

# Advanced Local Checking of Global Consistency in Heterogeneous Multimodeling

Harald König<sup>1</sup>(✉) and Zinovy Diskin<sup>2,3</sup>

<sup>1</sup> University of Applied Sciences, FHDW Hannover, Hanover, Germany  
harald.koenig@fhdw.de

<sup>2</sup> NECSIS, McMaster University, Hamilton, Canada

<sup>3</sup> Generative Software Development Lab, University of Waterloo, Waterloo, Canada  
zdiskin@uwaterloo.ca

**Abstract.** Software design requires deployment of interdependent models conforming to different metamodels. This set of models is called a *multimodel*, and it must satisfy a set of *global* constraints regulating interaction of the multimodel components. A straightforward approach to global consistency checking would require merging component metamodels modulo their overlap, adding, perhaps, new global constraints to this merge, merging component models modulo their overlap, and checking the latter merge against the constraints in the former one. Being a natural *definition* for global consistency, these steps can not be used algorithmically because of two major practical drawbacks: they involve costly (meta)model matching to specify overlaps, and require building big and unfeasible merged metamodels and models.

The present paper makes two contributions. First, it presents a new algorithm to check each global constraint individually, and *as local as possible*, i.e., only using those (meta)model elements that affect the validity of the constraint. Second, it develops a mathematical foundation that allows us to formally prove that this individual local consistency checking is sound and complete w.r.t. the definition of global consistency.

## 1 Introduction

Modeling a complex system normally results in a *multimodel*, i.e., a set of heterogeneous models each one conforming to its own metamodel. A fundamental fact about multimodeling is that the merge of legal local models can result in a model violating global constraints declared in the integrated metamodel. This can be easily observed even for the simple homogeneous case, when all local models, and hence their merge, are instances of the same metamodel. For example, suppose that the metamodel of a domain says that persons in the domain are uniquely identified by their names, i.e., attribute ‘name’ is a key to class ‘Person’. Then the merge of two perfectly legal local instances can violate the

---

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

constraint, if there are different persons with the same name but they do not appear in the same instance.

*Heterogeneous* multimodeling expands the issue of global consistency enormously. For example, consider a metamodel  $M_1$  that extends the class *Person* above with attribute ‘birthdate’, and a metamodel  $M_2$  that extends ‘*Person*’ with reference ‘drives’ to class ‘*Car*’ owning attribute ‘carType’. Suppose that the domain is subject to the constraint that persons with age under 25 only drive sporty cars. This *global* constraint cannot be declared in either of the metamodels (the first one knows nothing about cars, the second one does not know ages of persons), yet checking its validity for a multimodel  $(A_1, A_2)$  with  $A_{1,2}$  being legal instances of  $M_{1,2}$  is important. A more complex example is consistency between a UML sequence diagram specifying collaborative behavior, and a statechart specifying a state machine protocol for that behavior. An obvious consistency requirement that traces specified by the sequence diagram should be allowed by the statechart is again global and cannot be declared in either of the local metamodels. Following [6], we call such requirements *inter-metamodel constraints*.

A straightforward approach to global consistency checking would require merging component metamodels  $M_i$  modulo their overlap (class ‘*Person*’ with attribute ‘name’ in the example above), adding, perhaps, new global constraints to this merge (‘young persons drive sporty cars’), merging component models  $A_i$  modulo their overlap, and checking the model merge  $A_+$  against the constraints over the metamodel merge  $M_+$ . In fact, this specification can be regarded as a *definition* of global consistency of a multimodel [6]. However, using this definition algorithmically as a specification of a workflow for global consistency checking would be impractical because of (a) costly (meta)model matching needed to specify the overlaps, and (b) necessity to build big and unfeasible merges of metamodels and models. A more efficient approach proposed in [2, 6] prescribes to do matching, merging and checking not for entire component models but for their projections to the respective metamodel overlaps, hence, the name *local* consistency checking. It was a conjecture (not proven formally) that the local approach is sound and complete w.r.t. (i.e., equivalent to) the above mentioned definition of global consistency.

The present paper makes two essential contributions to the local approach. The first is pushing the local checking idea even further up to its extreme: we propose 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  $C$ . Based on this technique, we can control the granularity of consistency checking by combining constraints into groups checked separately. (The two extremes are a multitude of groups having one global constraint each, and one big group embracing all global constraints.) Thus, while the original local approach of [6] reduces one huge global consistency check to a set of several lesser but still significant checks (with the correspondingly significant matches and merges), in this paper we propose an approach with a set of small checks (based on respectively easy matches and merges) in a size-controllable way. Correspondingly, we call the

former local approach to consistency *collective*, while the latter one *individual*. Besides reduced matching and merging workload, additional advantages of the local-individual approach are (a) better tailored and stepwise model repairing (in the *per constraint* fashion), and (b) possibilities to realize the *living with inconsistency* paradigm [9], when non-urgent consistency repairs (together with the respective matching and merging) can be postponed.

Our second contribution to local checking is an accurately defined mathematical framework that allows us to prove that individual consistency checking is sound and complete w.r.t. the definition of global consistency, and is equivalent to collective checking of [6]. Having specification (definition) and implementation (algorithms) separated is always useful as the former defines an optimization space for the latter. In addition, although conditions for our equivalence results are not too restrictive, they are not absolutely universal and (as we will show) can be violated if the global constraint to be checked badly interacts with inter-model correspondence specification involving queries against component models.

The paper is structured as follows: Constraint checking in general is contained in Sects. 2.1 and 2.2. Multimodels are introduced in Sect. 2.3. Section 3 combines these two topics: it explains how global constraint declarations are managed and states the main theorem, which precisely formulates the above mentioned equivalence. Section 4 is devoted to related work, Sect. 5 concludes.

## 2 Background

Metamodels are usually specified by UML class diagrams. The compact syntax of the latter hides many details that need to be explicated and formalized to allow our machinery to work. In this section, we show how it can be done in the formal framework of typed graphs (e.g., [8]) and diagrammatic constraints. The formalism of diagrammatic constraints, first developed under the cryptic name of *generalized sketches* [3, 5], and then promoted as the *Diagram Predicate Framework*, *DPF* [18, 19], is less known, and we present in Sect. 2.2 its basics in the amount needed for our work in the paper to make it self-contained. Finally, Sect. 2.3 introduces multimodels.

### 2.1 From Class Diagrams to Graphs, I: Typing

The left lower quadrant of Fig. 1 presents a fragment of a simplified metamodel for UML class diagrams with several constraints declared. Three multiplicity constraints are depicted in the usual UML style. They prescribe each operation to have a name and belong to at most one class, and prohibit multiple inheritance. A more complex OCL-constraint is specified in the top right corner of the metamodel box and says that if there is no superclass, there should be at least one interface implementation and vice versa (which shall guide the developers to code their programs in a polymorphic style). The left upper quadrant shows a class diagram (model) instantiating the metamodel. To use our machinery, we

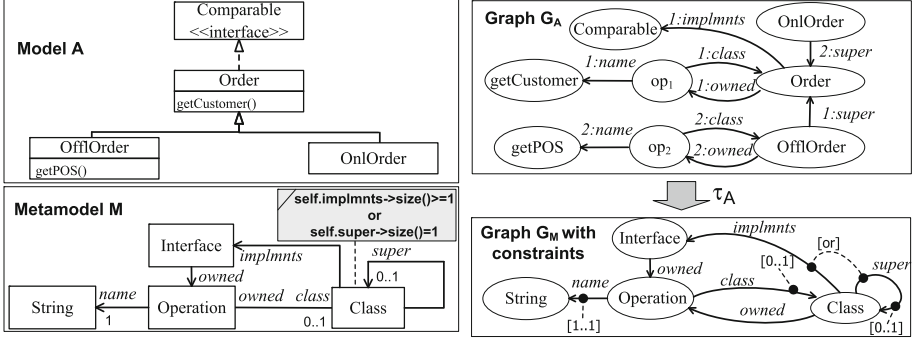


Fig. 1. UML model and metamodel represented as typed graph

need to translate the metamodel, the model, and the conformance relation into formal objects.

The right half of the figure shows the first step of the translation. The metamodel is presented by a pair  $M = (G_M, C_M)$  with  $G_M$  a *type* graph and  $C_M$  a set of four constraint declarations. Each of them consists of a constraint name given in square brackets, and the constraint *scope* shown by dashed lines, i.e., set of elements over which the constraint is declared. The model is a pair  $A = (G_A, \tau_A)$  with  $G_A$  a *data* graph, and  $\tau_A : G_A \rightarrow G_M$  a *typing* mapping between graphs, which assigns types to every data element, e.g.  $\tau_A(\text{Order}) = \text{Class}$ ,  $\tau_A(\text{op}_1) = \text{Operation}$ ,  $\tau_A(\text{getCustomer}) = \text{String}$ , as well as  $\tau_A(1:\text{implmnts}) = \text{implmnts}$ ,  $\tau_A(1:\text{super}) = \text{super}$  and so on. Model  $A$  is a *typed graph* and we will also say that  $A$  is *typed over*  $M$ , and often write  $a:T$  (read “element  $a$  is of type  $T$ ”) if  $\tau(a) = T$ . A standard formalization of the notion of graphs and mappings between them is briefly described below. Constraints and conformance of a model to constraints is specified in Sect. 2.2.

A (*directed multi*-)graph  $G = (V_G, E_G, s, t)$  consists of a set  $V$  of *vertices* (or *nodes*), a set  $E$  of *edges*, and two functions  $s : E \rightarrow V, t : E \rightarrow V$  that assign to each edge its source and target. Writing  $x \in G$  means that  $x$  is a node or an edge of  $G$ . We depict graph vertices by ellipses (or circles) and edges by arrows from their source to their target vertex, cf. Fig. 1, graphs  $G_A$  and  $G_M$ . A graph *mapping* or *morphism*  $f : G \rightarrow G'$  is a pair of functions  $f_V : V \rightarrow V'$  and  $f_E : E \rightarrow E'$  preserving the incidence between vertices and edges. Since the definition of  $f$  on an edge  $e$  determines its values for  $e$ ’s source and target, we will often omit the latter from the mapping definition.

## 2.2 From Class Diagrams to Graphs, II: 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 the *or*-constraint is two arrows with a common source, see Table 1.

To declare a constraint over a metamodel graph  $G_M$ , we recognize the constraint shape in the graph and visualize it as was shown in Fig. 1. Formally, this recognition is a graph mapping  $\delta : S^c \rightarrow G_M$  (called (*shape*) *binding*) from the shape  $S^c$  of a constraint with name  $c$  to graph  $G_M$ . E.g. in Fig. 1, we have constraint  $[or]$  declared by binding  $\delta : S^{[or]} \rightarrow G_M$  ( $S^{[or]}$  is shown in Table 1) with  $\delta(01) = implmnts$ ,  $\delta(02) = super$ , i.e.  $\delta(1) = Interface$ ,  $\delta(0) = Class = \delta(2)$ . The set of elements in  $G_M$  the shape is mapped to, is called the *image* of the binding. In the example, the image of  $\delta$  consists of vertices *Interface* and *Class*, and edges *implmnts* and *super*.

The pair  $(c, \delta)$  is a *constraint declaration*. The bindings of all relevant constraints of graph  $G_M$  are shown in detail in Fig. 2. Note the practicality of the DPF framework: for the  $[0..1]$ -declarations in  $G_M$  we can *reuse* shape  $S^{[0..1]}$  in two different bindings: one of them maps edge 12 to edge *super*, the other maps 12 to edge *class*. Thus, validation logic is encapsulated and can be reused for all constraint declarations of type  $[0..1]$ . In the sequel, we write a pair  $(c, \delta)$  as  $c@ \delta$ , meaning *constraint  $c$  is imposed on metamodel  $G_M$  at the image of binding  $\delta$* .

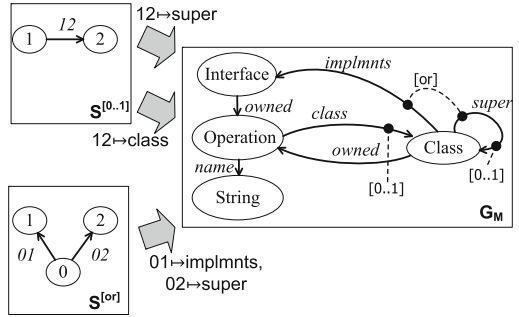
In order to check consistency of model  $A$ , i.e. typed graph  $A = (G_A, \tau_A)$ , against a fixed constraint declaration  $c@ \delta$ , we need to define  $c$ 's *semantics* irrespective of  $A$ . This is done by programming a function  $VALIDATE_c(B: Model): BOOLEAN$  which has input typed graph  $B = (G_B, \tau_B)$  where  $\tau_B : G_B \rightarrow S^c$ , i.e.  $B$  is a model typed over  $c$ 's shape only. For example, function  $VALIDATE_{[or]}$  acts on models typed over  $S^{[or]}$  (cf. Table 1): it returns *true* for a model  $X = (G_X, \tau_X : G_X \rightarrow S^{[or]})$ , iff each element of type 0 in  $G_X$  has an outgoing edge to some element of type 1 or to some element of type 2.

So defined semantics is used in the check function:

$CHECK(A: Model, c@ \delta: Constraint): BOOLEAN$

**Table 1.** Sample constraints

Name	Shape
$[0..1]$	
$[or]$	

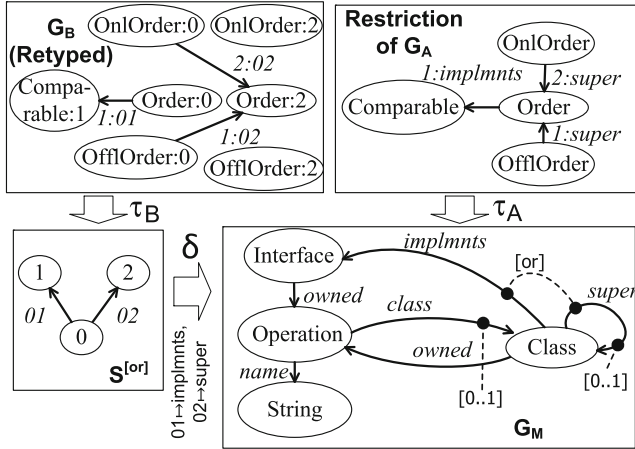


**Fig. 2.** Three constraint declarations

which, basically, performs three steps:

1. *Restrict*  $A$  to elements, whose types are in the image of  $\delta$  in  $G_M$ .
2. *Retype* elements of this new structure to formal typing over  $S^c$ . This yields typed graph  $B = (G_B, \tau_B)$ .
3. Return the result of  $\text{validate}_c(B)$ .

We say that  $A$  *satisfies*  $c@ \delta$  and write  $A \models c@ \delta$ , if  $\text{CHECK}(A, c@ \delta) = \text{true}$ . Model  $A$  is a *legal* model over metamodel  $M$ , if it satisfies all constraints declared in  $M$ . For example, checking constraint declaration  $[or]@ \delta$  is shown in Fig. 3. The image of  $\delta$  is shown in the lower right part (elements not in the image are greyed out), the restriction  $G_A$  is in the top right quadrant, and  $B = (G_B, \tau_B : G_B \rightarrow S^{[or]})$  is the corresponding retyping. As  $\text{validate}_{[or]}(B) = \text{true}$ , we conclude  $A \models [or]@ \delta$ . Note the copy procedure during retyping: for each *class*-instance in the restriction of  $G_A$ , we have to create *two* vertices in  $G_B$ , because we must incorporate their two possible roles as subclass (source of edge *super*) and superclass (target of edge *super*). This is a general procedure: each vertex or edge in  $G_A$  has to be represented  $n$  times in  $G_B$ , if its type in  $G_M$  has  $n$  preimages under  $\delta$ . In this way, we can consider elements in all possible occupied roles. This “role-based” retyping procedure is actually carried out via the general mathematical *pullback* construction [1, 13].



**Fig. 3.** How function CHECK works

### 2.3 Multimodeling

Modeling a complex system normally results in a *multimodel*, i.e., a set of heterogeneous models each one conforming to its own metamodel. Besides class diagrams, other types of UML diagrams are produced, for instance sequence diagrams, statecharts, activity diagrams, etc. Even class diagrams may conform

to different metamodels: Business analysts may use behavioural specifications only [10] with no attributes or associations,  $M_1$  in Fig. 4, whereas for another modeling team, class models are more technically oriented and associations and attributes are used ( $M_2$ ). In all cases, the models collectively represent a single system to be build, and any formal treatment has to consider *overlaps*, i.e. the definitions of common terminology in different models. E.g., the (meta) concept *class* occurs in both of the above-mentioned metamodels. Names of common concepts, however, may differ: one team may use the term *String*, while the other may use *Text*, yet speaking of the same concept.

In the binary case (two metamodels  $M_1$  and  $M_2$ ), *overlaps* can be specified by two graph mappings  $M_1 \xleftarrow{r_1} M_{12} \xrightarrow{r_2} M_2$  in which  $M_{12}$  contains all common concepts. Any pair  $x_1 \in M_1$  and  $x_2 \in M_2$  is declared to be the same, if there is  $x \in M_{12}$  such that  $r_1(x) = x_1$  and  $r_2(x) = x_2$ . We call this configuration of metamodels and mappings a *multimetamodel*  $\mathcal{M}$  and write  $\mathcal{M} = (M_1, M_2, M_{12}, r_1, r_2)$  or shorter  $\mathcal{M} = (r_1, r_2)$ , if domain and codomain of  $r_1$  and  $r_2$  are clear from the context.

In the sequel, all (meta)models will be (typed) graphs, such that we simplify notation by using letters  $M$  (metamodels) and  $A$  (models) with subscripts to distinguish different graphs. A multimetamodel  $\mathcal{M}$  is shown in Fig. 4:  $M_1$  was already used in Sect. 2.1, Fig. 1.  $M_2$  is the above mentioned technical metamodel. The overlap specification  $M_{12}$  declares *Class* together with its super-relation to be the same and, since  $r_1(\text{Str\_Txt}) = \text{String}$  and  $r_2(\text{Str\_Txt}) = \text{Text}$ , it declares sameness of *String* and *Text* (see the shaded vertices). The *merge* (union)  $M_+$  of the two components of  $\mathcal{M}$  is shown in the lower half of Fig. 4. We introduce merges in Sect. 3.

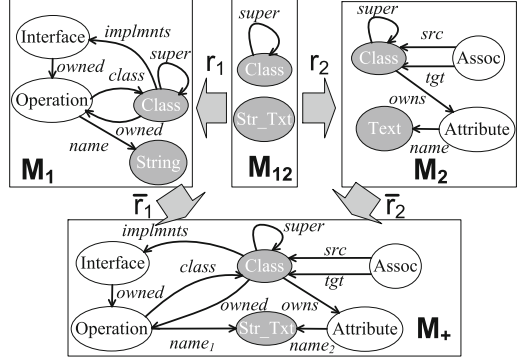


Fig. 4. Multimodel and merge

### 3 Managing Global Constraints

In the present section we analyse *global* constraints, i.e., constraints that reside in neither of the component metamodels alone, and thus involve elements from several metamodels. Correspondingly, we use the name *inter-metamodel constraints* that accurately describes the case. In Sect. 3.1, we will state a *definition* of *global satisfaction* against an inter-metamodel constraint. The definition treats the binary case only, but the generalization for the N-ary case is straightforward. Models typed over different metamodels are said to be *globally* consistent if they satisfy all imposed inter-metamodel constraints. We will argue that it is

impractical to use this definition as an algorithm for global consistency checking. Hence, in Sect. 3.2, we introduce another algorithm, in which global satisfiability against an inter-metamodel constraint is checked *locally*, and illustrate its advantages with a running example. Section 3.3 compares the global satisfaction definition of Sect. 3.1 with the local algorithm of Sect. 3.2 and, additionally, with the collective method of [6]. Finally, equivalence of all three methods is stated (main theorem).

### 3.1 Global Consistency

Global inter-metamodel constraints are spread over different components of a multimetamodel. Consider e.g. the binary multimetamodel in Fig. 4 with the following constraint declaration  $C$  (a standard requirement for Java Beans):

*For each attribute named “n” there must be an accessor operation with name “getN”!*

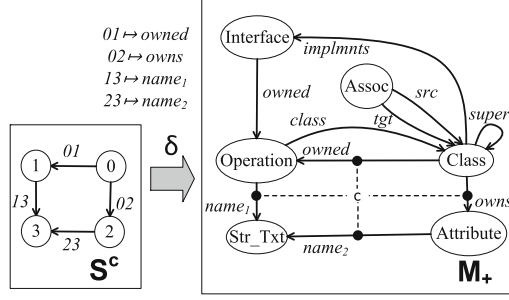
to be checked for models  $A_1$  over metamodel  $M_1$  and  $A_2$  over metamodel  $M_2$ .

In the diagrammatic constraint framework, to declare  $C$ , we need to find a corresponding constraint  $c$  and binding mapping  $\delta: S^c \rightarrow M$ . For this, we take for  $M$  the merge  $M_+$  of  $M_1$  and  $M_2$  w.r.t. overlap  $M_{12}$ . This is shown in the lower half of Fig. 4. Basically, it is the union of  $M_1$  and  $M_2$  modulo  $M_{12}$ : Since *Class* is common to both components, it appears only once in the merge. The same is true for *String* and *Text* being represented by *Str.Text* in  $M_+$ . However, the two edges labelled *name* in  $M_1$  and  $M_2$  are not unified: They are not declared the same in the overlap (one is an operation’s name, the other the name of an attribute).  $\bar{r}_1: M_1 \rightarrow M_+$  and  $\bar{r}_2: M_2 \rightarrow M_+$  map all elements of  $M_1$  and  $M_2$  to the corresponding elements in the merge. Now we can impose  $c$  to  $M_+$  via binding map  $\delta$ . This is shown in Fig. 5.

$c$ ’s intended semantics is controlled by function  $\text{VALIDATE}_c$  (cf. Sect. 2.2), which has input graph  $B = (G_B, \tau_B)$  typed over  $S^c$ . If  $S^c$  is bound as shown in Fig. 5, it will return true if and only if for each own class attribute with name  $n$ , there is an owned operation with name *getN* in the same class. Note that the *super* relation is not included in the image of  $\delta$ , because getters shall exist for *own* attributes only (inherited attributes already yield respective *get*-methods).

In Sect. 2.3, we described two modeling teams. Assume the first team creates legal model (one or more class diagrams)  $A_1$  typed over metamodel  $M_1$ , and the other team creates legal model  $A_2$  typed over metamodel  $M_2$ . Global consistency requires validity of the name alignment constraint  $c@_\delta$  introduced above. Conjoint treatment of models requires their *matching*, i.e., specifying their common concepts. But *model overlap* might not be possible to be inferred automatically: e.g., entity *Onl(ine)Order* in model  $A_1$  may be called *Onl(ine)Purchase Order* in  $A_2$ , cf. Fig. 6. In general, cross-(meta)model terminology may be very heterogeneous, and the structure of models may vary significantly while still reflecting identical concepts. Given a significant size of practical models, model matching can be a costly procedure that needs special tools and user input.





**Fig. 5.** Imposing global constraint on merged multimodel

Formally – and similarly to metamodels – one must determine two graph mappings  $A_1 \xleftarrow{r'_1} A_{12} \xrightarrow{r'_2} A_2$  that are compatible with typing<sup>1</sup>. We call this configuration of models and mappings a *multimodel*  $\mathcal{A}$  over multimetamodel  $\mathcal{M}$  and write  $\mathcal{A} = (r'_1, r'_2)$ <sup>2</sup>. Only now is it possible to merge multimodel  $\mathcal{A}$ , which, basically, is performed in the same way as for metamodels: One constructs the union  $G_{A_+}$  of the data graphs of  $A_1$  and  $A_2$  wrt. to  $A_{12}$ . This yields a unique typing mapping  $\tau_{A_+} : G_{A_+} \rightarrow M_+$  (this can formally be proved, because merging is a special case of the universal construction of pushouts [1]) and hence model merge  $A_+ = (G_+, \tau_{A_+})$ .

**Definition 1 (Global Consistency [6,20]).** Let  $c@ \delta$  be an inter-metamodel constraint over multimetamodel  $\mathcal{M} = (r_1, r_2)$ . We say that multimodel  $\mathcal{A} = (r'_1, r'_2)$  over  $\mathcal{M}$  satisfies  $c@ \delta$ , if the above constructed model merge  $A_+$  satisfies  $c@ \delta$  over  $M_+$ . If  $\mathcal{A}$  satisfies all inter-metamodel constraints imposed on  $\mathcal{M}$ , we call  $\mathcal{A}$  *globally consistent*.

We remark that the binary case can be generalized to the N-ary case by constructing  $M_+$  as *colimit*, a categorical construction encompassing binary merging [6,20].

Unfortunately, practical consistency checking along the lines of this definition, i.e., constructing *globally* typed data before checking, has major disadvantages:

1. One has to deal with the entire union of data (usually a huge structure) - independent of whether there is only a small portion being affected by the constraint.
2. To specify overlaps of typed data structures, this enormous collection of data has to be traversed manually or at least semi-automatically. Overlaps have to be complete, i.e. they are not specific to the given constraint declaration.

<sup>1</sup> Formally,  $r'_i$  ( $i \in \{1, 2\}$ ) map the data graph of  $A_{12}$  and respect behavior of  $r_i$ , i.e.  $r'_i; \tau_i = \tau_{12}; r_i$ .

<sup>2</sup> Again assuming domain and codomain of  $r'_1$  and  $r'_2$  to be clear from the context.

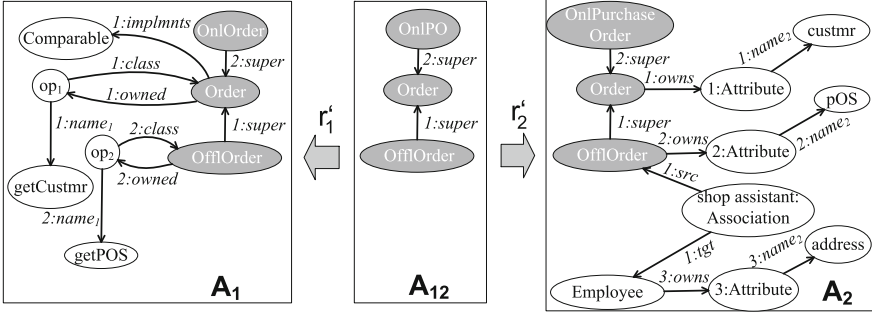


Fig. 6. Multimodel: models with overlap

Consider e.g. Fig. 6:  $A_1$  contains owned operations and implemented interfaces of the order classes.  $A_2$  represents the same order classes. Shaded nodes and their *:super*-links are in the overlap  $A_{12}$ , i.e. *OnlOrder* and *OnlPurchaseOrder* are declared to be the same classes despite their different names. Besides own attributes,  $A_2$  contains the *shop assistant* who processed the offline order (via a directed association).  $A_+$  is the union of all these elements w.r.t. the overlap. It is not reprinted due to lack of space.

If we want to check whether  $\mathcal{A} = (r_1', r_2')$  satisfies constraint  $c@ \delta$ , the above mentioned disadvantages manifest as follows:

1. Although  $c@ \delta$  only “talks” about classes and names of their attributes and operations, we have to deal with interface implementations and operation’s reference to its class (from  $A_1$ ), as well as (usually many) associations (from  $A_2$ ) but also with superclass relations (in the overlap).
2. The user must search the set of all classes and all their superclass relations for identical concepts. In the example he must specify sameness of *OnlOrder* and *OnlPurchaseOrder*, the other two identities *Order* and *OfflOrder* may automatically be proposed based on identical naming, yet have to be confirmed by the user. The user also has to declare several superclass relations to be the same although the constraint declaration does not talk about inheritance relation.

Both aspects become more severe, if there is a big number of diagrams, probably stored with different techniques. Moreover, our examples are small compared with real diagrams, where the proportion of matching (i.e. overlap specification) of non-relevant data (being outside the fragment that matters for checking) will be significantly bigger.

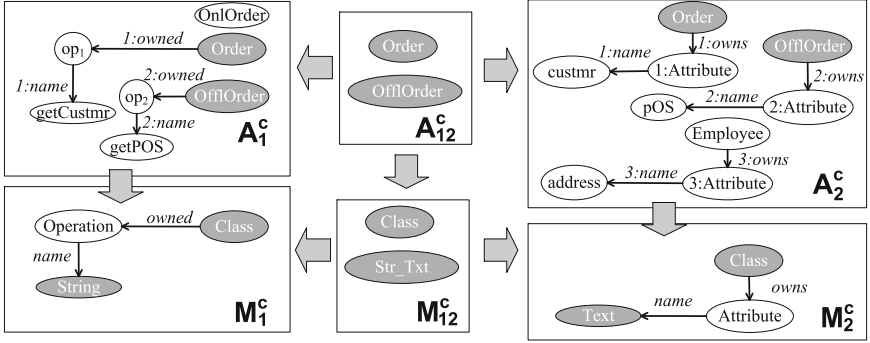
### 3.2 Local-Individual Checking

Is there a technique for checking inter-metamodel constraints that would be more efficient than a direct execution of the definition (Definition 1 in Sect. 3.1) as proposed in [20]? A better approach would be to consider only those pieces of

data and models and their overlaps that matter for checking, i.e., make checking constraints *as local as possible*:

**Definition 2.** The following algorithm for global consistency checking is called *local-individual* checking. Let  $\mathcal{A} = (A_1, A_2, A_{12}, r'_1, r'_2)$  be a multimodel over multimetamodel  $\mathcal{M} = (M_1, M_2, M_{12}, r_1, r_2)$ . Let  $\bar{r}_1$  and  $\bar{r}_2$  be inclusion maps of  $M_1$  and  $M_2$  into the metamodel merge  $M_+$  as is Fig. 4. An inter-metamodel constraint  $c@δ$  is verified as follows (the following four steps will be illustrated afterwards by way of example):

1. Let  $M_1^c$ ,  $M_2^c$ , and  $M_{12}^c$  consist of all elements of  $M_1$ ,  $M_2$ , and  $M_{12}$ , resp., which are mapped to the image of  $δ$  by  $\bar{r}_1$ ,  $\bar{r}_2$ , and  $r_1; \bar{r}_1 (= r_2; \bar{r}_2)$ , resp.<sup>3</sup>
2. Restrict models  $A_1$  and  $A_2$  to those elements being typed over  $M_1^c$  and  $M_2^c$  resp. Call this data  $A_1^c$  and  $A_2^c$ .
3. Determine overlap  $A_{12}^c$  of  $A_1^c$  and  $A_2^c$ .
4. Apply  $check(A_+^c, c@δ)$ , where  $A_+^c$  is the local merge of  $A_1^c$  and  $A_2^c$ .



**Fig. 7.** Individually local consistency checking: steps 1 to 3

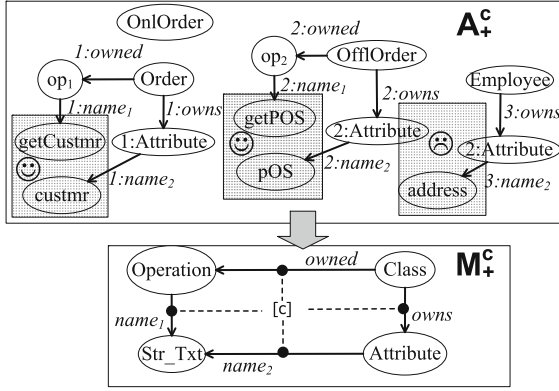
We illustrate application of the algorithm for our running example, where multimodel  $(r'_1, r'_2)$  from Fig. 6 will be checked against constraint declaration  $c@δ$  from Fig. 5. Steps 1 to 3 are illustrated in Fig. 7:

*Step 1* :  $M_1^c$ ,  $M_2^c$ , and  $M_{12}^c$  are depicted in the lower half. Once the complete overlap  $M_{12}$  is known, they are automatically derived from the scope of the constraint. Shaded vertices again depict overlap. The important improvement is that *Interfaces* together with their operations and interface implementations now vanish. In the same way, class membership of operations can be omitted. Since the constraint declaration does not involve superclass relations, they can be omitted, too. Moreover, we need not care about associations and their source and targets.

<sup>3</sup> We still consider  $c@δ$  to be imposed on the merge  $M_+$  of  $M_1$  and  $M_2$ , i.e.  $δ : S^c \rightarrow M_+$ . Recall that the *image* of  $δ$  is the set of those elements in  $M_+$ , the shape of  $c$  is mapped to.

*Step 2* : The upper half shows appropriately narrowed  $A_1^c$  and  $A_2^c$ . Again, this step can be carried out automatically (similar to the retype step of function CHECK as described in Sect. 2.2). Note that *OnlPurchaseOrder* is omitted since it does not possess own attributes and hence automatically satisfies constraint declaration  $c@δ$ .

*Step 3* : The only manual activity is overlap specification. It is now reduced to the selection of classes *Order* and *OfflOrder*. We do not have to deal with superclass relations and classes without attributes in the overlap. Additionally, no text matching is necessary, since model structures simplify accordingly. Moreover, declaration of *OnlOrder-OfflOrder-identity* is no longer necessary.



**Fig. 8.** Local consistency checking: step 4

*Step 4* : Calculation of local merge  $A_+^c$  is again an automatic procedure. In Fig. 8, it is depicted together with the part  $M_+^c$  of the integrated metamodel that matters for checking. The resulting data space ( $A_+^c$ ) now contains no superfluous elements. It is reduced to four involved classes only: *OnlOrder* still appears (but now only as automatic leftover from  $A_1$ ). The other three classes can easily be traversed. Function CHECK has input a narrowed model (only those model elements typed over elements in the image of the binding). In the example it detects satisfaction for classes *Order* and *OfflOrder* but violation for class *Employee* (grey rectangles).

The reader may compare the unstructured contents of Fig. 6 with the reduced data in the upper half of Fig. 7. The presented technique obviously reduces model merging and matching workload, if constraints shall be checked in the *per constraint* fashion. It can also be applied, if it is temporarily possible to live with inconsistency, i.e. with delayed non-urgent consistency repairs [9].

It remains to ensure *global-local-equivalence*, i.e. the algorithm must always yield the same result as the global definition (cf. Definition 1). This equivalence may seem obvious, but it can be invalid for model structures richer than simple

typed graphs. For instance, assume that (meta)models can be augmented with *derived associations*: if class  $C_1$  has an association  $a$  to  $C_2$  and  $C_2$  has association  $b$  to  $C_3$ , then there is a derived association  $/ab$  from  $C_1$  to  $C_3$  (note dashed arrow in  $M_+$  in Fig. 9). In Fig. 9, model  $M_+$  is the merge of  $M_1$  and  $M_2$ , in which class  $C_2$  is assumed to be common in both metamodels.

Now consider a constraint declaration “Each object instantiating  $C_1$  must reference at least one  $C_3$ -object via an  $/ab$ -link.” Its binding map  $\delta$  has image consisting of the derived association and classes  $C_1$  and  $C_3$ . Then the global check procedure of Definition 1 for multimodel  $\mathcal{A}$  with models  $(A_1, A_2)$  that share the object  $:C_2$  (in Fig. 9, object identifiers are omitted) will construct the whole model merge with a derived  $/ab$ -link from the  $C_1$ -object to the  $C_3$ -object. Hence,  $\mathcal{A}$  is consistent w.r.t. Definition 1.

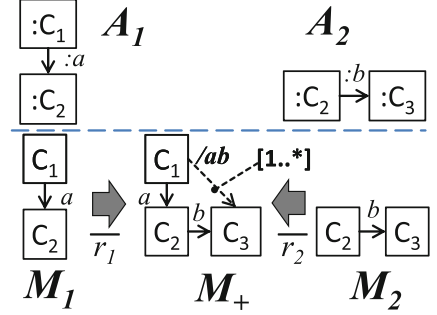


Fig. 9. Derived association

However, the local algorithm (Definition 2) hides class  $C_2$  in step 1, because  $C_2$  is not in the image of the constraint declaration. Hence, the restricted model  $A_1^c$  only contains object  $:C_1$ , and  $A_2^c$  only contains  $:C_3$  (step 2), and we then necessarily have an empty overlap (step 3). The local merge (step 4) does not contain any association, and hence no derived one. Thus, in contrast to the global check, the local check returns *false*!

This mismatch shows that investigation of global-local equivalence must be carried out carefully, specifically, when dealing with correspondences involving derived elements [6]. However, in the next section we show that the equivalence always holds in the framework of typed graphs without derived elements.

### 3.3 Global-Local-Equivalence

In order to ensure that all proposed algorithms are correct in the context of typed graphs, we have to prove (a) that the new local-individual approach is correct w.r.t. the original definition of global consistency in Sect. 3.1, and (b) that local-individual checking is equivalent to the local-collective grouping technique proposed in [6], where special portions of the metamodel are determined such that constraint groups imposed on this portion can be checked simultaneously.

We include a short proof sketch of the global-local-equivalence theorem, which is stated in the end of this section (a detailed proof is given in [13]): both, the above definition and the invented algorithm contain a merging step: one for the entire metamodel (which yields  $M_+$ ) and one for only those parts that matter for the constraint (which yields  $M_+^c$ ). In the proof of our theorem we compare both approaches by - virtually - carrying them out in parallel: We use the fact that these *simultaneous merges* can be controlled with the so-called *Van*

*Kampen Property* [8], whenever one of the two graph mappings  $r_1$  and  $r_2$  is injective<sup>4</sup>. It is an exactness property for typed graphs, which guarantees that both operations, merge and restriction, behave and interact well. It fails in categories where augmentation effects as described above occur. Basically, one can show that the well-behavedness of the simultaneous metamodel merge ( $M_+$  and  $M_+^c$ ) carries over to the model level ( $A_+$  and  $A_+^c$ ). Then it is not difficult to deduce that  $\text{check}(A_+, c@{\delta}) = \text{validate}_c(\text{retype}(A_+^c))$ , where  $\text{retype}$  performs Step 2 of function CHECK in Sect. 2.2. Global-local-equivalence then follows, since, by construction,  $A_+^c$  is the restriction of  $A_+$  (Step 1 of function CHECK).

It is also important to compare the approach with the local-collective method of [6], in which checking the global consistency of multimodel  $\mathcal{A}$  against a *group* of constraints  $C = \{c_1@{\delta_1}, \dots, c_n@{\delta_n}\}$ , which is locally satisfied by component models, is reduced to checking consistency against  $C$  at the model overlap. It is not difficult to show that this setting can be seen as a special case of our framework, in which a global constraint declaration  $c@{\delta}$  encodes the entire group  $C$ : constraint  $c$  is a logical conjunction of constraints  $c_i$  in some precisely defined sense, and the image of  $\delta$  is the union of images of  $\delta_1, \dots, \delta_n$ . Then collective checking of group  $C$  is equivalent to individual checking of constraint  $c@{\delta}$ .

From all these considerations we deduce our *main theorem*:

**Main Theorem.** Let metamodels be graphs, models be typed graphs, and model mappings are typed graph morphisms. Let  $\mathcal{M} = ( M_1 \xleftarrow{r_1} M_{12} \xrightarrow{r_2} M_2 )$  be

a multimetamodel with  $r_1$  or  $r_2$  injective,  $\mathcal{A} = ( A_1 \xleftarrow{r'_1} A_{12} \xrightarrow{r'_2} A_2 )$  be a multimodel over  $\mathcal{M}$ , and  $c@{\delta}$  be an inter-metamodel constraint declaration over the merged metamodel  $M_+$ . Then the following statements are equivalent:

- $\mathcal{A}$  satisfies  $c@{\delta}$  according to Definition 1.
- The local-individual algorithm of Definition 2 returns true for  $c@{\delta}$ .
- If  $c@{\delta}$  encodes a group  $C$  of constraint declarations, then  $\mathcal{A}$  satisfies  $C$  according to the local-collective approach of [6]. □

## 4 Related Work

Approaches to heterogeneous multimodeling can be roughly divided into *global* and *local*. For the former, heterogeneity is managed by relating all local models to one global model, and checking consistency wrt. this global model. In contrast, there is no global model in local approaches.

The most direct (and most well-known) global approach to consistency checking is via monitoring satisfiability of consistency rules. All local models are considered as instances of some all-embracing global model given a priori, and inter-model consistency is given by *rules* specified in a special language “understanding” all local models. A representative of this approach is described in [15]:

<sup>4</sup> All examples in the present paper are such that *both*  $r_1$  and  $r_2$  are injective.

inter-metamodel constraints are called inter- or multi-feature rules, and are investigated in the context of feature-oriented software development. Inconsistency detection, for instance, is performed by mapping feature models to propositional logic. Different to our approach, matching is only allowed when elements have same types *and* same names. Hence, matching can well be automated. A mini-survey of similar approaches can be found in [6].

Another global approach is *consistency checking via merging (CCVM)* proposed in [20] for homogeneous structural modeling, and earlier discussed in [7] for behavioral modeling; in [6], it was generalized for the heterogeneous case. The global model is not given a priori but is computed by merging all local models modulo their correspondences; the latter must be explicitly specified. An essential advantage of CCVM approaches over monitoring consistency rules is that complex types of model matching are allowed. Contributions of the present paper into CCVM were discussed in the introduction in detail.

For *local approaches*, explicit specification of inter-model correspondences is a central issue, and different types of notation and techniques were developed [17]. Besides the usual distinction between manual and (semi-)automatic procedures, e.g. [23], more sophisticated approaches have been elaborated [12]. A distinctive feature of our approach is that the set of correspondences is reified as a special model endowed with correspondence mappings – a span. This is a standard categorical idea, which was repeatedly employed in homogeneous multimodeling frameworks based on category theory, the most prominent being [21], where spans are themselves subject of evolution. The most difficult issue is *indirect correspondences*, when sets of elements in different models are related but their relationships cannot be specified by equating the elements. Such correspondences are usually specified by *correspondence rules* [17], but their formal treatment needs the machinery of Kleisli mappings [4]; incorporating the latter into the framework developed in this paper is our important future work.

Finally, the Van Kampen property (originally invented in algebraic topology) reveals a remarkable correspondence between software engineering and a mathematical method for inferring properties of a global structure from its known local characteristics, cf. [8, 22]. Since our work is also about interconnection of the local and the global, it is not surprising that the Van Kampen property is fundamental for our framework as well.

## 5 Conclusion

We presented a new approach for local checking of constraints imposed on heterogeneous multimodels, which significantly reduces model matching and merging workload. Our second contribution is a formal underpinning of global consistency, which essentially employs the diagrammatic nature of constraint. In this framework, we were able to prove the equivalence of two local approaches to the global consistency definition, so that the latter provides an optimization space for the former.

The most important direction for future research is to generalize the proposed binary overlapping algorithm together with a necessary equivalence theorem for the general N-ary overlapping case considered in [6]. Moreover, view definitions (on metamodels) and view execution (on models) [6] should also be taken into consideration. The challenge will be to find appropriate generalization and extensions of our mathematical machinery for model correspondences involving derived elements. Another direction of future research is to extend the scope of underlying graphical structures beyond simple directed (typed) graphs and include, e.g., attributed graphs [8]. Obviously, this also requires a generalization of the underlying diagrammatic framework.

We also plan to evaluate the algorithm in the tooling framework developed at Bergen University College [14,16]. Our idea is to enhance DPF editors to make them inter-metamodel aware. Alternatively, we can try to integrate our approach with another constraint checking tools, e.g. USE, a tool to specify and check OCL constraints [11].

**Acknowledgement.** We are sincerely grateful to anonymous reviewers for useful comments and suggestions. Financial support was provided by Automotive Partnership Canada via the Network on Engineering Complex Software Intensive Systems (NECSIS).

## References

1. Barr, M., Wells, C.: *Category Theory for Computing Sciences*. Prentice Hall, New York (1990)
2. Diskin, Z.: Towards generic formal semantics for consistency of heterogeneous multimodels. Tech. Rep. GSDLAB 2011-02-01, University of Waterloo (2011)
3. Diskin, Z., Kadish, B., Piessens, F., Johnson, M.: Universal arrow foundations for visual modeling. In: Anderson, M., Cheng, P., Haarslev, V. (eds.) *Diagrams 2000*. LNCS (LNAI), vol. 1889, pp. 345–360. Springer, Heidelberg (2000)
4. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: de Lara, J., Zisman, A. (eds.) *Fundamental Approaches to Software Engineering*. LNCS, vol. 7212, pp. 163–177. Springer, Heidelberg (2012)
5. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. In: *Proceedings of the Second Workshop on Applied and Computational Category Theory (ACCAT 2007)*, pp. 19–41. ENTCS (2007)
6. 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)
7. Easterbrook, S.M., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: *ICSE*, pp. 411–420 (2001)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformations*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
9. Fickas, S., Feather, M., Kramer, J.: *Proceedings of ICSE 1997 Workshop on Living with Inconsistency*, Boston, USA (1997)
10. Fowler, M.: *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading (1997)



11. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* **69**(1–3), 27–34 (2007). <http://dx.doi.org/10.1016/j.scico.2007.01.013>
12. Kessentini, M., Ouni, A., Langer, P., Wimmer, M., Bechikh, S.: Search-based meta-model matching with structural and syntactic measures. *J. Syst. Softw.* **97**, 1–14 (2014). <http://dx.doi.org/10.1016/j.jss.2014.06.040>
13. König, H., Diskin, Z.: Individually local checking of global consistency in heterogeneous multimodeling: the categorical story behind the scenery. Tech.rep., University of Applied Sciences, FHDW Hannover (2016). <http://fhdwdev.ha.bib.de/public/papers/02016-01.pdf>
14. Lamo, Y., Wang, X., Mantz, F., Bech, Ø., Sandven, A., Rutle, A.: DPF workbench: a multi-level language workbench for MDE. *Proc. Est. Acad. Sci.* **62**, 3–15 (2013)
15. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) *ECMFA 2010*. LNCS, vol. 6138, pp. 217–232. Springer, Heidelberg (2010)
16. Rabbi, F., Lamo, Y., Yu, I., Kristensen, L.: A diagrammatic approach to model completion. In: 4th Workshop on Analysis of Model Transformations Co-Located with MODELS 2015, pp. 56–65 (2015)
17. Romero, J., Jaen, J., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: *EDOC*, pp. 163–172. IEEE Computer Society (2009)
18. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A diagrammatic formalisation of MOF-based modelling languages. In: Oriol, M., Meyer, B. (eds.) *Objects, Components, Models and Patterns*. LNBIP, vol. 33, pp. 37–56. Springer, Heidelberg (2009). [http://dx.doi.org/10.1007/978-3-642-02571-6\\_4](http://dx.doi.org/10.1007/978-3-642-02571-6_4)
19. Rutle, A., Wolter, U., Lamo, Y.: A diagrammatic approach to model transformations. In: *Proceedings of the 2008 Euro American Conference on Telematics and Information Systems (EATIS 2008)*, pp. 1–8. ACM (2008)
20. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: *RE*, pp. 221–230. IEEE (2007)
21. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) *WG 1994*. LNCS, vol. 903. Springer, Heidelberg (1995)
22. Sobocziński, P.: Deriving process congruences from reaction rules. Tech. Rep. DS-04-6, BRICS Dissertation Series (2004)
23. de Sousa Jr., J., Lopes, D., Claro, D.B., Abdelouahab, Z.: A step forward in semi-automatic metamodel matching: algorithms and tool. In: Filipe, J., Cordeiro, J. (eds.) *Enterprise Information Systems*. LNBIP, vol. 24, pp. 137–148. Springer, Heidelberg (2009). [http://dx.doi.org/10.1007/978-3-642-01347-8\\_12](http://dx.doi.org/10.1007/978-3-642-01347-8_12)

<http://www.springer.com/978-3-319-42060-8>

Modelling Foundations and Applications

12th European Conference, ECMFA 2016, Held as Part  
of STAF 2016, Vienna, Austria, July 6-7, 2016,

Proceedings

Wąsowski, A.; Lönn, H. (Eds.)

2016, XVIII, 265 p. 123 illus., Softcover

ISBN: 978-3-319-42060-8