from the local metamodels without duplication. Models' data graphs are also merged with PO, and it can be shown that the result of data graph PO is properly typed over the metamodel graph PO (we omit the figure to save space). However, as our discussion in Sect. 2 shows, some constraints can be violated and have to be checked. In addition, inter-metamodel constraints may be added to the merged metamodel, e.g. the above mentioned "one method [or] attribute" constraint shown in green in Fig. 3.

3.3 Diagrammatic Constraints

A key feature of constraints used in metamodeling is their *diagrammatic* nature: the set of elements over which a constraint is declared is actually a diagram of some shape specific for the constraint. For example, the shape of any multiplicity constraint is a single arrow, while the shape of constraint [or] discussed above is a span of two arrows.

Table 1. Sample constraints

Name	Shape
[0..1]	① 12 ②
[or]	① 01 ← 0 → 02 ②
[=]	0 → 1 (01), 0 → 2 (02), 1 → 2 (12)

To declare a constraint named c over a metamodel graph M, we recognize the constraint shape in the graph and label the respective configuration by constraint name c. Formally, we first declare a *signature* of constraints, i.e., a set of constraint names/labels, each one assigned with its *(arity) shape* denoted, for a constraint c, by S^c. For example, Table 1

specifies a simple signature consisting of three constraints. Now, to declare a constraint c over a graph M, we need to specify a graph morphism $\delta\colon S^c \to M$ called *(shape) binding*. E.g. in Fig. 4, constraint $c = [or]$ is declared via binding δ with $\delta(01) = owns\_m$, $\delta(02) = owns\_a$, which automatically implies $\delta(1) = Method$, $\delta(0) = Class$, $\delta(2) = Attribute$. The elements in M the shape is mapped to, is called the *image* or the *scope* of the binding; in Fig. 4 the elements beyond the scope are veiled. The same formal mechanisms underlines commutativity constraint in Fig. 3: labeling an arrow square by [=] is a syntactic sugar for adding the diagonal arrow, and declaring the constraint [=] from Table 1 for the two triangles (by mapping the triangle shape to the respective triangle in the graph).

The pair (c, δ) is called a *constraint declaration*. In the sequel, we write $c@\delta$, meaning that constraint c is imposed on metamodel M at the image of binding map δ.

Constraint name "or" already suggests its semantic interpretation in this context: "Each class shall own at least a method *or* an attribute". Importantly, *semantics* of a constraint is, in general, defined irrespective to the binding by defining a *validating* function $\text{VALIDATE}_c(X : \text{MODEL}[S^c])$: BOOLEAN which inputs a typed graph $X = (G_X, \tau_X : G_X \to S^c)$, i.e. a model typed over c's shape, and outputs Boolean truth iff the model is considered to be satisfying the constraint. The validating function must be stable under isomorphism: if $X \cong X'$, then $\text{VALIDATE}_c(X) = \text{VALIDATE}_c(X')$.

Now checking consistency of model $A = (G, \tau : G \to M)$ against a fixed constraint declaration $c@\delta$ in M is performed by function

$$\text{CHECK}(A\colon \text{MODEL}[M], c@\delta\colon \text{CONSTR})\colon \text{BOOLEAN}$$

which performs three steps:

1. *Restrict* A to elements, whose types are in the image of δ in M.
2. *Retype* elements of this new structure to formal typing over S^c. This yields typed graph $A^{c@\delta} = (G^{c@\delta}, \tau^{c@\delta})$.
3. Return the result of $\text{VALIDATE}_c(A^{c@\delta})$.

In Fig. 4 the steps of function CHECK can be tracked: VALIDATE_c acts on models typed over S^c: It returns true if each element of type 0 in $G^{c@\delta}$ has an outgoing edge to some element of type 1 or to some element of type 2. Graph G is restricted and retyped by pulling back τ along δ, $\tau^{c@\delta}$ is the retyping.

Model A satisfies $c@\delta$, written $A \models c@\delta$, if

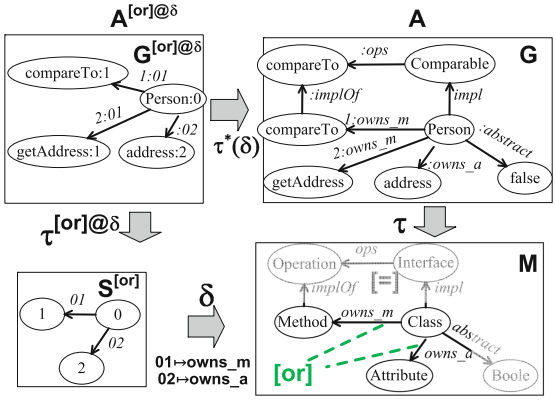


Fig. 4. Constraint declaration and check

CHECK$(A, c@\delta)$=*true*. A is a *legal* model over metamodel M, if $A \models c@\delta$ for all constraints $c@\delta$ declared in M.

The framework described above allows us to give an accurate formal *definition* of global consistency. In a nutshell, we specify local (meta)model overlap by a span, then merge using PO, specify global constraints over the the merged metamodel, and finally check the merged model against global constraints.

3.4 Global Consistency Revisited: Local Constraint Checking

As mentioned above, using this definition of global consistency as an algorithm for consistency checking is very inefficient due to the expensive operation of model matching. A better technique is given in [4]: Let $\mathcal{A}$ be a multimodel with base (A_1, A_2), $A_i = (G_i, \tau_i\colon G_i \to M_i)$ $(i = 1, 2)$ defined over a multimetamodel $\mathcal{M} = (M_1, M_0, M_2, r_1, r_2)$. An inter-metamodel constraint $c@\delta$ is verified as follows (see Appendix for how the pullback operation works).

1. *Binding projection:* Identify those fragments of M_1, M_2, and M_0, that matter for checking, by pulling δ back along $\overline{r}_1$, along $\overline{r}_2$, and along $r_1;\overline{r}_1$ (or, equivalently, along $r_2;\overline{r}_2$ as the square is commutative), cf. Fig. 4. This results in mappings $\overline{r}_1^*(\delta) : S_1^{c@\delta} \to M_1$, $\overline{r}_2^*(\delta) : S_2^{c@\delta} \to M_2$, and $(r_1;\overline{r}_1)^*(\delta) : S_0^{c@\delta} \to M_0$. Let's call these maps *localised bindings* (of δ to M_1, M_2, M_0. resp.). Let

$$S_i^{c@\delta} \xrightarrow{k_{iM}} I_i^{c@\delta} \hookrightarrow M_i \quad \text{for } i = 1, 2, 0$$

   be their epi-mono-factorisations, i.e.$I_i^{c@\delta}$is the image of δ's localised binding to M_i.
2. *Model restriction and retyping:* Carry out steps one and two of the constraint checking algorithm of Sect. 3.3 *locally*, i.e. construct consecutive pullbacks of τ_i along the two morphisms of the above epi-mono-factorisation, yielding

$$B_i^{c@\delta} \overset{\text{pb}}{\xrightarrow{k_i}} A_i^{c@\delta} \overset{\text{pb}}{\hookrightarrow} A_i$$

   The right pullback yields subgraph $A_i^{c@\delta}$ of A_i comprising those model elements that are typed in $I_i^{c@\delta}$, the left pullback retypes elements of $A_i^{c@\delta}$ such that they are typed over $S_i^{c@\delta}$. Let $A_i^{c@\delta} = (G_i^{c@\delta}, \tau_i^{c@\delta}\colon G_i^{c@\delta} \to I_i^{c@\delta})$ and provide the modeler with model data $G_1^{c@\delta}$ and $G_2^{c@\delta}$.[1, 2]
3. *Matching:* Determine compatibly typed overlap $A_0^{c@\delta} = (G_0^{c@\delta}, \tau_0^{c@\delta}\colon G_0^{c@\delta} \to I_0^{c@\delta})$ of these two data graphs, including correspondence span $r_1'\colon A_0^{c@\delta} \to A_1^{c@\delta}$ and $r_2'\colon A_0^{c@\delta} \to A_2^{c@\delta}$.[3] We will refer to the triple $(A_0^{c@\delta}, r_1', r_2')$ as a *constraint specific (correspondence) span* and denote it by $\mathsf{span}(c@\delta)$.

[1] In Fig. 3, $G_1^{c@\delta}$ comprises classes, interfaces, and operations; $G_2^{c@\delta}$ contains classes, interfaces, and attributes.

[2] Recall the fact that $B_i^{c@\delta}, A_i^{c@\delta}$, and A_i are typed graphs, such that the arrows of the form $\overset{\text{pb}}{\to}$ depict morphism *pairs* in a pullback square, cf. Sect. 3.1.

[3] Hence, in Fig. 3, $G_0^{c@\delta}$ contains only certain classes.

4. *Validation:* Compute pullback $B_0^{c@\delta} = (H_0^{c@\delta}, \sigma_0^{c@\delta}\colon H_0^{c@\delta} \to S_0^{c@\delta})$ of this overlap along k_{0M} and apply $validate_c(B_1^{c@\delta} +_{B_0^{c@\delta}} B_2^{c@\delta})$.

The key point of this algorithm is that the constraint-tailored correspondence span $\mathsf{span}(c@\delta)$ can be much smaller than the span specifying all correspondences between the component models. E.g. checking our sample multimodel against the constraint "One attribute or one method" specified in Sect. 2 requires to match classes in models A_1 and A_2 while matching interfaces is not necessary. For this toy example, the difference is not significant, but for practical models comprising thousands of elements, the performance gain is essential.

4 From Constraints to Model Matching, Incrementally

This section introduces the main contribution of the paper. Since the most expensive step in the algorithm of Sect. 3.4 is model matching (Step 3), we focus on minimizing this effort by computing the required correspondence span incrementally. The idea is briefly explained in Sect. 4.1, Sect. 4.2 describes our main technical vehicle for the constraint grouping task, and Sect. 4.3 explains incrementality in detail.

4.1 Incrementality in a Nutshell

Suppose we need to check global consistency wrt. a set of constraints

$$C = \{c_1@\delta_1, ..., c_n@\delta_n\}.$$

In the next section we will show that any such set gives rise to a constraint declaration $c@\delta$ for some new constraint symbol c with a new binding map δ such that for any multimodel $\mathcal{A}$ we have $\mathcal{A} \models c@\delta$ iff $\mathcal{A} \models C$ (where, as usual, $\mathcal{A} \models C$ means $\mathcal{A} \models c_i@\delta_i$ for all $i = 1, .., n$). We will denote this new constraint declaration by $\bigwedge C$ and call it *consolidation* of C. Thus, we can replace checking $\mathcal{A}$ against C by checking it against a single constraint $\bigwedge C$ with our algorithm in Sect. 3.4, so that model matching is reduced to discovering the correspondence span $\mathsf{span}(C) \stackrel{\text{def}}{=} \mathsf{span}(\bigwedge C)$.

Assume now that new constraints are added to group C resulting in a bigger group $C' \supseteq C$. To check $\mathcal{A}$ against C', we need to build a correspondence span $\mathsf{span}(C')$, which, as we mentioned several times, is an expensive procedure. Our idea is to build $\mathsf{span}(C')$ incrementally (rather than from scratch) using the previously built correspondence span $\mathsf{span}(C)$. Indeed, we will define a "delta" span $\mathsf{span}(C, C')$ and an operation $\uplus$ of span union such that $\mathsf{span}(C') = \mathsf{span}(C) \uplus \mathsf{span}(C, C')$, so that GCC can be done incrementally with an effective reuse of the model matching knowledge.

4.2 Constraint Grouping

Logical programming enables definition of new formulas with the help of conjunction of already known formulas, e.g.

$$pythagoreanTriple(x, y, z) \;:=\; (x^2 + y^2 = z^2) \wedge isInteger(x) \wedge isInteger(y).$$

This classical *consolidation* of three small formulas by defining their conjunction can be carried out in the same way with diagrammatic constraints: For the sake of simplicity we explain the idea for two constraint declarations only. The general case of an arbitrary (finite) number is straightforward. Let $c_1@\delta_1$ and $c_2@\delta_2$ be imposed on metamodel M. We can define a new constraint symbol $c_1 \wedge c_2$ (read "c_1 and c_2") with arity graph $S^{c_1 \wedge c_2} := S^{c_1} + S^{c_2}$, i.e. the *coproduct* of the two arity graphs (whenever, in the sequel, a term is printed in italics, we refer to the appendix' terminology). In the classical case this corresponds to the disjoint union of all variable slots in the atomic formulae: We obtain 5 slots $s_1, \ldots, s_5$ for the arity of the consolidated formula. For the diagrammatic conjunction of $c_1@\delta_1$ and $c_2@\delta_2$ we take $[\delta_1, \delta_2] : S^{c_1} + S^{c_2}$ (*universal* morphism) to be the corresponding binding map. In the classical example above, this means that the slots are mapped $s_1 \mapsto x$, $s_2 \mapsto y$, $s_3 \mapsto z$, $s_4 \mapsto x$, $s_5 \mapsto y$, placing x, y, z accordingly into the slots.

Semantics of $c_1 \wedge c_2$ is defined as follows. For any model X with $\tau_X : G_X \to S^{c_1 \wedge c_2}$, we set $X \models c_1 \wedge c_2$ iff $i^*_{c_1}(X) \models c_1$ and $i^*_{c_2}(X) \models c_2$, where $i_{c_1} : S^{c_1} \to S^{c_1 \wedge c_2}$ and $i_{c_2} : S^{c_2} \to S^{c_1 \wedge c_2}$ are the coproduct's *canonical injections* (recall that $S^{c_1 \wedge c_2} = S^{c_1} + S^{c_2}$), and $i^*_{c_1}(\_)$, $i^*_{c_2}(\_)$ are the respective PB operations (acting, in fact, on τ_X — see Appendix). Stability under isomorphisms is obvious.

$(c_1 \wedge c_2)@[\delta_1, \delta_2]$ is called a *consolidated constraint declaration* (composed of $c_1@\delta_1$ and $c_2@\delta_2$). Note that in the partially ordered (by $\models$) set of all constraint declarations, $(c_1 \wedge c_2)@[\delta_1, \delta_2]$ is the g.l.b. of $c_1@\delta_1$ and $c_2@\delta_2$.

The construction defined above for the case of two constraint declarations in the group, is directly generalized for the case of any finite number of constraints $C = \{c_1@\delta_1, \ldots, c_n@\delta_n\}$. We will denote the corresponding consolidated constraint by $\bigwedge C$.

Theorem. Given a set of global constraints $C = \{c_1@\delta_1, \ldots, c_n@\delta_n\}$ (declared over the metamodel merge), let $\bigwedge C$ be its consolidated constraint declaration as defined above. Then $\mathcal{A} \models C$ iff $\mathcal{A} \models \bigwedge C$.

4.3 From Constraints to Correspondence Spans

Given a constraint declaration $c@\delta$, let $S^{c@\delta} \xrightarrow{k_M} I^{c@\delta} \hookrightarrow M$ be its epi-mono factorisation as described in Sect. 3.4.

Definition. Given two constraints, $c@\delta$ and $c'@\delta'$, we say the latter *(semantically) entails* the former, and write $c'@\delta' \models c@\delta$, if $I^{c@\delta} \subset I^{c'@\delta'}$ and $\mathcal{A} \models c@\delta$ for any multimodel $\mathcal{A}$ with $\mathcal{A} \models c'@\delta'$.

Corollary. Given a metamodel M, the space of all constraint declarations over M is a (thin) category, say, CONSTR(M), whose arrows are entailments. [4] □

Specifically, it is easy to see that given two groups of constraints such that $C \subset C'$, we have $\bigwedge C' \models \bigwedge C$ for their consolidations. This is our main motivating example, but proofs are easier to build in a bit more general situation of semantic entailment.

Given entailment $c'@\delta' \models @\delta$, we have a diagram

$$S^{c'@\delta'} \qquad S^{c@\delta} \longrightarrow I^{c@\delta} \hookrightarrow I^{c'@\delta'} \hookrightarrow M \tag{1}$$

with $I^{c@\delta}$ and $I^{c'@\delta'}$ being images of δ and δ' resp., which gives rise (through backward propagation) to inclusions

$$A_1^{c@\delta} \subseteq A_1^{c'@\delta'} \text{ and } A_2^{c@\delta} \subseteq A_2^{c'@\delta'}$$

where models A with subindexes are constraint-specific restrictions of local models produced in Step 2 of the algorithm (Sect. 3.4). Thus, there will be further automation potential for matching in Step 3, if two elements $x_1 \in A_1^{c@\delta}$ and $x_2 \in A_2^{c@\delta}$ are declared to be the same: In this case neighbors (reachable via an edge in the data graph) $y_1 \in A_1^{c'@\delta'} - A_1^{c@\delta}$ (of x_1) and $y_2 \in A_2^{c'@\delta'} - A_2^{c@\delta}$ (of x_2) are likely to be identical, too.

We demonstrate the effects for the simple situation of a singleton $C = \{c_1@\delta_1\}$ and $C' = C \cup \{c_2@\delta_2\}$. Consider for this the metamodel merge M in Fig. 3. Suppose again that classes shall either possess an attribute or a method (constraint $c_1@\delta_1$), and, additionally, the following property (constraint $c_2@\delta_2$) has to hold for any class c:

$$(\sim c.abstract \wedge c.impl = i) \text{ implies } (\forall op \in i.ops\colon \exists m \in c.owns\_m\colon m.implOf = op)$$

i.e. each operation of an implemented interface has to be instantiated in each concrete class. Let $c'@\delta'$ be the consolidation of C'. Its scope consists of the complete merge M in Fig. 3. For applying our algorithm for checking validity of (A_1, A_2) against $c_1@\delta_1$, the user has to specify sameness of model elements. Since the image of δ_1 only covers *Class* in the complete overlap of M_1 and M_2, the user only needs to match classes. Thus, in Fig. 2, he will declare classes *Person* to be the same. In contrast, extended constraint declaration $c'@\delta'$ covers the complete overlap M_0 in Fig. 3. Hence the user, additionally, has to specify sameness of interfaces. Since *Person*-classes have already been matched, it is likely that interfaces *Comparable* and *TotalOrder* are the same, and the system can propose their matching to the user, which he can confirm or reject.

In the rest of the section, we investigate the nature of mapping `span`, which maps a constraint $c@\delta$ to its specific correspondence span. We will show that it can be extended to arrows by mapping an entailment $c'@\delta' \models c@\delta$ to the respective inclusion of correspondence spans. The latter can be seen as an increment for model matching.

[4] A thin category is nothing but a partially preordered (big) set: for any pair of objects, the set of mediating arrows between them is either empty or a singleton.

It is easy to verify that image inclusion of two constraints faithfully propagates back to the local metamodels and its overlap by *Preservation* properties of pullbacks. Thus diagram (1) is fully propagated back to mappings with codomain M_1, M_2, and M_0 in step 1 of the algorithm in Sect. 3.4, meaning that we get the same shaped diagram (including image properties) for the localised bindings:

$$S_i^{c'@\delta'} \qquad S_i^{c@\delta} \longrightarrow I_i^{c@\delta} \hookrightarrow I_i^{c'@\delta'} \hookrightarrow M_i \tag{2}$$

for all $i \in \{0,1,2\}$. In step 2, pullback of τ_i along these mappigs ($i \in \{1,2\}$) is carried out. If verification of $c@\delta$ and $c'@\delta'$ would be performed simultaneously, the system would present to the modeler typed graphs $(A_i^{c@\delta}) \overset{\mathsf{pb}}{\hookrightarrow} (A_i^{c'@\delta'})$ for $i \in \{1,2\}$ where inclusion is provided by *preservation* properties and one can show that the pullback property arises from its *decomposition* property (see Appendix). Suppose the modeler has already specified model overlap $A_0^{c@\delta} = (G_0^{c@\delta}, \tau_0^{c@\delta})$ for checking $c@\delta$, then the question is, how to efficiently fill the gaps (question marks and dashed arrows) in

$$\begin{array}{ccccccc}
A_1^{c@\delta} & \xleftarrow{\quad} & A_0^{c@\delta} & \xrightarrow{\quad} & A_2^{c@\delta} & = & \mathsf{span}(c@\delta) \\
\mathsf{pb}\downarrow & r_1' & \mathsf{pb}\downarrow & r_2' & \mathsf{pb}\downarrow & & \downarrow \\
A_1^{c'@\delta'} & \prec\!\!-? & A_0^{c'@\delta'} & ?\!\!\succ & A_2^{c'@\delta'} & = & \mathsf{span}(c'@\delta')
\end{array} \tag{3}$$

Whereas the two horizontal dashed correspondence morphisms declare the extended overlap, the vertical dashed line guarantees coherence with the overlap w.r.t. $c@\delta$.

Note that for any solution $A_0^{c'@\delta'} := (G_0^{c'@\delta'}, \tau_0^{c'@\delta'} : G_0^{c'@\delta'} \to I_0^{c'@\delta'})$ the codomain $I_0^{c'@\delta'}$ is already known, cf. (2). Thus, we have to find $G_0^{c'@\delta'}$ and its typing. We claim that $G_0^{c'@\delta'}$ is of the form $G_0^{c@\delta} + G_0$, where G_0 can be any subset of elements of

$$\{(x_1,x_2) \in G_1^{c'@\delta'} \times G_2^{c'@\delta'} \mid \exists t_0 \in I_0^{c'@\delta'} - I_0^{c@\delta} : \tau_1^{c'@\delta'}(x_1) = r_1(t_0) \wedge \tau_2^{c'@\delta'}(x_2) = r_2(t_0)\},$$

which turns $G_0^{c'@\delta'}$ into a legal graph. We call G_0 the *match-extension* and define $\tau_0^{c'@\delta'} = \tau_0^{c@\delta}$ on $G_0^{c@\delta}$ and $\tau_0^{c'@\delta'}(x_1,x_2) = t_0$. Note that this is unique since we assumed in the beginning of Sect. 3.2 r_1 and r_2 to be jointly injective. Moreover the correspondence maps must be taken to be projections $(x_1,x_2) \mapsto x_1$ and $(x_1,x_2) \mapsto x_2$ on match-extension and such that they coincide with r'_{1G} and r'_{2G} on $G_0^{c@\delta}$. Finally the model part of the vertical dashed map is the inclusion of $G_0^{c@\delta}$ into $G_0^{c'@\delta'}$. It can now be shown that $G_0^{c'@\delta'}$ is indeed a graph, the above diagram becomes commutative, all mappings on the model level are proper graph morphisms and are compatibly typed, and the three vertical arrows are inclusion pullbacks, as desired.

Thus, in the example above, $G_0^{c'@\delta'} = G_0^{c@\delta} + G_0$, where graph $G_0^{c@\delta}$ has exactly one node *Person*. For graph G_0, there are three cases:

1. $G_0 = \emptyset$ (no extension)
2. $G_0 = \{(Comparable : Interface, TotalOrder : Interface)\}$.
3. $G_0 = \{(Comparable : Interface, TotalOrder : Interface), (impl_1, impl_2)\}$, where $impl_1$ specifies that *Person* implements *Comparable* and $impl_2$ specifies that *Person* implements *TotalOrder*.

The second case results in a double declaration of $Comparable = TotalOrder$ to be implemented by *Person*, which can automatically be rejected by the algorithm. In addition to that, the algorithm can propose the third case, because it is likely that *Comparable* and *TotalOrder* can be declared to be the same, since otherwise *Person* should not be in the original overlap (because then it implements different behavior). The user must only confirm this choice. If he rejects, the algorithm outputs case 1.

Given a multimodel base (A_1, A_2) over the multimetamodel $\mathcal{M}$, the construction described by diagram (3) defines mappings between model correspondence spans over $\mathcal{M}$, which makes the space of spans a category $\textsc{Span}(\mathcal{M})$. We can summarize our discussion by formulating an important requirement to the model matching tool: in order to preserve the matching knowledge, mapping $\mathsf{span} : \textsc{Constr}(M) \to \textsc{Span}(\mathcal{M})$ should be a functor. This requirement is well aligned with matching algorithms based on similarity flooding [6]: global constraints provide information about model correspondences, which can be used for matching (e.g., as it was done in our example above).

5 Conclusion: Future Work

We plan to extend the functorial nature of mapping $\mathsf{span} : \textsc{Constr}(M) \to \textsc{Span}(\mathcal{M})$ towards a richer structure over the spaces. Namely, we want to make them lattices formed by Boolean logical operations for the former space, and by Boolean operations over spans for the latter space. Then it should be possible to establish a structure compatible map (homomorphism) from the former algebra to the latter, which would allow the user to do matching in a compositional way with extensive reuse. Another direction is to investigate interaction between our incremental approach and similarity flooding matching algorithms, which potentially can enhance tools for both model matching and global consistency checking. We also plan to develop a tool support for the approach in collaboration with the Bergen group, whose ongoing work on tooling for diagrammatic constraint checking, and subsequent model repairing [5,8] looks very promising.

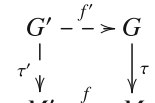
A Appendix. Some Operations Over Graphs and Models

Two operations over graphs and graph morphisms heavily employed in the paper are sketched below; a detailed specification can be found in, say, [3].

Coproducts. The *coproduct* $G_1 + G_2$ of two graphs G_1, G_2 is their disjoint union. Importantly, any coproduct is endowed with two canonic injections $i_k : G_k \hookrightarrow G_1 + G_2$, $k = 1, 2$, which map each element to itself in the union.

Any pair of graph morphisms $f_{1,2} : G_{1,2} \to H$ gives rise to a unique morphism $[f_1, f_2] : G_1 + G_2 \to H$ compatible with injections: $i_{1,2}; [f_1, f_2] = f_{1,2}$. This property of coproducts is called *universality* and morphism specified above *universal*. It is easy to see that universality allows us to define the following operation over models (typed graphs): having typed graphs A_1, A_2 we define $A = A_1 + A_2$ by setting $G_A = G_1 + G_2$, $M_A = M_1 + M_2$ and $\tau_A = [\tau_1; i_1, \tau_2; i_2]$ where $i_{1,2} : M_{1,2} \hookrightarrow M_A$ are coporduct injections.

Restriction/Retyping and Pullbacks. Given a model $A = (M, G, \tau)$ and a type graph map as shown in the inset diagram, we can define a new model $A' = (G', \tau')$ over M' by setting $G' = \{e = (t, x) |\ t \in M', x \in G, f(t) = \tau(x)\}$ [5] with projection mappings $\tau'(e) = t$ and $f'(e) = x$. Further in the paper, we will often denote map f' by $\tau^*(f)$ and say that it is obtained by *pulling f back along τ*, and similarly $\tau' = f^*(\tau)$ is obtained by pulling τ back along f; correspondingly, the entire operation of producing a span (τ', f') from cospan (τ, f) is called *pull-back(PB) (of graphs)*.



If f is inclusion, then PB provides the *(retyped) restriction* of model A over the M' part of the metamodel graph. Pullback operation can be seen as a generalization of model restriction for arbitrary mappings f, and we will often call it so. As any PB square is commutative, we can consider it as a special model morphism, which we will denote by a special arrow $f : A' \overset{\text{pb}}{\to} A$.

Preservation properties. It is known that If f is inclusion, injective or surjective, then f' is, resp., inclusion, injective or surjective as well.

Pullback composition and decomposition. Given $f : A \overset{\text{pb}}{\to} B$ and $g : B \overset{\text{pb}}{\to} C$, their composition is also PB, i.e., $f; g : A \overset{\text{pb}}{\to} C$. Moreover, given that the second arrow and the composition are PBs, $f; g : A \overset{\text{pb}}{\to} C$ and $g : B \overset{\text{pb}}{\to} C$, it is possible to prove that the first arrow is also PB, $f : A \overset{\text{pb}}{\to} B$.[6]

Coproducts and pullbacks (Extensivity). Given three typed graphs and morphism pairs $A_1 \longrightarrow A_0 \longleftarrow A_2$, then $A_1 \overset{\text{pb}}{\to} A_0 \overset{\text{pb}}{\leftarrow} A_2$ if and only if $A_0 \cong A_1 + A_2$.

References

1. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21210-9_16

[5] It is easy to show that G' is equipped with a graph structure in a unique way, see, e.g., [3].

[6] But $f; g : A \overset{\text{pb}}{\to} C$ and $f : A \overset{\text{pb}}{\to} B$ do not, in general, imply $g : B \overset{\text{pb}}{\to} C$.

2. Egyed, A.: Fixing inconsistencies in UML design models. In: ICSE. pp. 292–301 (2007)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Tranformations. Springer, Heidelberg (2006)
4. König, H., Diskin, Z.: Advanced local checking of global consistency in heterogeneous multimodeling. In: Modelling Foundations and Applications - 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings, pp. 19–35 (2016). http://dx.doi.org/10.1007/978-3-319-42061-5_2
5. Lamo, Y., Wang, X., Mantz, F., Bech, Ø., Sandven, A., Rutle, A.: DPF workbench: a multi-level language workbench for MDE. In: Proceedings of the Estonian Academy of Sciences, vol. 62, pp. 3–15 (2013)
6. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: ICDE, pp. 117–128. IEEE Computer Society (2002)
7. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: ICSE, pp. 455–464 (2003)
8. Rutle, A., Rabbi, F., MacCaull, W., Lamo, Y.: A user-friendly tool for model checking healthcare workflows. In: (EUSPN-2013) and ICTH, pp. 317–326 (2013). http://dx.doi.org/10.1016/j.procs.2013.09.042
9. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: RE, pp. 221–230 (2007)