Specifying Overlaps of Heterogeneous Models for Global Consistency Checking

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki

Generative Software Development Lab.,
University of Waterloo, Canada
{zdiskin, y6xiong, kczarnec}@gsd.uwaterloo.ca

Abstract. Software development often involves a set of models defined in different metamodels, each model capturing a specific view of the system. We call this set a *multimodel*, and its elements *partial* or *local* models. Since partial models overlap, they may be consistent or inconsistent wrt. a set of *global* constraints.
We present a framework for specifying overlaps between partial models and defining their global consistency. An advantage of the framework is that heterogeneous consistency checking is reduced to the homogeneous case yet merging partial metamodels into one global metamodel is not needed. We illustrate the framework with examples and sketch its formal semantics based on category theory.

1 Introduction

Software development often involves a set of heterogeneous models, such as use cases, process models, UML design models, and code. These models are defined by different metamodels, and are often built by different teams, but collectively represent a single system. Due to possible overlaps between models, individually consistent models may be *globally* inconsistent if taken together. Many existing approaches focus on checking consistency of a single model or a pair of models [1]. However, individual consistency or pairwise consistency do not guarantee global consistency. For example, Fig. 1 shows three UML class diagrams $D_{1,2,3}$, where the classes connected by a dashed line are considered to be the same class (even though named differently). Each of the three diagrams is consistent, and each pair of them is consistent, but taken together the three diagrams are inconsistent: there is a cycle in the inheritance chain.

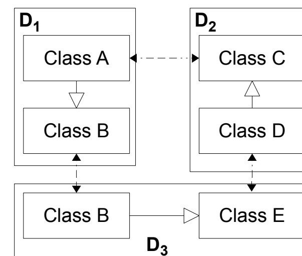

Fig. 1. Three globally inconsistent models

The example shows two phases in checking global consistency. First, we need to specify the models' overlap. For models like code and UML class diagrams extracted from code, we may know their overlap by matching the elements by name. But for models in the conceptual stage, we cannot deduce their overlap

automatically. For example, an entity "Person" created by a business analyst and a table "Employee" existing in a legacy database may refer to the same concept despite their different names. Moreover, there are cases when elements in different models are related but their relationship cannot be specified by direct linking and we need something more intelligent. For example, Fig. 2 shows two models that present basically the same information but structure it differently. Whatever means are used for specifying the overlap, in the second phase we need to check the global consistency of the system (= models + overlap).

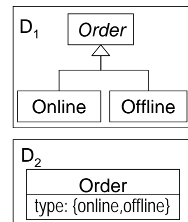

Fig. 2. Indirect correspondence

Sabezadeh et al.[2] proposed to check global consistency of homogeneous models by their merging. The models' overlap is specified by a *correspondence diagram*: a set of auxiliary models and mappings "in-between" the local model, which declare some elements in different local models as being actually the same. Then all local models are merged into one model modulo the correspondence, i.e., elements declared to be the same in the correspondence diagram become one element. Finally, consistency of the merged model is checked against the constraints declared in the metamodel. Thus, verifying global consistency amounts to checking consistency of a single model. However, the approach was developed for the case of homogeneous models only, and indirect overlaps (like shown in Fig. 2) were not considered.

The goal of the paper is to adopt the *consistency-checking-by-merging (CCM) idea* for the heterogeneous situation. A straightforward solution could be, first, to merge all involved metamodels so that all local models become instances of the same global metamodel; then we can merge these instances and check the result wrt. the constraints in the global metamodel. Though theoretically possible, in practice this approach leads to dealing with huge models and metamodels resulting from the merge, which is cumbersome and not effective. We present another approach in which merging metamodels is significantly reduced to an unavoidable minimum, and merging models is reduced to only merging their relevant parts. Briefly, we find common views between metamodels, project related models to spaces of instances (*overlaps*) determined by those views, and then apply the CCM approach to each of the homogeneous sets of projections.

Realization of the approach requires several challenging issues to be solved: type-safe model matching, specification of indirect overlap between metamodels, inter-metamodel constraints, and constraints over the entire schema of metamodels and their overlaps. We will discuss these issues in more detail in Section 2.2 after we briefly outline the basics of CCM-approach in Section 2.1. Section 3 describes our main techniques with simple examples. In Section 4 we abstract the examples and sketch a much more general framework. Section 5 presents a brief survey of approaches to heterogeneous multimodeling, and highlights the advantages of our framework. Section 6 concludes.

The present paper is an extended version of our MDI'2010 Workshop paper [3]. It presents a new issue of consistency between correspondence spans, and a

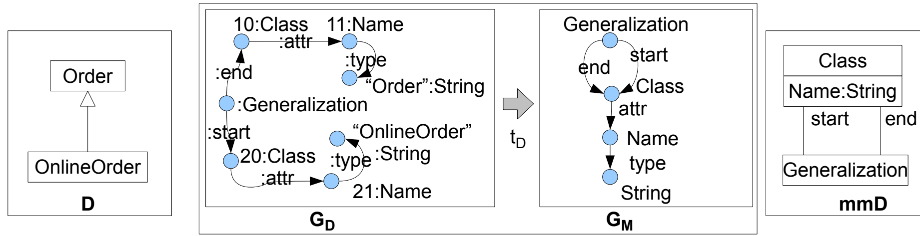

Fig. 3. Graph representation of a UML class diagram

new survey of approaches to heterogeneous multimodeling. Description of our formal framework is omitted due to space limits but Section 4 presents a rough sketch of the ideas.

2 From homo- to heterogeneous multimodeling

2.1 Background: Homogeneous overlap and consistency

We briefly review the basics of the CCM-approach, and also show how to manage conflicts between values.

Software models are typed graphs. We follow the approach to metamodeling developed by the graph transformation community, and treat models as typed graphs. A metamodel is a pair $M = (G_M, C_M)$ with G_M a graph and C_M a set of constraints. A model (M's instance) is a graph *typed* over M, i.e., a pair $D = (G_D, t_D)$ with G_D a graph (typically much bigger than G_M) and $t_D\colon G_D \to G_M$ a graph mapping (which preserves the incidence relationship between arrows and nodes) such that all constraints in set C_M are satisfied.

For example, Fig. 3 shows how to represent a UML class diagram D as a typed graph $t_D\colon G_D \to G_M$ with G_M being the graph representing a simple metamodel *mmD* for class diagrams. Classes, attributes, primitive values and generalization relations are represented as nodes; their relationships are captured by arrows. The value of mapping t_D at element $e \in G_D$ is given after colon, e.g., expression "10:Class" means $t_D(10)$=Class for node 10; identifiers of arrows are omitted but their types are kept.

Any UML class diagram can be represented by a typed graph as above but not the converse. To ensure that a typed graph is a correct diagram, constraints must be declared and added to the metamodel. Examples of constraints are (C1) a class has only one name; (C2) a class has only one parent class; (C3) classes with stereotype 'singleton' are instantiated with at most one object.

Matching models via spans. Suppose two business analysts have independently built two UML diagrams, D_1 and D_2 in Figure 4. To check their global consistency, we first need to specify overlap between the diagrams. Suppose we know that class 'OnlineOrder' in diagram D_1 and class 'Order' in D_2 refer to

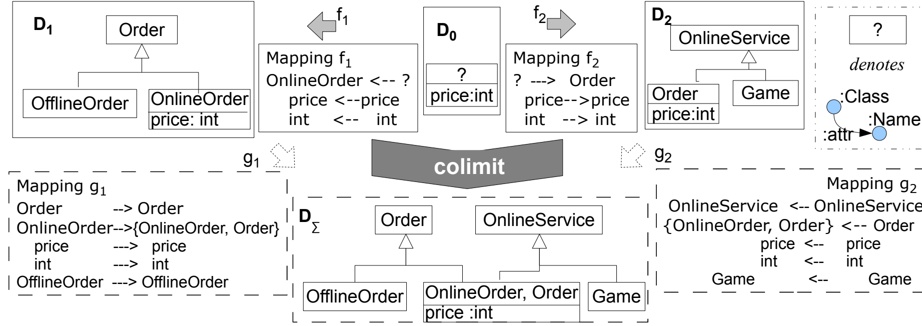

Fig. 4. Homogeneous Model Matching. Frames of models provided by the user are solid, those computed automatically are dashed; user-defined mappings are shaded, computed mappings are blank.

the same class of objects, and their 'price' attributes refer to the same attribute. We could write the following two informal equations: (E1) OnlineOrder@D_1 = Order@D_2; (E2) price@D_1 = price@D_2.

These equations conform to the type system of class diagrams: we match a class to a class and an attribute to an attribute. Hence, we can represent the set of equations by a class diagram D_0 shown in the middle of Fig. 4 equipped with two functions $f_i\colon D_0 \to D_i$, $i = 1, 2$, mapping "equations" to their left and right terms resp. Formally, f_1 and f_2 are graph mappings which map nodes to nodes and arrows to arrows so that their incidence is preserved. The question mark indicates that the name of the class is unknown and the corresponding Name-slot is empty (see the fragment in the top-right corner of the figure). Thus, equation (E1) encodes two formal equations (for classes and Name-slots) and (E2) gives three (equating, in addition, two string values).

We call a pair of mappings with a common source a *(binary) span*. The source (model D_0) is called the *head* of the span, mappings f_1, f_2 are *legs* and their targets (models D_1, D_2) are *feet* (these names are borrowed from category theory). Thus, an overlap of two homogeneous models is specified by a *correspondence* span over the same metamodel; for n models we need an n-ary span with n legs and feet. Note that the span pattern allows us to record inconsistencies and keep them for future resolution according to the *living with inconsistencies* paradigm [4]. A precise formalization and details can be found in [5, Section 3].

Merging and conflicts. After specifying the overlap by a correspondence span, we merge two models into one and check whether it satisfies all constraints declared in the metamodel.

The merge procedure consists of two parts. We first disjointly merge the graphs underlying the models, and then glue together elements declared to be the same by the span. The result is shown as diagram D_Σ in Fig. 4, in which the merged graph has five rather than six class nodes because of gluing. Class named {`OnlineOrder,Order`} has one Name-slot because the two local slots were

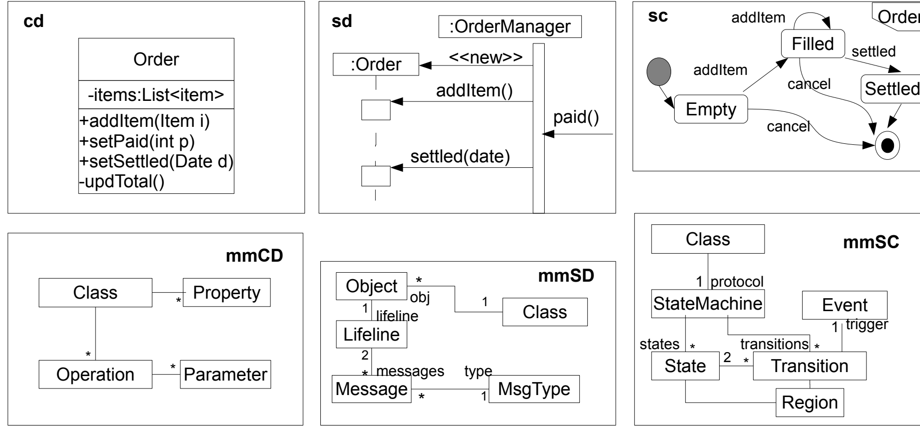

Fig. 5. Motivating example: mmXX is the metamodel of model xx

glued, but this slot holds two names since they are not (and cannot be) equated in the head. Besides graph D_Σ, merging also produces two graph mappings $g_i : D_i \to D_\Sigma$ that show how the local models are embedded into the merge.

The merge procedure is fully automatic and can be precisely formalized in terms of the *colimit* operation developed in category theory. A detailed explanation and examples of how colimit works can be found in [6] or [5]. It follows from general properties of colimit that the merged graph G_{D_Σ} is correctly typed over graph G_M (with M denoting the metamodel of class diagrams).

To make reading figures like Fig. 4 easier, we adopt the following notation. Frames of models provided by the user, and those computed automatically, are solid and dashed resp; user-defined mappings are shaded whereas computed mappings are blank.

After the merged graph is built, we can check whether it satisfies all constraints declared in the metamodel (say, with a checking tool). In our example, we find that constraints (C1) and (C2) specified above are violated.

2.2 The problems

Existing CCM-approaches [2] handle the homogeneous case well, but software models are often heterogeneous. For example, Fig. 5 presents three UML models of the same system developed independently by three teams: a class diagram cd, a sequence diagram sd, and a statechart sc (with their simplified metamodels). Since the models are developed independently, we need to specify their overlap and check the global consistency. However, the heterogeneity of the models gives rise to several new problems.

A) Type-safety is important for overlap specification. In the homogeneous situation, we allow only elements of the same type to be matched to ensure type safety. However, in heterogeneous cases different models are declared in different metamodels, and hence their elements have disjoint types. We need a new method to ensure type-safety in overlap specifications.

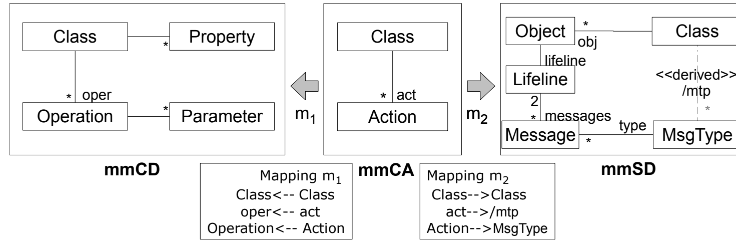

Fig. 6. Matching basic and derived meta-elements

B) Indirect overlap often occurs in heterogeneous multimodeling. For example, in class diagrams operations are linked to their owning classes. Such linking also exists but is implicit in sequence diagrams (through consecutive linking `Classes`, `Objects`, `Lifelines`, `Messages`, and `MsgTypes`). Hence, we cannot use direct matching to describe overlap between sets of Class-Operation links in class diagrams and Class-MsgType links in sequence diagrams.

C) Inter-metamodel constraints appear in heterogeneous multimodeling. For example, we may require that the interaction described by the sequence diagram should conform to the state machine described by the state chart. Such constraints regulate *interaction* of partial models, and hence are not captured by metamodels of any of them. Such constraints are inherently global and should be explicitly specified.

D) Metamodel inter-relations are crucial for heterogeneous multimodeling. "The metamodel" of a heterogeneous multimodel is a system of metamodels *together* with their relationships rather than a discrete set of isolated metamodels. We need a language for specifying systems of interacting metamodels.

3 Heterogeneous overlap and consistency by examples

In this section we incrementally introduce our approach. We will consider very simple examples addressing the four challenges.

3.1 Type-safety and indirect overlap

To ensure type-safety in heterogeneous case, we first need to know which types are safe to be matched. We get this information by asking the user to specify the overlap between metamodels first. For example, suppose in Fig. 5 we know that class `Order` together with methods `addItem`, `setSettled` in `cd` refer to the same elements as class `Order` together with message types `addItem, settled` in `sd`. To match these elements, we first match their metamodels, `mmCD` and `mmSD`, as shown in Fig. 6. Since metamodels are graphs, we can match them as homogeneous models. We state that metaclasses `Class@mmCD` and `Class@mmSD` refer to the same concepts, and `Operation@mmCD` and `MsgType@mmSD` are also "the same".

However, as we described in the previous section, there is also an indirect overlap between the metamodels: operations and message types are both related to classes, but operations are directly related by an association while message types are indirectly related via four associations. To declare this indirect overlap, we augment metamodel mmSD with a new element `mtp` (read "messageType") and specify how it is derived (e.g., in OCL):

```
context Class
inv: self.mtp=self.objects.lifeline.messages.type.
```

Now we declare the sameness of associations oper@mmCD and mtp@mmSD by placing association `act` into the head of the span as shown in Fig. 6, and defining $m_1(\texttt{act}) = \texttt{oper}$, $m_2(\texttt{act}) = \texttt{/mtp}$. The indirect overlap in Fig. 2 can be specified in a similar way. We first augment diagram D_2 with two derived subclasses of class Order (defined by the respective two queries), and then declare their sameness with the corresponding classes in diagram D_1.

After we have the overlap of metamodels, we can match models type-safely. Consider again the span we declared. We may consider the head of the span mmCA as a view on both models, and the two legs m_1 and m_2 as view definitions. Then the view definitions can be executed on models. For example, view definition m_1 : mmCA $\rightarrow$ mmCD can be executed for any instance of mmCD (i.e., for any class diagram) by extracting its mmCA-portion and its respective retyping. A concrete view execution is shown in Fig. 7, where class diagram cd shown in left upper corner is translated into diagram cd′ typed over metamodel mmCA. We write $\mathsf{cd}' = \mathsf{get}^{\mathsf{m}_1}(\mathsf{cd})$ with $\mathsf{get}^{\mathsf{m}_1}$ denoting the operation of view execution (*getView*) determined by view definition m_1 (in figures we omit the superscript). We will also say that model cd is *projected* into the *overlap space* mmCA, and call model cd′ the mmCA-*projection* of cd. Note also that *getView* not only produces cd′ but also the traceability mappings $\overline{\mathsf{m}_1}\colon \mathsf{cd}' \rightarrow \mathsf{cd}$.

Similarly, sequence diagram sd in the top right corner of Fig. 7 is translated into diagram $\mathsf{sd}' = \mathsf{get}^{\mathsf{m}_2}(\mathsf{sd})$ also typed over mmCA, along with its traceability mapping $\overline{\mathsf{m}_2}$. (This translation involves execution of the OCL-query specified above). Since both views are instances of the same metamodel, we can type-safely build a span $(\mathsf{ca}_1, f_1, f_2)$ to match them and check consistency. This span and the corresponding merge (colimit) are shown in the middle part of Fig. 7, and the two models are consistent with respect to the constraints in mmCA.

3.2 Inter-metamodel constraints

So far we only checked the constraints declared in the head of the correspondence span (mmCA in our examples). These constraints are common for both feet metamodels (mmCD and mmSD). However, there may be important constraints residing in neither of the feet metamodels. For example, traces of actions exhibited by a sequence diagram must conform to the state machine specified by the corresponding statechart. We will denote this constraint by ***t♯sm*** meaning "Traces are to conform to the StateMachine". Since constraint ***t♯sm*** involves elements of both metamodels, mmSD and mmSC, it cannot be declared in either of them. Hence, a new metamodel in which ***t♯sm*** could be specified has to be

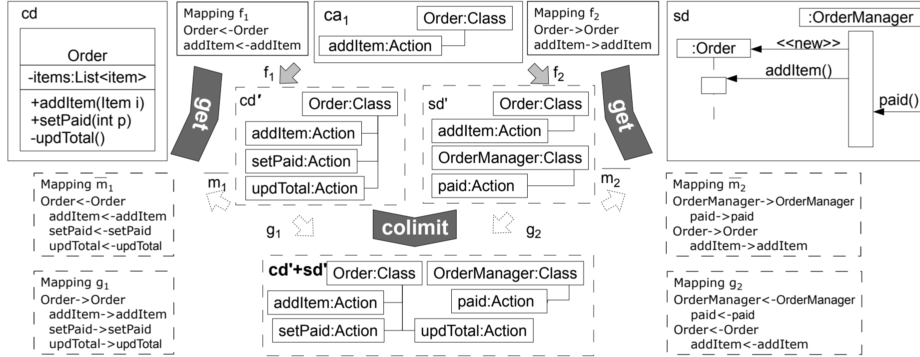

Fig. 7. Matching basic and derived elements (see Fig. 6 for view definitions)

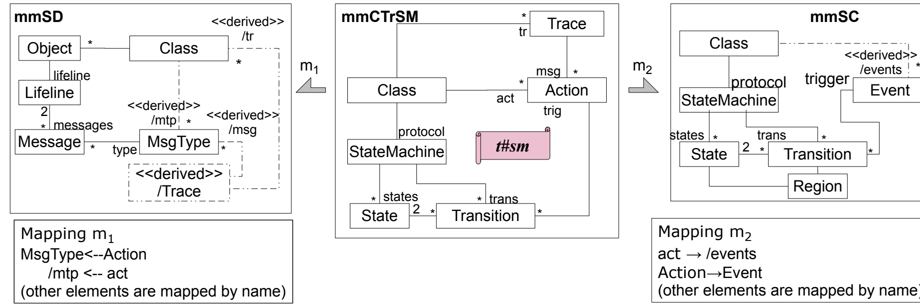

Fig. 8. Specifying inter-metamodel constraints

built. Below we first show how to build such a metamodel; then we show how to project partial models sd and sc to the space of this metamodel instances, in which projections can be matched, merged and checked against ***t♯sm***.

To declare ***t♯sm***, we need a metamodel encompassing metaclasses for Classes, Traces (sequences of actions), StateMachines, and related notions: States, Transitions, Events as specified by metamodel mmCTrSM in the middle of Fig. 8. The upper half of this metamodel is "taken" from the sequence diagram metamodel mmSD as specified by mapping m_1 in Fig. 8. Note that m_1 maps class `Trace`@mmCTrSM to derived class `/Trace`@mmSD, whose instances are sequences of actions described by the sequence diagram and hence can be computed by a suitable query. The lower half of mmCTrSM is taken from the statechart metamodel mmSC as specified by mapping m_2 in Fig. 8 (and we again use derived elements). Having built metamodel mmCTrSM, we declare in it the constraint ***t♯sm*** with its intended semantics. We call the configuration $(m_1, \mathsf{mmCTrSM}, m_2)$ a *partial* span because mappings m_1 and m_2 are partially defined (on the upper and lower halves of mmCTrSM resp.). In Fig. 8 and other figures below, a semi-arrow head indicates partiality of the mapping.

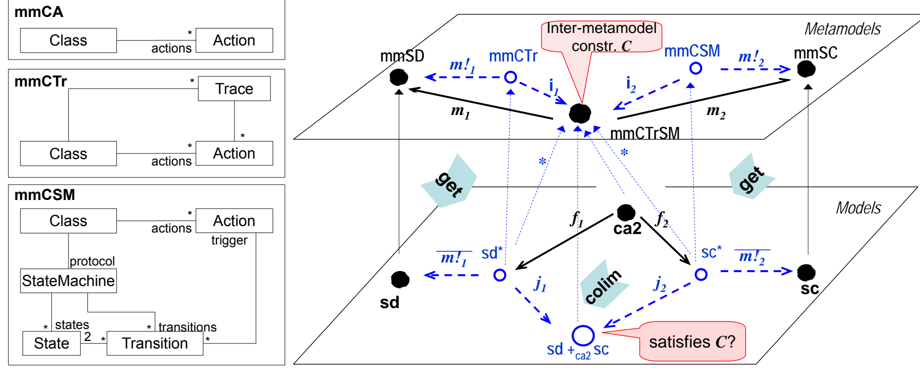

Fig. 9. Verifying inter-metamodel constraints

The next step is to project models sd and sc to the metamodel mmCTrSM. We cannot directly execute view definitions m_1, m_2 because they are partial, but we can execute them in three steps.

Step 0. We explicitly specify the domains mmCTr and mmCSM of mappings m_j on which they become totally defined mappings $m!_j$ ($j = 1, 2$; see Fig. 9); inclusion mappings i_j embed the domains into the head of the span.

Step 1. Total view definitions $m!_j$ are executed for models sd and sc and produce views sd* and sc* over metamodels mmCTr and mmCSM resp.

Step 2. As the two latter metamodels are included into mmCTrSM, we may consider their instances as "partial" instances of mmCTrSM. Formally, we compose typing mappings of models sd*, sc* with inclusion mappings i_1, i_2 and get new typing mappings into mmCTrSM (marked by $*$ in Fig. 9).

The three steps are performed automatically and may be hidden from the user, for whom operations get^{m_1} and get^{m_2} appear as if mappings m_j were ordinary total view definitions.

Now we have two models sd* and sc* over the same metamodel mmCTrSM. To finish consistency checking, the user must match the models and build a correspondence span, say, $(f_1, \mathsf{ca2}, f_2)$. The head is denoted by ca2 because it is an instance of metamodel mmCA built in Section 4.2 (it can be formally proved). After that, models are automatically merged modulo the span and checks the result against the constraints in mmCTrSM, including the inter-metamodel constraint $\boldsymbol{t\sharp sm}$. The right half of Fig. 9 specifies the entire procedure: data provided by the user are shown with bullet nodes and solid arrows (and are black), data automatically computed are shown with blank nodes and dashed arrows (and are blue). Note that span $(f_1, \mathsf{ca2}, f_2)$ is a part of the multimodel.

3.3 Metamodel inter-relations

We consider our full example with three models, cd, sd and sc.

First we build a ternary span $(\mathsf{mmCA}, m_1, m_2, m_3)$ specifying "the sameness" of the concepts of operation, message and transition in the respective metamodels

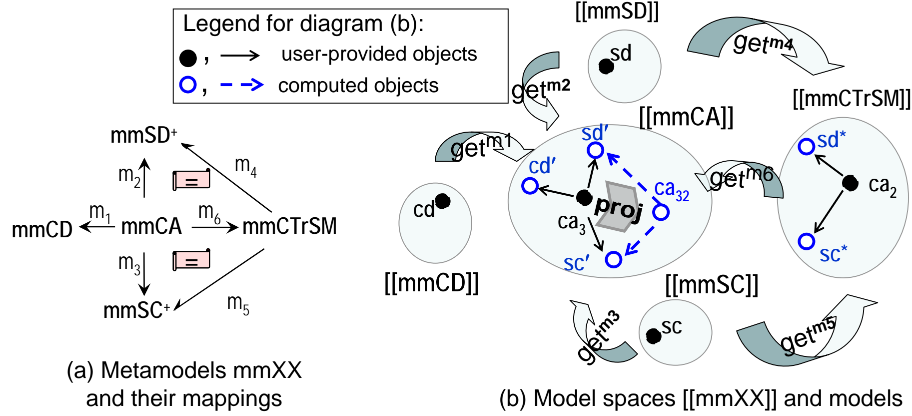

Fig. 10. Consistency checking of the example in Fig. 5

as shown in Fig. 10(a); superscript '+' near a target metamodel indicates that it is augmented with derived elements defined by queries. Ternary span mmCA is a straightforward extension of binary span mmCA built in Section 3.1 with a new leg towards mmSC. Then we turn to models. We project the three models to head mmCA (see Fig. 10(b)), match projections ca′, sd′, sc′ with a ternary correspondence span ca3, merge projections modulo ca3, and finally check the merge against the constraints declared in mmCA.

In a similar way we check consistency of models sd and sc wrt. the inter-model constraint ***t♯sm*** declared in mmCTrSM as explained above (span ca_2 describes correspondences between projections sc^* and sd^*). However, now a new aspect of global consistency checking appears: we need to check that model correspondence spans ca_3 and ca_2 are consistent between themselves.

An important property of the metamodel schema in Fig. 10(a) is commutativity of the two triangle diagrams (denoted by "=" labels):

$(=)_{\mathsf{m}}$ $\quad \mathsf{m}_6;\mathsf{m}_4 = \mathsf{m}_2$ and $\mathsf{m}_6;\mathsf{m}_5 = \mathsf{m}_3$.

Because view execution and retyping are compatible with metamodel mapping composition, we have commutativity for view execution mappings as well:

$(=)_{\mathsf{get}}$ $\quad \mathsf{get}^{\mathsf{m}_4};\mathsf{get}^{\mathsf{m}_6}=\mathsf{get}^{\mathsf{m}_2}$ and $\mathsf{get}^{\mathsf{m}_5};\mathsf{get}^{\mathsf{m}_6}=\mathsf{get}^{\mathsf{m}_3}$.

Hence, the mmCA-views of xx^*-models must be equal to the respective xx'-models. Now we can check the consistency between spans. We first derive a binary projection ca_{32} of the ternary span ca_3, which relates sd' and sc'. Then we check whether the mmCA-view of the span ca_2 is equal to ca_{32}.

The simple example above shows how local model interaction is governed by the multimodel schema specifying metamodels' inter-relationships. The example also demonstrates that N-ary multimodeling may exhibit sufficiently complex metamodels schemas bearing their own constraints like commutativity.

4 Making multimodeling precise: A general framework

A key message of the paper is that a multimodel is not just a set of models. A multimodel is a set of *base* models *and* a structure of auxiliary models and model mappings specifying *correspondences* between base models. For instance, the multimodel of our example in the previous section consists of three models (cd,sd,sc) *and* two inter-model spans (ca_3, ca_2) shown in Fig. 10(b). Respectively, the metamodel of a multimodel is a graph consisting of base metamodels *and* a system of spans specifying their overlap like shown in Fig. 10(a); we call it the *metamodel schema*.

In a nutshell, a heterogeneous multimodel is a pair $(\mathcal{A}, \mathcal{C})$ with $\mathcal{A} = \{A_1{:}M_1..$ $..A_k{:}M_k\}$ a family of base models A_i over their metamodels M_i, and $\mathcal{C} = \{C_1{:}O_1..$ $..C_l{:}O_l\}$ a system of model correspondence spans C_j over a system of (heads of) spans O_j specifying metamodel overlap. In other words, the correspondence part of a multimodel is a network of auxiliary models and model mappings in-between models A_i, which resides over the respective network of auxiliary metamodels and mappings in-between metamodels M_i. The two-level structure of the overlap specification is essential: models may overlap only via paths declared in the metamodel schema.

Our formal framework [7] provides a detailed description of the sketch above. The three basic ingredients are (a) metamodels and their mappings, (b) models and their mappings, and (c) a mechanism of model translation from one metamodel to another. A (minimal in a sense) mathematical framework integrating these constructs turns out close to the *institution* theory [8] — a framework for model translation developed in mathematical logic and model theory. In more detail, the concept of *abstract multimodeling framework* described in [7] is a variant of the so called *liberal institutions*, which have two translation mechanisms corresponding to our view computation and retyping. The framework is fairly abstract: no details are given on what conformance of a model to a metamodel is, or how the view mechanism is realized. Nevertheless, the notions of heterogeneous multimodel and its consistency can be well defined and give rise to the corresponding algorithm for global consistency checking.

To bridge the gap between the abstract framework and practical applications, the notion of a *concrete multimodeling framework* is also defined in [7]. For a concrete framework, conformance of models to metamodels is realized via typing mappings (and retyping plainly amounts to mapping composition), and the view mechanism is realized via an algebra of query operations. A wide class of multimodeling systems appearing in practice are instances of concrete frameworks. Any concrete framework gives rise to an abstract framework, and thus the general algorithm of global consistency checking can be applied.

5 Related work and discussion

Approaches to heterogeneous multimodeling can be roughly divided into *global* and *local*. For the global approaches, 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 (including ours). Another crucial dimension of multimodeling is how correspondences between local models and their inter-relationships are specified. Below the space of existing approaches is discussed in more detail.

Global approaches. We distinguish two main types.

1) Monitoring satisfiability of consistency rules. This is the most direct global approach to consistency checking. All local models are considered as partial instances of some all-embracing global model given a priori, e.g., *System model* in [9] or the entire UML model (if UML modeling is treated as suggested by OMG). Inter-model consistency is given by *rules* specified in a special language "understanding" all local models. For this goal, local models are translated into an expressive common formalism, e.g., FOL in the well-known *Viewpoints* framework [10], XMI in *xlinkit* [11], description logic in [12], and again FOL in Egyed's framework [13]. Configuration of model overlap (which may be very intricate as our examples show) is thus flattened and hidden in arrays of formulas. As a result, the approaches mainly handle cases with simple overlap structures, e.g., binary overlaps with elements matched by names.

2) Consistency checking via merging. Close relations between consistency and model merging are noticed in [14] for behavioral, and in [2] for structural modeling. 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. Much work in this direction is done in databases in the context of *view integration*, where they work mainly with enhanced ER-diagrams [15] or similar but more expressive formalisms [16]. A serious limitation of this work is that only the homogeneous case is considered because so far it was unclear how to merge heterogeneous models.

To manage heterogeneity, local models can be translated into an a priori given common expressive formalism (e.g., *generalized sketches* [17] or graph transformation systems [18]), where they are merged. A more intelligent approach is to *build* a minimal common formalism by merging together all local metamodels. Different versions of this idea have been elaborated in the area of model composition [19]; a survey can be found in [20]. Of course, composition of local metamodels requires correspondences between them to be explicitly specified. Usually only binary cases are considered, but [17, 18, 2] address also the general N-ary case by using the colimit operation.

Local approaches. Although the idea of local consistency checking seems intuitive, we are not aware of its practical realization. A partial reason for this may be that an abstract general formulation of the framework is not easy (compare our concrete examples in Section 4 and their abstract description in [7]). The problem is currently being investigated by the Algebraic Specification community within the institution framework [8]. Models are translated into theories in suitable institutions, and relationships between the latter are specified by spans (or cospans) of institution comorphisms (resp., morphisms) [21]. The community is experimenting with different types of structures specifying institution

overlaps, and a recent paper [22] uses mixed pairs (comorphism, morphism) to relate two institutions. It is not clear how this mixed setting can be extended to the multi-ary situation.

A fundamental distinction between these and our frameworks is that they do not consider derived elements in correspondence specifications. It makes the theory much simpler but much less expressive (and inapplicable to practically interesting situations we considered in the paper). Another fundamental distinction is that they consider local models consistent if their projections to the overlap are equal (or one is a subset of the other), but matches between projections are not considered. In contrast, in our framework model matches are an integral part of the multimodel. Other distinctions are (a) they consider only binary correspondences, (b) do not work with inter-metamodel constraints, and (c) treat consistency semantically (the set of instances is not empty) rather than syntactically (as in our framework). However, if the institution satisfies the corresponding completeness theorem, syntactic and semantic consistencies coincide. Also, we do not translate metamodels into theories: for us metamodels *are* theories, and model translation is given by the view execution mechanism.

Correspondences via spans. For local and global-2 approaches, explicit specification of inter-model correspondences is a central issue, and different types of notation and techniques were developed [23]. A distinctive feature of our approach is that the set of correspondences is reified as a special model endowed with projection mappings — a span. This is a standard categorical idea, which was repeatedly employed in homogeneous multimodeling frameworks based on category theory, eg, [24, 17, 18, 6, 25]. Independently, the same idea of reifying correspondences by a model was discovered in work on model management in databases [26, 16].

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 (e.g., Fig. 2). Such correspondences are usually specified by *correspondence rules* [23] or *expressions* [26] attached to nodes reifying correspondences. When such annotated spans are composed, it is not clear how to compose the rules — the importance and difficulty of this problem was stressed in [26]. In our approach, the problem is solved with specifying indirect correspondences by equations involving derived elements, then composition amounts to term substitution (see [5, 27] for examples and details). Moreover, the use of derived elements allows us to specify structural conflicts between models uniformly by equations; e.g., all structural conflicts considered in [16] can be managed in this way [28].

6 Conclusion

The paper describes a general approach to global consistency checking of heterogeneous multimodels. It is based on finding common views between metamodels of the models involved, projecting all models to these views, merging projections and checking the result against the constraints specified in the view. The

approach gives rise to a novel framework for heterogeneous multimodeling, in which a network of interrelated metamodels — the metamodel schema — plays the central role.

The framework has a number of advantages. First, heterogeneous consistency checking is reduced to homogeneous with a minimal amount of metamodel merging; the latter is unavoidable if we want to treat inter-metamodel constraints yet we work as locally as possible. Second, the framework is applicable to a wide class of models and metamodels satisfying not too restrictive conditions. Third is the adaptability of the framework to the *living with inconsistencies* paradigm [4]: conflicts between models can be recorded in the heads of the correspondence spans and resolved later. Forth, heterogeneous multimodeling becomes directly related to the institution theory and hence to a source of important mathematical results about interrelation of logical theories and their models.

However, the approach still needs practical, and in part also theoretical, validation. On the practical side, the main question is how effectively a multimodeling tool based on the framework could be implemented. On the theoretical side, the cornerstone of the approach is a default assumption that our "as local as possible" consistency checking is equivalent to consistency checking via building a global metamodel (global-2 approaches). There are strong formal arguments justifying this assumption but an accurate proof is still to be completed.

Another important theoretical line of future work is to develop a useful classification of heterogeneous multimodels. We may classify multimodels by the type of their metamodel schema: whether it is a plain collection of spans, or there are spans over spans over spans, or perhaps even more complex configurations. Types of mappings in the metamodel schema are also essential: whether they are plain projections or complex views involving non-trivial queries. Complexity of queries involved in the metamodel schema of a multimodel is its important property, and many useful results can be found in the database literature. Defining multimodeling in abstract mathematical terms [7] would allow useful interaction of the two fields.

Acknowledgement. We are grateful to the organizers of the MDI'10 Workshop for encouragement and the invitation to prepare this paper. Thanks to Antonio Vallecillo for pointing out several important related papers. Special thanks to Michał Antkiewicz for valuable discussions of multimodeling.

Financial support was provided by the Ontario Research Fund.

References

1. Egyed, A.: Instant consistency checking for the UML. In: ICSE. (2006) 381–390
2. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: RE. (2007) 221–230
3. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: First Int. Workshop on Model-Driven Interoperability, MDI'2010, ACM Press (2010) 42–51
4. Balzer, R.: Tolerating inconsistency. In: ICSE. (1991) 158–165
5. Diskin, Z.: Model synchronization: mappings, tile algebra, and categories. In R. Lämmel et al., ed.: Postproceedings GTTSE 2009. Volume 6491 of LNCS., Springer

6. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. **11**(3) (2006) 174–193
7. Diskin, Z.: Towards a formal semantics for consisterncy of heterogeneous multimodels. Technical Report 2010-07, The University of Waterloo (2011) http://gsd.uwaterloo.ca/node/330.
8. Mossakowski, T., Tarlecki, A.: Heterogeneous logical environments for distributed specifications. In: WADT. Volume 5486 of Springer LNCS. (2009) 266–289
9. Broy, M., Cengarle, M., Rumpe, B.: Semantics of UML — towards a system model for UML: The structural data model. Technical Report TUM-IO612, Techniche Universität München (2006)
10. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: ICSE. (2003) 676–683
11. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: ICSE. (2003) 455–464
12. Straeten, R.V.D., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML Models. In: UML. (2003) 326–340
13. Egyed, A.: Fixing inconsistencies in UML design models. In: ICSE. (2007) 292–301
14. Easterbrook, S.M., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: ICSE. (2001) 411–420
15. Spaccapietra, S., Parent, C.: View integration: A step forward in solving structural conflicts. IEEE Trans. Knowl. Data Eng. **6**(2) (1994) 258–274
16. Bernstein, P., R.Pottinger: Merging models based on given correspondences. In: VLDB. (2003)
17. Cadish, B., Diskin, Z.: Heterogenious view integration via sketches and equations. In: ISMIS. (1996) 603–612
18. Engels, G., Heckel, R., Taentzer, G., Ehrig, H.: A combined reference model- and view-based approach to system specification. Int. Journal of Software and Knowledge Engeneering **7** (1997) 457–477
19. Bézivin, J., Bouzitouna, S., Fabro, M.D., Gervais, M., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.: A canonical scheme for model composition. In: ECMDA-FA. (2006) 346–360
20. Vallecillo, A.: On the combination of domain specific modeling languages. In: ECMFA. (2010) 305–320
21. Wirsing, M., Knapp, A.: View consistency in software development. In: RISSEF. Volume 2941 of LNCS., Springer (2002) 341–357
22. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What is a multi-modeling language? In: WADT. Volume 5486 of LNCS., Springer (2009) 71–87
23. Romero, J., Jaen, J., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: EDOC, IEEE Computer Society (2009) 163–172
24. Fiadeiro, J.L., Maibaum, T.S.E.: Interconnecting formalisms: Supporting modularity, reuse and incrementality. In: SIGSOFT FSE. (1995) 72–80
25. Liang, H., Diskin, Z., Dingel, J., Posse, E.: A general approach for scenario integration. In: MoDELS. (2008) 204–218
26. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR'2003. (2003) 209–220
27. Diskin, Z.: Mathematics of generic specifications for model management. In Rivero, Doorn, Ferraggine, eds.: Encyclopedia of Database Technologies and Applications. Idea Group (2005) 351–366
28. Diskin, Z., Easterbrook, S., Miller, R.: Integrating schema integration frameworks, algebraically. Technical Report CSRG-583, University of Toronto (2008) http://ftp.cs.toronto.edu/pub/reports/csrg/583/TR-583-schemaIntegr.pdf.